Establishing Maintainability in Systems Integration: Ambiguity, Negotiations, and Infrastructure

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

Abstract

This paper investigates how maintainability can be established in system integration (SI) projects where maintainers have no direct access to the source code of the third-party software being integrated. We propose a model for maintainability in SI focusing on post-release activities, unlike traditional maintainability models where focus is on pre-release activities. Our model describes maintainability as a process characterized by ambiguity and negotiation that is supported through an infrastructure of debugging and coordination tools. Further, we describe how the process going from a software failure to establishing the fault causing the failure can be managed in SI. The results presented in this paper are based on observations from an ethnographic study of the Gentoo open source software (OSS) community, a large distributed volunteer community of over 320 developers developing and maintaining a software system for distributing and integrating third-party OSS software packages with different Unix versions.

1. Introduction

It has been repeatedly established over the past 30 years that more than half of the total life-cycle cost of software systems goes into software maintenance activities. The figures vary between 50 to 80 percent of the total life-cycle cost [6]. This research indicates that the maintenance burden has been increasing over the decades rather than decreasing. To face this challenge, *maintainability* has been proposed as a software quality measure. This measure is used to assess how easy it is to maintain a system and what decisions to make in design of a system to limit the maintenance costs. Existing research on maintainability builds on the premise of application software that is maintained by a single team of developers with full access to and control over the source code. However, with increasing attention on systems integration (SI) through component-based development [5], Web services [22], and information and enterprise systems integration [14], this may no longer be a valid premise. A number of distinguishing characteristics of SI diverge from application software: systems integrators usually have limited or no access to the source code of the software being integrated, and control over the development and maintenance of the software being integrated is carried out by numerous third-party organizations [11]. Given these differences, we ask: **how can maintainability be established in SI?**

In this paper, we seek to explore a possible solution to this problem; a solution that rests upon the premise of software maintenance as knowledge-intensive work. By studying the activities involved with reporting software failures and determining their related faults, we propose that corrective maintenance in SI unfolds within an environment of *ambiguity* [1]. Ambiguity is an uncertainty where the correct meaning of a phenomenon cannot be established given sufficient or appropriate information. Instead, ambiguity involves uncertainties that cannot be resolved or reconciled due to the absence of agreement on boundaries, clear principles, or solutions. Ambiguity means that multiple meanings or several plausible interpretations of the observed phenomenon exist, and their meaning can only be established through *negotiation*. The process of establishing certain interpretations of ambiguous phenomenon as stable scientific facts has been a primary concern within the field of science studies. In these accounts, this process is seen as unfolding within an *infrastructure* of experimental tools, scientific artifacts, social interaction, and practices [15]. It is an infrastructure of scientific facts; the behind-the-scenes, messy or boring items that form a crucial part of how facts are made.

Building upon the notions of ambiguity, negotiation, and infrastructures, we propose that *maintainability can also be understood as a function of the external environment within which the software is being maintained*. Maintainability is a function of the infrastructure of tools used during maintenance, the texts produced by these tools, knowledge about the system embedded in the tools, and tools for supporting and coordinating interaction between developers. This supplements existing models that focus on maintainability as a function of characteristics of the application software. The proposed explanation is based on an empirical study of maintenance work in a large-scale open source software (OSS) integration project. OSS is well suited for studying software maintenance, as OSS development is often understood as a perpetual cycle of perfective and corrective maintenance [20].

Limiting our inquiry to the issue of maintainability in connection with corrective maintenance in SI, we study the activities involved with reporting software failures and determining the related fault. Through a detailed narrative analysis of these activities, we propose a model for the corrective maintenance process that supports our suggestion for establishing maintainability in SI.

The rest of the paper is organized as follows. Section 2 reviews existing research on maintainability and approaches to establishing maintainability during pre-release activities. Section 3 describes the research methods employed and the materials collected during our field study. Section 4 describes a detailed narrative analysis of the activities with reporting software failures and determining the corresponding faults. Section 5 concludes the paper by discussing our findings in relation to establishing maintainability in SI.

2. Related work

Intended as an indicator of the costs of maintaining a software system, maintainability can be broadly defined as the ease with which a software system can be understood and modified [10]. By making the software more maintainable, i.e. increasing its maintainability, organizations should be able to reduce the maintenance effort and free needed resources for more new system development. Maintainability can be viewed from different perspectives. In this section we presents two of these:

- establishing and assessing maintainability using software quality models; and
- making a system maintainable by using design techniques when creating the software architecture of the application

We then conclude the section by discussing the issue of maintainability in relation to OSS development.

2.1. Quality-based approaches

McCall [18] provides an overall description of approaches to developing software based on software quality frameworks. At the outset of a software development effort quality factors are identified based on the specifics of the software being developed. Maintainability is one such quality factor. Once the important factors are identified, they are specified as requirements of the systems development by providing their definition, identifying supporting software attributes, and providing measurements to assess their attainment. The software development is then periodically measured in a quantitative fashion to assess if the software product is capable of meeting its identified requirements. Based on this assessment of the software product's quality, decisions are made as to efforts needed to improve the software product. This process is repeated until the quality requirements, in this case the requirements for maintainability, are met and the product can be released.

Several approaches to assessing the software product's maintainability have been proposed. McCall [18], Martin & McClure [17], Boehm et al. [4], and ISO9126 define maintainability as a quality factor in their quality models. Wherein McCall limits maintainability to include only corrective maintenance, both Boehm et al., Martin & McClure, and ISO9126 provide definitions that encompasses both corrective, perfective, and adaptive maintenance. Boehm et al. defines maintainability to include the quality criteria testability, understandability, and modifiability. Martin & McClure argues for an expanded view of maintainability, arguing that its definition needs to be expanded with a high degree of reliability, portability, efficiency, and usability in addition to the attributes provided by Boehm et al. Landing on the middle ground, ISO9126 defines maintainability as analyzability, changeability, stability, and testability. In all of the above models, the quality criteria are broken into a set of metrics for measuring code characteristics.

2.2. Architecture approach

In the software architecture domain, software maintainability is a quality of the end-system the developer can obtain by carefully choosing the correct structures and making the correct decisions when designing the system. Different terms are used to describe maintainability.

In Bass et al. [2], maintainability is described in terms of the quality attributes modifiability and testability. Modifiability describes the costs of changing the system. Typical changes can be both changes of functionality as well as changes of nonfunctional properties of the system like performance, availability, change of operating system etc. Testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. To obtain a high level of modifiability and testability in a system, the developers must consider both architectural and non-architectural aspects. The architectural aspects typically concerns *important design decisions* that affect the way the software is organized, structured and decomposed.

Non-architectural aspects typically concern implementation details, graphical layout of user-interfaces etc. Bass et al. use the term *architectural tactics* for important design decisions that affects the software architecture. Several such tactics have been collected over the years based on experiences from several software projects. Examples of tactics to obtain high maintainability involves recommended design guidelines for object-oriented systems like maintaining semantic coherence, hide information, restrict communication paths, use of intermediary, etc. There are also similar tactics to obtain a high level of testability in a system.

2.3. OSS and maintainability

The OSS development cycle have three distinguishing characteristics. First, source code is made available on the Internet, released early, long before all functionality is in place and faults have been eliminated. Second, by releasing the software early, developers around the world can contribute code, adding new functionality and improving the present functionality. This is often called parallel development [9]. Parallel debugging is the third characteristic of the development cycle, wherein failure reports and fixes are submitted to the project. This process has been characterized as a perpetual cycle of perfective and corrective maintenance.

Seeing OSS development as software maintenance, the question can be raised whether the success of OSS development can be explained by the software's maintainability? In determining the categories of maintenance work in two large OSS products, 53.4% of all changes to the source code of these products stem from corrective maintenance [21]. Given that the cost of corrective maintenance are at least an order of magnitude more expensive to fix than those found during testing [7], the question concerning OSS success and maintainability becomes even more pressing.

In measuring the maintainability index of five OSS projects, Samoladas et al. [20] finds that OSS code quality suffers from the very same problems observed in closed source software (CSS) projects. Maintainability deterioration over time is a common phenomenon in CSS, and they project that is reasonable to expect this as OSS products age, too. In a comparison of OSS and closed source software products, Paulson et al. [19] investigates the claim that OSS succeeds because of code simplicity. Measuring the overall project complexity, average complexity of all functions, and average complexity of functions added, they find that for all three metrics the OSS projects had higher complexity than the CSS projects. Similarly, they compare the perfective maintenance of OSS and CSS by measuring the growth rate of the projects. They find that OSS and CSS have similar growth rate. Albeit based on a limited population, the inference from combining the conclusions of Samoladas et al. and Paulson et al. is that the maintainability of OSS and CSS is mostly the same.

Paulson et al. also reports that faults are found and fixed more rapidly in OSS projects. Holding to the definition of maintainability as the ease with which a software system can be understood and modified, questions may be raised with basis in these findings as to how to understand maintainability? It seems that commonly used maintainability metrics do not correspond to the actual facts of maintainability as measured in ease of which software systems can be understood and modified.

3. Methods and materials

This paper reports on one of the authors' study of software maintenance in a large OSS community. The study is based on the view that to better understand software engineering, "it is imperative to study … software practitioners as they solve real software engineering problems in real environments" [16]. As such, the study has been conducted as ethnographic fieldwork, expanding on a growing body of ethnographic studies of software engineering practice. Ethnography is a research method where the researcher participates with the subjects being studied. Through longitudinal observations of naturally occurring activities, the researcher builds an increased understanding of the object under study. However, if we want to understand how software is developed in practice, it is important not to start out assuming what we want to explain. Therefore the ethnographer does not give any prior significance to particular features of practice. Giving primacy to the empirical data, ethnography is a systematic approach for reaching empirically validated conclusions.

In Section 3.1 we present the research setting. In Section 3.2 we present the data collection process. In

Section 3.3 the data analysis process is presented, and in Section 3.4 we discuss the validity of our findings.

3.1. Research setting

This paper reports on an ethnographic study of the Gentoo OSS community. As of March 2006 Gentoo is made up of over 320 developers distributed across 38 countries and 17 time zones. We use the term community here about those involved with Gentoo, as users play an important role in OSS development [9]. Enumerating the number of users in the community is difficult because there are no lists of purchased licenses or registered users available.

Gentoo is a large systems integration project. Broadly speaking, the Gentoo community develops and maintains a software system for distributing and integrating third-party OSS software packages with different Unix versions. The software is distributed in the form of installation scripts, one script for every supported version of each package distributed. As of March 30 2006 Gentoo distributes one or more versions of 8486 software packages, for a total of 23911 installation scripts. As well as integrating software for 5 different hardware architectures for the GNU/Linux operating system, the installation scripts can also integrate software with both the MacOS X, FreeBSD, and OpenBSD operating systems. Such heterogeneity is a defining characteristic of integrated systems [11].

In distributing software developed by other OSS projects, the development and maintenance of these packages are outside the control of the Gentoo community. Such autonomy is also a distinguishing characteristic of integrated systems [11], but also manifest a variety of human interests and activities. In defining largeness of software systems, Belady & Lehman [3] find variety to be a distinguishing characteristic. In terms of largeness, the software distributed is outside the scope of a single individual and require not only one group of people to develop and maintain the software, but numerous groups; both the Gentoo community developing and maintaining the installation script and the third party OSS communities who develop and maintain the software distributed. Furthermore, the installation scripts developed and maintained by the community is also outside the grasp of a single individual. Gentoo is organized into 124 teams, each responsible for a discrete set of installation scripts.

There are complex interactions between parts of Gentoo, both technologically and socially. Complex interaction is another characteristic of largeness. Technologically these interactions manifest themselves in the specific relations between different packages and that the same package is supported both on different hardware platforms and for different operating systems. This is made further complex by the introduction of virtual packages, identical functions that are provided in different packages. Socially, the complex interactions are not only between members of the Gentoo community or among the teams, but also in the interface between the Gentoo community and the OSS communities developing the software distributed by Gentoo.

So far, we have used the term Gentoo without any clear definition. This is done on purpose, as the term itself is ambiguous. The term has three meanings. First, it is used for talking about the Gentoo community of developers and users. Second, it is used about the Gentoo GNU/Linux distribution. Sometimes the term Gentoo Linux is used to specify this. Third, Gentoo is a software system for distributing OSS software packages for different Unix implementations. The distributed packages are developed by third-party OSS projects, and the Gentoo community develops and maintains installation scripts for these packages. These scripts are made available through a central repository.

The term Gentoo is ambiguous; it is particularly problematic to draw a clear boundary between Gentoo Linux and the Gentoo software distribution system. At the heart of Gentoo Linux is the Gentoo distribution system. Historically, however, the distribution system has grown out of the Gentoo Linux distribution. The term Gentoo is used interchangeable between the two, and often used by developers as a means to avoid drawing the problematic boundary between the two. Technically speaking, there are both installation scripts and other files distributed by the Gentoo distribution system that are particular to Gentoo Linux. However, most installation scripts distributed are not specific for the GNU/Linux distribution.

The lack of consensus on boundaries is a trait of ambiguity. Both variety and complex interactions produces unclear technological boundaries and ambiguity in the Gentoo software product.

3.2. Data collection

The first author conducted the ethnographic fieldwork. We therefore present this section in first-person view. Participant-observation is the primary method for data collection in ethnographic fieldwork [12]. In this study this meant that I observed the Gentoo developers online through dedicated Gentoo IRC channels, dedicated mailing lists, the Gentoo Web site, and Web-based front-ends for Gentoo's defect tracking system and revision control system. My participation included submitting and assisting in resolving bug reports, submitting installation scripts, as well as participating in a large restructuring effort of Gentoo's package manager. I used both Gentoo Linux and MacOS X with a Gentoo installation as operating systems on my workstations during the period of fieldwork. I made no formal interviews with participants in the Gentoo community, but conducted informal talks with participants on a regular basis to test my own informal theories.

Throughout the period, I made daily field notes [12]. In addition, the Gentoo IRC channels were logged to disk throughout the period of study; one file each day for each IRC channel totaling 1027 files. 71 documents were collected throughout the period and organized in a documentary database. I also surveyed online data sources that provide static data. These include the Gentoo bug tracking database, the Gentoo mailing list archives, and the Gentoo revision control system. As the Gentoo Web site is under revision control, I did not organize relevant documents from this Web site in the documentary database. Instead, I decided to rely on Gentoo's revision control system.

3.3. Data analysis

Ethnographic data analysis is an ongoing process from the moment the field worker enters the field until the complete research report is written. During field work the data analysis is informal. Upon withdrawing from the field, the data analysis is gradually formalized. Informal data analysis is a continuous activity through out the period of fieldwork. Because this analysis is closely connected with the daily details of fieldwork, there are no clear links between this analysis and the topics discussed in this paper. We have therefore opted for a more general description of the activities of informal analysis, and instead present the details of the formal analysis as this is closely connected with the topic of this paper.

Informal analysis takes the form of writing out the notes that have been quickly and briefly jotted down in the notepad during the day's observation, and organizing them into more coherent field notes. By relating the day's observations to previous field notes, I continuously looked for patterns in my observations for building informal theories. These informal theories, in turn, inform how I continued performing the fieldwork. This way, I was able to better fit the way I performed my fieldwork with basis in an increased understanding of the research setting.

Upon withdrawing from the field the first author spent a year working systematically through the collected data, looking for recurring patterns. Once the recurring patterns are identified and formulated, formal data analysis commenced. Formal data analysis is a process of incrementally generalizing from a multitude of singular observations to increasingly more generalized descriptions of activities. Throughout this process, non-recurring details of the singular observations are omitted and recurring issues included. However, determining which details to omit in the final analysis and which to include is an iterative process of working on generalizing the descriptions while continuously returning to the more detailed analyses looking for recurring patterns that may shed light on the generalized description.

During formal analysis we identified a set of bug reports in the Gentoo bug tracking system. The bug reports were identified to capture the width of bug reports submitted. The selection criteria were based on the field notes and experiences from the fieldwork. Upon identifying a set of relevant bug reports, we went through each report, reconstructing a time line for the bug report based on the bug report activity log feature provided by the defect tracking system. Into this time line we also placed discussions about the bug from the Gentoo IRC channel logs collected during the period of fieldwork, the Gentoo mailing list archives, and the Gentoo Web forums. In this timeline we simply cut and pasted from the various data sources. With basis in this, we wrote an executive summary of the bug report's life cycle as well as writing out a complete narrative of the bug report's life cycle with explanations.

With basis in these narratives, one for every bug report in the set, we started relating our data to theory. At this stage we focused on establishing relevance and context of our observations. We tried a number of theories for interpreting our data; ranging from social theories on decision-making, via theories on distributed cognition from the field of computer supported cooperative work, to more standard software engineering theory on software maintenance. From this analysis the focus on maintainability, which led us to the last part of the formal analysis, which is to write up the results and analysis presented in section 4.

3.4. Research validation

Ethnographic research does not follow a step-wise process. Rather, the data collection requires flexible responses to the specific circumstances of the moment. This flexibility also means that the research design changes in the face of in-field realities that the researcher could not anticipate at the outset of the study. It is therefore difficult to ground the study's validity in the procedural rigor of controlled experiments. Instead, the validity is established through a rigor in argumentation by following the seven principles for conducting fieldwork [13] as shown in Table 1.

4. Results and analysis

Following the definition of maintainability as the ease with which a software system can be understood and modified, we are focusing on the aspect of system comprehension in this paper. In this section we discuss systems comprehension in relation to each of the three concepts raised in the introduction – ambiguity, negotiation, and infrastructure – relating them to the empirical data collected and existing literature. The main point is that systems comprehension is a collective process of generating a consensus-based comprehension of the system and how it causes the observed failures.

4.1. Ambiguity

Some software systems fail. A software failure is an externally observable error in the program behavior. Software failures are caused by software faults that are triggered under specific circumstances during execution. Upon experiencing a software failure that cannot be corrected locally, Gentoo users submit a bug report to the Gentoo defect tracking system (http://bugs.gentoo.org). The bug report is analyzed by Gentoo developers and resolved either by rejecting the reported failure as a real failure, by correcting the fault causing the failure, or by forwarding the report upstream. As the Gentoo developers repackage software developed by external OSS projects, forwarding bug reports upstream means that the failure is not caused by Gentoo specific code or interaction of components distributed by Gentoo, but found to be caused by faults in the third-party software. This is the overall description of Gentoo's corrective maintenance process.

Gentoo uses Bugzilla, a Web-based OSS defect tracking system. In Bugzilla, failures are reported as bug reports in a standardized Web form. Bugzilla provides a standardized schema for describing the failure and for administration of bug reports. This schema is mostly used for assigning bug reports and tracking their status. A recurring pattern in the use of Bugzilla is that the Gentoo users and developers use the optional text field at the end of the bug report, named *Additional comments*, during corrective maintenance. Why is that?

Table 1. Research validation

Principle	Description	Validation
1. The fundamental principle of the hermeneutic circle	This principle suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form.	Discussion of the iteration between the day's findings and previous field notes during informal data analysis, and the process of working on generalized descriptions while returning in detailed analysis, Section 3.3.
2. The principle of contextualization	This principle requires critical reflection of the social and historical background of the research setting	Discussion of the shift from application software to SI, Section 1. Relating Gentoo to SI and discussing of the historical relationship between Gentoo Linux and distribution system, Section 3.1.
3. The principle of interaction between researcher and subjects	Requires critical reflection on how the research materials were socially constructed through the interactions between the researchers and participants.	Discussion of interviews during participant observation, Section 3.2.
4. The principle of abstraction and generalization	Intrinsic to interpretive research is the attempt to relate the particulars described in the unique instances observed to abstract categories and concepts that apply to multiple situations.	Presentation of ambiguity, negotiation, and infrastructure as theoretical constructs, Section 1. Relating the analysis to these constructs, Section 4.
5. The principle of dialogical reasoning	Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research and the actual findings.	Discussion of establishing relevance and context of observations, Section 3.3.
6. The principle of multiple interpretations	This principle requires the researcher to be sensitive to difference in interpretations among the studied subjects.	Central topic throughout analysis and conclusion, Sections 4 and 5. Multiple interpretations the process of negotiation is discussed in Section 4.3.
7. The principle of suspicion	Requires sensitivities to possible biases and systematic distortions in the narratives collected from the participants.	Discussion of no clear principles for resolving bug reports, Section 4.2.

It need not be obvious what the failure "really is". Reporting failures is a balancing between providing too little information and too much information, but sufficient and relevant information [23]. However, it is difficult for a user to determine what sufficient and relevant information is when it is not obvious what the failure really is. Instead, the process of describing the failure is often a series of exchanges where the developers ask the user reporting the failure to generate more information about the failure. These exchanges may span over days, weeks, or even months before the bug report is resolved, and this is what the *Additional comments* field of the bug report is used for.

Martin & McClure [17] argue that programmers doing corrective maintenance "do not know where to look and often waste a great deal of effort looking in the wrong place". The exchanges back and forth between Gentoo users and developers may seem like a process of trial and error like that described by Martin & McClure. However, the view that corrective maintenance is a question of following the infection chain from the observed failure to its fault, presupposes that the observations of the software failure are unambiguous. However, as Endres [8] notes, "[t]here is, of course, the initial question of how we can determine what the error really was". He equates the error with the correction made, noting that this is not always correct but sometimes the bug lies too deep to be grasped or corrected. In SI the most significant problem is that failures are caused by external packages that the Gentoo community cannot control. Typically, this would lead to rejecting the bug report [23], but in Gentoo this problem is so prevalent that the developers have to bypass it.

The software being integrated by Gentoo is developed and maintained by other OSS projects, 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. It is therefore usually not possible to trace the infection chain of the failure. Instead the Gentoo developers use standard Unix tools and diagnostic tools developed specifically for Gentoo to generate indirect information about the failure. Along with the textual information provided in the bug report, we call the output of these tools *debug texts*. It is often impossible to establish what the failure "really is" from these indirect observations. However, during this exchange between users and developers, the users iteratively provide developers with additional debug texts in an attempt to reconcile the data. During this process multiple interpretations of what the failure "really is" are constructed from combining elements from the different debug texts. "Ambiguity means that a group of informed people are likely to hold multiple interpretations or that several plausible interpretations can be made without more data or rigorous analysis making it possible to assess them" [1]. As such, these failures can be considered ambiguous because what the failure "really is" cannot be established given sufficient information. Instead, this information gives rise to several plausible interpretations of the failures.

With ambiguity the possibility of clear cause-effect relationships and exercised qualified judgment becomes seriously reduced. Cast another way, the understandability of the software becomes seriously reduced. Instead, an understanding of the software failure and its corresponding fault is established through negotiation.

4.2 Negotiation

Gentoo as a software system lies outside the intellectual grasp of a single individual, requiring several organized groups of people to develop and maintain it (see section 3.1). As no single individual can have complete systems comprehension, understanding failures and their corresponding faults becomes a collective activity where individual Gentoo developers' partial comprehension is combined. This is further accentuated by the fact that there is no single Gentoo installation, but thousands of Gentoo installations where software failures occur. As such, the users' knowledge of local system configuration is an important part of the knowledge required to generate a comprehension of the software failure. An understanding of the failure is therefore reached through an iterative process where the user produces new debug texts and the developers generate interpretations of these texts by negotiating over the meaning of the texts. These negotiations often lead to new requests for debug texts in an iterative cycle until a consensus interpretation of the failure is reached. As such, negotiation is the collective process of sharing existing system comprehension and generating new through the production of debug texts. However, this is also a process of reducing the number of interpretations to reach a closure of the bug report. Through consensus interpretations are made invalid.

During negotiation there is often a wide variety in interpretations of the source of failures. It is often hard to find the source of failures resulting from unpredictable interaction of several packages, and as "deciding upon who is to blame is a political process" [23]. Complex interactions among the packages provided by Gentoo produce similar situations in Gentoo. Such interaction effects can also be observed in the interface between the software distributed by Gentoo and the underlying operating system. Varying standards of system calls among Unix versions can also increase the complexity of the failure. This is a sort of interaction effect akin to architectural mismatch [5]. Finally, failures may also be caused by specific configurations of the user's system. Common to the above failures is that it is hard to locate the fault. The failures are ambiguous in the sense that they lack clear boundaries.

Negotiation is the approach for overcoming this problem. As such it is very much like the political process described by [23]. If it cannot be resolved technically, the fault is located through consensus. However, there are no clear principles for doing so. For instance, one might assume that failing to reproduce a failure would be an indication that the fault is with the user's local configuration and be grounds for rejecting the bug report. Sometimes irreproducibility means the rejection of a bug report. At other times, irreproducible failures or even failures found to be caused by user configuration are resolved. What we see is that the criteria for resolving or rejecting failures varies from bug report to bug report. This is but one of many examples of a pattern of no clear principles to determine what constitutes a valid failure or for resolving unclear boundaries in failures.

Such a lack of clear principles is another trait of ambiguity, and can be seen as the result of several people with differing priorities and practice doing corrective maintenance. This is a reasonable explanation and can in part explain the lack of clear principles. However, the explanation should not overshadow the interpretation that some of this lack of principles is also a product of the ambiguity of software failures as a result of the complexity and variability of Gentoo. This can explain the uncertainty, complexity, instability of principles, and uniqueness in the way bug reports are handled. The lack of clear principles raises issues of power, but this is outside the scope of this paper.

One might be tempted to see the process of negotiation as a way of reducing or overcoming ambiguity. Yet, at its very heart lies the need for ambiguity. It is not uncommon that developers refuse to assist in helping to resolve bug reports even though the fault can be identified within their area of responsibility. When this happens, ambiguity plays a role in getting the bug report back on track again. If there were no room for interpretation, there would be no way of proceeding with resolving the bug report. However, with multiple interpretations it is possible to pursue another interpretation in order to resolve the bug report.

4.3. Infrastructures

In the above analysis we have moved from the ambiguity generated in the technical domain to the social processes of interpretation and negotiation to cope with and handle this ambiguity. In this section we will once again return to the technical domain, albeit with a definite connection to the social. From the above analysis we see that knowledge and systems comprehension may be understood as a product or an effect of various materials. It occurs in the form of debug texts, in the skills for using the debug tools embodied by the Gentoo users and developers, and in the knowledge about the system and typical failures embedded in the debug tools. Not only is systems understanding the product of these materials along with the tools and people generating them, but through knowledge about the system and frequently occurring failures embedded in the tools the tools themselves participate in generating the possible interpretations. As such, corrective maintenance is made possible by this network, or infrastructure, of tools and people [15].

We find that the Gentoo infrastructure of debug tools consists of two groups of tools. Tools in the first group are standard Unix tools like, for instance, `strace` for tracing system calls and signals or `ldd` for printing shared library dependencies. These are debug tools known to most Unix developers. The other group of tools is the custom tools specifically made for Gentoo. Among these are tools that are distributed as part of Gentoo, tools available from private home pages of developers and super users, and tools available from an unofficial repository for Gentoo tools. Debug tools are also proposed and discussed on the IRC channels, and it is common for people to submit debug tools they have developed as bug reports in the Gentoo defect tracking system.

The infrastructure of debug tools is used for generating debug texts. As such, their role is to generate data and to support the negotiation over possible interpretations of these data. We include the Bugzilla defect tracking system as part of the infrastructure of debug tools, too, since it both supports the communication among developers as well as being used for marking duplicate bug reports. Duplicates often provide valuable information on invariants of a software failure.

While the Gentoo developers are not explicit on the process of developing and maintaining the Gentoo-specific debug tools nor on the importance of this job, in practice they are performing a process where knowledge about typical error situations and typical diagnostic actions are inscribed in tools. As typical failures change over time, tools are made obsolete and new tools are added either in the official distribution or on the unofficial locations such as home pages and the tools repository. It is quite common to see references to Web pages with tools on the developers' IRC channel. This devising of relevant debug tools and the demise of irrelevant tools is a continuous process contingent upon the current reported failures.

4.4. A proposed maintainability model

We see, then, that developing and maintaining Gentoo involves ambiguity both in product as described in the research setting and in process as described in the results and analysis above. This ambiguity of process and product manifests itself in the corrective maintenance activities. Tracking down the source of failures is a process of generating systems comprehension through the production and interpretation of debug texts. We see from the above analysis that tracking down the bug need not be all that simple in practice. It need not be obvious what the bug "really is". Rather, it is subject to interpretation and negotiation. A number of possible interpretations are discussed, and none are dismissed on conclusive evidence but rather made less plausible. Alternative explanations for what the failure "really is" are constructed from combining elements of the different debug texts. The explanations are made more or less plausible both by producing new debug texts, trying to reproduce the failure, drawing on external texts like installation scripts and change logs, or simply by refusing to enter a discussion over possible interpretations.

What we see then, is that reaching an agreement as to what the failure really is, is made with both ambiguous and inconclusive evidence and is more or less open throughout the process. Finding the source of the problem is a process where the person reporting the failure and those trying to understand it work together to find relevant pieces of information and producing additional debug texts. Making the software maintainable can therefore be interpreted as a collective process including both the person submitting the bug report, those trying to understand and resolve the problem, as well as the tools involved in producing the various debug texts being interpreted. The software is made maintainable by iteratively producing debug texts, extracting fragments of information from these texts and assembling these fragments into meaningful combinations.

With basis in this, we propose a model to describe the corrective maintenance process to support our explanation of maintainability. We present two views of this model. Figure 1 shows the cyclic process of producing new debug texts and generating new interpretations through negotiation. The vertical arrow in the middle of the cycle illustrates the number of interpretations.

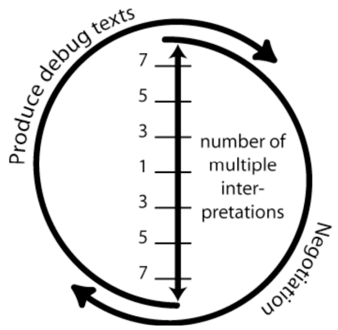

Figure 1. Cyclic view of the corrective maintenance process

Through iterations of the process, the number of interpretations may contract or expand. This is shown in Figure 2. This figure provides a temporal view of the process from the bug report is submitted until it is closed. The number of interpretations is a function of both the level ambiguity and the degree of consensus among developers. Reaching the point of closure can therefore be achieved through the elimination of ambiguity or simply by reaching a consensus about how to resolve the bug report by possibly rejecting it without any technical basis. These are the extremes. More commonly, though, bug reports reach their closure through reducing the ambiguity and reaching a consensus.

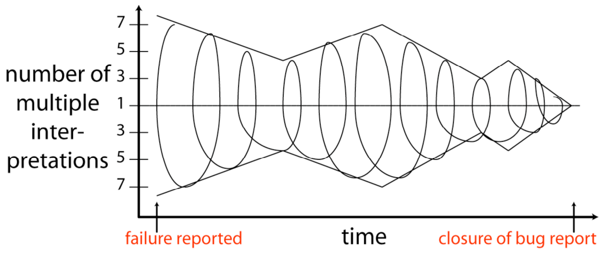

Figure 2. Temporal view of the corrective maintenance process

5. Conclusion

With this basis, we return to our research question: how is maintainability established in systems integration? We find that maintainability is established through the development, operation, and maintenance of a debug infrastructure. This infrastructure mostly supports interaction between developers, like the way

Bugzilla, IRC, and mailing lists are used in Gentoo. The infrastructure must also consist of tools that generate relevant debug information. This is done by constantly evaluating the usefulness of existing debug tools towards the typical failures reported. For Gentoo, we see that this is a continuous process of developing new tools, revising existing tools, and the demise of tools that are no longer useful.

With basis in this we may rephrase our solution to the problem of establishing maintainability in SI. Maintainability in SI may be established through an infrastructure that bridges both the geographical and knowledge gaps between actors in the corrective maintenance process.

6. References

[1] Alvesson, M., *Knowledge Work and Knowledge-Intensive Firms*, Oxford University Press, Oxford, 2004.

[2] Bass, L., P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Boston, Massachusetts, 2003.

[3] Belady, L.A. and M.M. Lehman, "Characteristics of Large Systems", *Research Directions in Software Technology*, P. Wenger (ed.), pp.108-138, MIT Press, Cambridge, Massachusetts, 1978.

[4] Boehm, B.W, J.R. Brown and J.R. Kaspar, *Characteristis of Software Quality*, TRW Series f Software Technology, Amsterdam, North Holland, 1978.

[5] Boehm, B. and C. Abts "COTS Integration: Plug and Pray?" *IEEE Computer,* pp. 135-138, January 1999.

[6] Calzolari, F., P. Tonella and G. Antonioli, "Dynamic model for maintenance and testing effort", *Proceedings of the International Conference on Software Maintenance, ICSM'98*, pp. 104-112, 1998.

[7] Dalal, S.R., J.R. Horgan and J.R. Kettering, "Reliable software and communication: Software quality, reliability, and safety", *Proceedings of the 15th Conference on Software Engineering, ICSE'93*, pp.425-435, 1993.

[8] Endres, A., "An Analysis of Errors and Their Causes in Systems Programs", *Proceedings of the 1975 Conference on Reliable Software*, pp. 327-336, 1975.

[9] Feller, J. and B. Fitzgerald, *Understanding Open Source Software Development*, Addison-Wesley, Boston, Massachusetts, 2002.

[10] Gibson, V.R and J.A. Senn, "Systems Structure and Software Maintenance Performance", *Communications of the ACM*, pp. 347-358, March, 1989.

[11] Hasselbring, W., "Information Systems Integration", *Communications of the ACM*, pp. 32-38, June, 2000.

[12] Fetterman, D.M., *Ethnography*, Newbury Park, CA: Sage Publications, 1998.

[13] Klein, H.K. and M.D. Myers, "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems", *MIS Quarterly*, pp.67-93, January, 1999.

[14] Lam W. and V. Shankararaman, "An Enterprise Integration Methodology", *IT Professional*, p. 40-48, March/April, 2004.

[15] Law, J., "Notes on the Theory of the Actor Network", 1992, http://www.lancs.ac.uk/fss/sociology/papers/law-notes-on-ant.pdf

[16] Lethbridge, T.C., S.E. Sim and J. Singer, "Studying Software Engineers: Data Collection Techniques for Software Field Studies", *Empirical Software Engineering*, pp.311-341, July, 2005.

[17] Martin, J. and C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[18] McCall, J. A. "Quality Factors. In Marciniak", *Encyclopedia of Software Engineering, Vol. II*, John J. (Ed.), John Wiley & Sons, New York, pp. 958-969, 1994.

[19] Paulson, J.W., G. Succi and A. Eberlein, "An Empirical Study of Open Source and Closed-Source Software Products", *IEEE Transactions on Software Engineering*, pp. 246-256, 2004.

[20] Samoladas, I., I. Stamelos, L. Angelis and A. Oikonomou, "Open Source Software Development Should Strive for Even Greater Code Maintainability". *Communications of the ACM*, pp. 83-87, 2004.

[21] Scach, S.R., B. Jin, L. Yu, G. Z. Heller and J. Offutt, "Determining the Distribution of Maintenance Categories: Survey versus Measurement", *Empirical Software Enginering*, pp. 351-365, 2003.

[22] Vogels, W., "Web Services are not distributed objects", *IEEE Internet Computing,* pp. 59-66, November/December, 2003.

[23] Zeller, A., *Why Programs Fail: A Guide to Systematic Debugging*, Elsevier, Amsterdam, 2006.