Thrashing, Tolerating and Compromising in Software Development

Tamara Lopez[1], Marian Petre[1], and Bashar Nuseibeh[1,2]

[1] Centre for Research in Computing, The Open University
t.lopez, m.petre, b.nuseibeh@open.ac.uk
[2] Lero - The Irish Software Engineering Research Centre, Ireland

Abstract. Software engineering research into error commonly examines how developers pass judgement: to isolate faults, establish their causes and remove them. By contrast this research examines how developers experience and learn from things that go wrong. This paper presents an analysis of retrospective accounts of software development gathered from a single organisation. The report includes findings of how work is conducted in this organisation, and three themes that have emerged in analysis are discussed: thrashing, tolerating and compromising. Finally, limitations and implications for future research are given.

1 Introduction

Software rarely works as intended when it is first written. Things go wrong, and developers are commonly understood to form theories and strategies to deal with them [4]. This research began with a desire to examine in more detail how developers reason while making software, and in particular, how they reason when things go wrong.

Software engineering research, by contrast, is commonly concerned with *passing judgement*, assessing why a piece of software has failed and who is to blame [11] , or why a piece of software is flawed and how to prevent such flaws in the future[7]. This kind of research commonly examines errors in the context of bugs, elements of software *as written* that produce undesirable, unexpected and unintended deviation in behaviour[1]. It tends not to examine errors from the perspective of individual developers, nor to consider how things that go wrong along the way are fixed, thus contributing to better software and better developers.

In the rest of this paper, we report a study designed to address these gaps.

2 Related Work

Understanding the personal strategies and theories that developers have for why things go wrong and how to deal with them is an old, but under-examined concern in software engineering research[4]. It is particularly evident in root-cause analysis, a method used to improve software engineering process.

Root-cause studies identify the kinds of faults that predominate in a system, and determine how process can be altered so as to prevented their occurrence in the future. They draw upon data from bug and modification reports [9, 10, 6], but also make use of in-process questionnaires [2] and retrospectively administered surveys [8]. Data are analysed and classified into taxonomies that identify the root-causes for errors. The classified set of data forms the basis for additional examination of particular code features such as complexity [13], interface defects [9, 10] or more generally, environmental factors that influence software dependability [2]. The findings similarly address familiar software engineering themes. Complexity is found both to correlate to error frequency [13], and not to [2]. Application programming interfaces are found to have particularly high frequencies of errors associated with them [9, 10] while these and other causes are evaluated in terms of the cost associated with finding and fixing faults [13, 2, 6].

Over the years, these studies have consistently made two suggestions for additional research. The first is that data about errors should be collected from the entire development cycle, not

just at points of testing and integration[4][7], and should not be collected too long an interval of time after events have passed[8]. The second is that studies should be made that examine the causes of "human erring"[4, p.331], including factors such as problems of understanding[4], inexperience[10], lack of information[8], and skill mismatch[6].

Unfortunately, though these papers offer clear ideas about what to examine, they do not offer many suggestions for how to go about doing it. They do, however, suggest likely challenges. To get around the fact that time erodes knowledge about errors, Perry suggests that programmers be asked to classify their errors as a part of closing modification and bug reports [7], a technique found not to work particularly well by Leszak et al.[6]. Organisational access is noted to be difficult to attain when it requires sharing information about mistakes[7], and management can seriously constrain study design, in extreme cases resulting in retrospectively gathered, anonymous self-reports[8].

3 Talking to Developers: Critical Decision Method

We wish to address a long-held concern in software engineering research about how developers experience and learn from things that go wrong in their work. Methodologically, we seek to address the gaps identified by the root-cause analyses: to gather evidence about error from the full development cycle, with particular emphasis paid to those areas that have been associated with "human erring": problem understanding, experience, information and skill.

To explore both gaps, we adapted the critical decision method, as described in *Working Minds: A Practitioner's Guide to Cognitive Task Analysis*[3]. The critical decision method was developed to study how decisions are made in real world settings. In addition to illuminating how people think on the job, the larger framework of cognitive task analysis assists researchers in understanding expertise in individual domains, by revealing the differences between how experts and novices approach and manage their work.

This method has been used before to study expertise in software development. One notable study was conducted at Bell Labs to produce a training course in expert debugging skill[14, 12, 5]. It began with the premise that software development is a domain in which there is a clear difference between the way experts and novices perform their work and where experts become so only after years of experience on the job. Data was drawn from sixteen critical decision method interviews with experts, and two interviews with typical (i.e., not expert) developers. Findings were corroborated and enlarged through a focus group and via surveys distributed to developers throughout the company.

The study found that expert debuggers, like experts in other domains, think extensively about problems before taking action. This manifested in waiting longer to employ debugging tools, and in finding information about what to try next rather than jumping into "poorly directed" but hopeful activities. Study participants suggested that less experienced developers, by contrast, were reported to "thrash" around, trying to find solutions.

The authors also reported differences in how experts and novices handled "close reading" of code to establish what it does, and the past history of the data it handles. Both novices and experts were found to read comments, but novices were less critical of what comments say, tending to take them at face value. By contrast, experts treated the comments as evidence of the number of hands present in a piece of software, and the conditions under which authors were working. Finally, the authors found that experts read code as a last resort, preferring instead to seek help from other developers with detailed knowledge of the software.

Though this study collected detailed information about technical aspects of bug fixing, feedback given by participants led the analysts to focus their efforts on explicating the social aspects of debugging. It did not examine expertise in the context of other kinds of development activity, and reported findings based on the views of acknowledged experts in a single organisation. The findings are thus compelling but leave room for more detailed examination of development activities other than debugging, and the inclusion of non-expert perspectives.

3.1 Study Execution

For our study, seven individuals were interviewed over the course of four weeks. Participants perform a range of software development tasks in an established UK digital humanities centre, described in 2008 by the Council on Library and Information Resources as an environment "where new media and technologies are used for humanities-based research, teaching, and intellectual engagement and experimentation"[15, p. 4]. Each informant was asked to recount an incident in which they played a discrete role, in audio-recorded sessions that lasted from between forty-five and seventy-five minutes.

Interviews were conducted by a single person, but otherwise followed the basic procedures for conducting a critical decision method interview. These entail examining a single incident in four semi-structured "sweeps". In the first sweep, the informant and the researcher identified an incident, broadly defined as one having taken place in the previous two weeks and in which the informant was a key decision maker. In the second, a timeline was established to note critical decision points. In the third, deepening probes were used to develop comprehensive and detailed understanding about the incident. Though researchers often selectively use probes at this stage to examine one or two cognitive phenomena, this study made opportunistic use of a range of probes, with an aim to identify in analysis those which are most effective for learning about things that go wrong. Finally, the informant was asked to consider hypothetical alternatives to decisions taken.

Each interview concluded with questions about the informant's educational and professional background. Informants were asked to give the researcher copies of artefacts mentioned in the discussion.

A Note About the Workspace All of the interviews took place at developers' desks; five of the seven developers were located in the same open plan office. Desks were located in close proximity to one another, and employees were aware that interviews were being conducted. The sixth developer was located in a different open plan office, and the final interview was held in the participant's private office.

The choice to conduct these interviews *in situ* was deliberate. It was felt that conducting them in the developer's own environment would allow for better access to physical and digital artefacts that came up in conversation. Given the focus on problems and challenges, it was also hoped that holding these discussions in the open would signal to informants that the topic was not being pursued in order to assign blame, but rather in a spirit of inquiry. Informants gave no indication that the choice of venue made them uncomfortable, though it was noted in several cases that informants displayed discretion in referring to colleagues located in the same office, either by lowering their voices or by referring to them simply as "my colleague".

Information on computer screens, paper diagrams and a poster on the wall were used to initiate discussion in three cases. In addition, informants shared source code with the interviewer, explained the output of stack traces and demonstrated debugging tools, prototypes and the software being built. Several developers appeared to remember with their fingers, orally recounting details while at the same time accessing files and websites and conducting internet searches similar to those they had used in solving problems.

3.2 Data Analysis

Data analysis began at the point of collection. Terms were checked with informants and information previously given was stated back for clarification and correction. In several instances, restating information also resulted in the addition of omitted details. Immediately following each interview, notes taken during the interview were annotated and expanded. In addition, reflection was made to describe impressions and details of the major topics raised in the interview, and to evaluate application of the method.

In addition to the in-interview corroboration, informants were sent follow-up email messages seeking additional materials mentioned in-interview, and inviting them to provide additional comment. Informants have also been sent draft copies of reports featuring information pertaining to their case.

A near-verbatim transcription was created of each interview. Each transcript was read and annotated in an inductive, iterative process to identify themes in the data. The analysis of individual texts was supplemented by the development of matrices and diagrams to explore points across cases.

3.3 Validity

Given its exploratory aims and its focus on a single organisation and method of data collection, the results of this study alone cannot make strong claims of validity. In addition, it must be stated that the primary researcher had prior understanding about the organisational culture in which the informants work. However, this researcher had no direct knowledge of any of the projects discussed.

4 Results

This section presents results of the cross-case analysis of six interviews. The seventh interview did not result in the identification of a clear incident, and is not included in this report. This interview did yield information about how work happens in this organisation, which has been used in analysis for comparison with other cases. The views of individual developers are presented using pseudonyms.

4.1 Participants

Given the focus on understanding more about things that go wrong, it was felt that having access to participants with a range of experience would be useful. To that end, this study did not strive to identify and interview acknowledged experts. Seven people were interviewed, six men, and one woman. The youngest participant was in her late twenties, and the oldest was in his sixties. The rest of the participants were in their thirties and forties.

Developers old and new to the organisation were interviewed, with one having less than a year at the organisation, and one over ten years. Two participants had computing degrees, one had a computing postgraduate degree, one had a computing applications postgraduate degree, and one had a postgraduate computing diploma. Three had industry computing experience in the web media, financial, education and GIS sectors. Two had post-graduate or research degrees in the social sciences and humanities. There were also humanities computing specialists, with one informant having at least two decades experience in digital humanities work, and a second having a decade and a half. These informants had both worked in multiple organisations on digital humanities projects, while for the remainder, this was their first position in a digital humanities centre.

4.2 Projects

Informants described work performed for three projects. James described refactoring a piece of software for a tool he is building that facilitates note-taking and annotation (Project A). Valentin described a project for which he was the sole application developer, tasked with creating both an editorial tool and a web edition for displaying a critical edition of texts (Project B). The remaining four participants, Joaquim, Marisa, Evan and Richard described performing different tasks for a single project to support detailed annotation and display of medieval handwriting (Project C).

The latter two projects (B and C) follow the same general model: tools are created for use by domain specialists to manage and create data related to physical, often historical materials. These data are in turn presented to the public via other pieces of software that are also developed by the centre. Public facing outputs take the form of web editions of texts and web reference tools. In some cases, projects also produce print monographs.

This model is common in this centre, and as a consequence, development staff routinely produce software for multiple user groups with different functional requirements, and different usability thresholds. Domain specialists are full partners in the project. They closely interact with developers and analysts and are prepared to work with tools that require complicated installation procedures or which have a less than finished feel. Their priority is to have a piece of software finished enough so that they can advance their research. By contrast, readers of public facing web editions and reference tools, themselves also typically domain specialists, have an expectation that the tools they use will be finished to a very high standard. Developing tools for the public, and doing so in new and innovative ways, is a high priority for the centre, but the requirements for these tools emerge slowly, sometimes over a period of years as the specialists work with original materials and develop their understanding about what they mean.

The data produced and managed in these projects are different from commercial data: they are less structured, orientated around natural language and approximate. One developer characterised them in this way:

> "So a good example are dates. If you say the date of this manuscript is around 1113 well it could be this date or it could be that date. Or even worse somebody is saying it is that date, somebody is saying it is that date, somebody is saying it is that date. In the commercial world it is just a single precise date to the millisecond. Here you want many dates by different people and you want all the opinions shown on your website and preserved. So the interpretation is very important."

4.3 How work is done

Developers in this centre tend to work alone, even when assigned to tasks for the same project. A single person may be assigned to work on all deliverables, or different people may be assigned to different areas of the software. It is common for developers to work on the same project at different times. Developers know the others who are working on their projects, and report that they attend meetings at which other developers are present, however each works in reference to the overarching project team. Participants identified having an area of technical expertise such as in application or interface development or in data modelling. Despite this, several recounted the need to learn new skills to meet requirements for projects that emerged over time. For example, one application programmer described learning and implementing client-side technologies, while another developer who was proficient in XML data modelling described a need to learn relational data modelling.

Developers also take the initiative for prioritising and organising their work. This can involve adopting new working practices, as in the case of one developer who described introducing a new working style on his project as "agile-like," with rapid iterations and frequent meetings with project partners. Another described how his responsibilities at the organisation are growing, and how in his present project he extended the original task to do more on his own initiative, more or less as "the accepted order of things". Ad-hoc technical teams are important, with one participant describing "finding" his way on a project with the help of an "amazing" colleague who offered technical advice and guidance about how to manage relationships with partners. Another described "luck" in finding the solution to his issue as due in part to the technical expertise of a colleague who was not working on the project. A third described looking to a trusted colleague for help before relying on internet fora and other technical documentation.

4.4 The Problems

Though an aim of this study was to examine things that go wrong in contexts other than in bug reporting and bug fixing, bugs did feature in some of the stories. The starting point for discussion with Joaquim was a bug that had been reported to him by a project partner. Valentin, despite describing his issue as "not necessarily a bug, it's an improvement" recounted that his issue nevertheless manifested as a bug four times in the course of a year and a half, in different pieces of software that were being used and built. The first manifestation brought the issue to his attention, two later occurrences were "expected," and one was a surprise, helping him to realise how "widespread" the issue was, and causing him to re-prioritise and plan for solving it. James reported finding and fixing small "secret" bugs[4] as a part of his work to refactor the storage layer for his software.

Three participants described issues directly connected to writing software. Joaquim described an issue in implementing an interaction model for annotating digital images in a browser. Valentin described problems in rendering special characters from historical texts in web browsers. Evan described the personal challenges he faced in getting a local copy of an application framework up and running. These included resolving version dependencies in installed software and correcting filesystem path information in a configuration file.

In three other cases, the issues described revealed how external factors influence software development in this organisation. Richard described developing a data model for a project that would take into account two legacy models, work that involved close interaction with a project partner to identify and fully specify concepts. His issues primarily arose from pressures on his time, the need to "get something working" for this project, while still meeting the demands of other projects. James described a breakthrough in his thinking about how to re-architect a piece of software. His story detailed not the thing going wrong, but its flip side, how longstanding "motivators for change" that came from the larger research community in which he participates were suddenly resolved. Marisa described issues she faced in meeting complicated, ambitious requirements for a user interface. Her incident included technical aspects, but also involved learning how to recognise her own limits and to manage the expectations of her project partners.

4.5 Time

Every participant reported a working pattern of "fits and starts", the need to pick up a task and set it down as required to meet the demands of multiple projects. Perhaps as a result of this practice, the issues described by participants included relevant details that were at times temporally distant from one another. In the stories recounted by Richard and James, details were given that had occurred at least a year prior to the interview. Evan and Marisa reported events with two clear intervals, one occurring in the weeks leading up to the December holiday break, and the other in the weeks following the break. Valentin reported events that occurred a year and a half prior to the interview, and the timeline for his interview included several distinct decision points, moments when the issue manifested in a way which changed priorities and actions. Evan's timeline was much more compressed, comprising the events of a single day which had occurred in the week prior to the interview. He set out the major episodes at the beginning of the interview, and these remained relatively stable for the rest of the interview.

In some cases, timelines were more difficult to establish, perhaps because work related to the issue was ongoing at the point at which the stories were collected. Valentin, with whom a timeline was quickly established, was coming to the end of his development on the project he discussed. Evan was at the very beginning of his work on the project, and did not have a long history of experience to relate. By contrast, Joaquim, James, Richard and Marisa were interviewed in the midst of longstanding work on a project, and seemed to have more difficulty in establishing a sequence of linked events. A timeline did emerge, but it was established in analysis, and wasn't particularly useful in structuring the interview protocol. In these cases, the timeline was also not as precise, and possibly not as accurate.

As might be expected, finer detail was collected about events that occurred close to the point of interview. For example, Joaquim was able to report in some detail his interactions with a debugger and IDE for work related to the incident that had occurred the day previous, but did not report in such detail the process he used to research technologies at the start of his work for the project some four months prior. Likewise Evan was able to recall in detail his experiences of a few days previous, but did not provide as much detail about troubleshooting a related incident some two months prior.

5 Discussion

Three themes have emerged in analysis of the cases of Joaquim, Valentin and Evan, the informants who reported incidents directly connected to writing software. This section identifies and defines these themes, and presents a report of each.

Thrashing is identified in the Bell Labs study as poorly directed, ineffective problem solving. There is some suggestion that novices and experts thrash, but novices fail to realise they are doing it in good time, and fail to break out of it. Experts, by contrast, realise when they are thrashing, and seek help from colleagues with more experience. In these interviews, Evan describes a day in which he spent time thrashing while trying to set up a local copy of a web application framework.

Tolerating - Valentin describes how he tolerated an error for over a year, implementing temporary solutions along the way. He reports this activity as strategic, a behaviour that is consistent with descriptions of expert debuggers in the Bell Labs study.

Compromising - These interviews suggest that developers settle at times for sub-optimal solutions in order to move work along. This is exemplified in Joaquim's story, in which a bug fix results in a working solution, but one with which he is intuitively dissatisfied.

5.1 Thrashing: At that point I made a cup of tea

> Evan described a day in which things went wrong while setting up a local copy of an open source web application framework, a task necessary to complete in order that the "real work" could begin. Though relatively new to the framework and to the language it is implemented in, he had done the task before for a different project a couple of months prior. That time the process had not been smooth, and he had not written anything down, and his goal now was to cement the approach that had been followed. The task began well. He was able to locate and install all of the required software modules, and to get the web server to start without reporting any errors. However images weren't loading and web-pages looked funny when loaded in the web browser. After "poking around in the dark" for an hour or so, and becoming increasingly annoyed and confused, he was able to narrow the issue down to one of configuration, and to copy the necessary setting from the previous project's files. He was relieved, until he loaded a page that he expected to work and a different error message appeared. At that point, he stepped away from the computer and made a cup of tea.

The story recounted by Evan is unique among the interviews in the rich perspective it provides about things that go wrong as they occur. His is a vivid and sometimes harrowing account of "thrashing," described in the Bell Labs study as an ineffective process of going over and over a problem. He is aware that his approach was unsystematic, flawed and risky, calling it a process of attrition, and noting at one point an awareness that "if I plugged the dam somewhere it was going to burst somewhere else". At other points he described his approach as "basically experimenting" and "hacking around." He is also aware that he is a novice. In fact it is because he is "wary of screwing things up" that he has undertaken this task in the first place.

External factors may have contributed to his difficulty. He was working on a home computer, tunneling into a virtual machine hosted on his machine in the office. He became confused in switching between environments, turned around about what he had done, "what I'd changed and what hadn't changed."

He faced two obstacles. The first was one of conguration and is noted in the tutorials and documentation of the open source framework to be tricky, suggesting that other developers have encountered similar difficulty. The second involved a dependency between modules of the framework which could only be resolved by downgrading one of the two to an earlier version. Both of these issues had been encountered in an earlier installation of a local environment for a different project nearly two months prior, before a holiday break.

Thrashing is described as a negative novice behaviour in the Bell Labs study. Novices were found to fail to recognise that thrashing is happening, and to be unable to break out of it. The reports suggested that experts might thrash, but that they were able to attend more quickly to emotional cues that they were doing it, and to seek help sooner from colleagues with greater expertise.

By contrast, in Evan's story, thrashing is not only "annoying and confusing" it is also useful, because it forces him to take a closer look at the software he is using and building. As he puts it:

> "You know this is quite informative 'cause obviously you would get something and it would work out of the box and you don't really think about it again, so even though this was an annoyance, it was quite useful to actually have to look into those relationships."

Evan was also able to recognize when he had reached the limits of his knowledge and experience. His description of the strategy he employed at this stage is insightful:

> "I'd spent long enough messing with the configuration files. I realised either it wasn't there or I'd broken it completely. Let's get it back to how it was - you know I think you take a step back and you think okay it should be working the way it is so let's move on to the next thing and try and understand."

In the end, Evan gets everything working, but his confidence in the solution is not high. He is not seeing any error messages or funny behaviour which suggest to him that everything is working now, "touch wood". He considers the day to have been a "personal failure," but he is more confident when describing how his knowledge of the application framework has grown:

> "I'm comfortable with creating that environment, I'm comfortable with getting up and running and also I'm much more aware of creating something that's got a bit of longevity."

5.2 Tolerating: I wouldn't say "cropped up". I expected to see the error.

Valentin describes an issue that surfaced as a bug several times over the course of nearly two years, in tools used by the developer and in different areas of the software being developed. The issue was related to the use of Unicode, which presented particular complexities when introduced to the domain. Though Valentin has years of commercial software development experience and considers himself to be well-versed in this standard, a solution was not immediately apparent. He explains that the first occurrence of the issue in the organisation's documentation wiki prepared him for later manifestations. These he managed by "setting aside the complexity" of the problem and by making assumptions about users' environments. At two stages, he implemented temporary solutions, a strategy that allowed him to concentrate on fulfilling more important requirements for the project, and to analyse the problem "in the background". In the end, a permanent solution was found with the help of a colleague who is more skilled in client-side development.

Valentin's story is striking in its clarity. Though this particular situation is new to him, he is an experienced developer, and this is reflected in the way he describes organising his work:

> "I didn't want to be in the situation where I'm approaching deadline, a phase where we have to do a demonstration or release this on the live website and I have to find a solution in just a very limited time for a problem I've never encountered before. So I'd rather prepare the thinking and explore things in different directions to be sure that I will be ready for that."

The findings of the Bell Labs study relate specifically to expert debugging behaviour, however they resonate at points with comments made by Valentin. That study found that expert debuggers think extensively about problems before taking action. For Valentin, this takes the form of partial, temporary solutions. As with expert debuggers, Valentin provides evidence that he uses the time to find information about what to try next, he plans and prepares before taking action.

The story recounted by Valentin, like the Bell Labs reports, also includes evidence of sophisticated interpersonal skills in interacting with domain experts. He used feedback from domain experts to prioritise and plan when the issue appeared in an unforeseen area of the software. This occurrence made him realize that the problem was more widespread than he had thought, and also clarified for him its importance to the domain experts. He notes that he felt pressure at this point to identify a strategy for addressing the issue:

> "I had to say something, to tell them that I have a strategy, not necessarily a solution, but a strategy."

Valentin is confident and pleased with the ultimate solution, describing it as "very clean" and "well established". However, he realises that if he had not had the help of a colleague, he may have had to accept an inferior alternative. He is also keenly aware of how his own limits contributed to the issue. Though he had a superficial awareness of the solution that was adopted, he admits that his knowledge was lacking and that he "hadn't done his job" at keeping up with user interface development.

5.3 Compromising: I'm just not that happy with it yet

> Joaquim described fixing a recently reported bug in a tool he is building to support detailed editorial work on medieval manuscripts. Recreating the flawed behaviour was tricky because the conditions under which it occurred were not accurately reported, and triggering the bug required performing an unplanned-for action multiple times. However the fix itself was trivial, involving altering basic conditional behaviour in a single function. Now he has a produced a solution which is meeting requirements. He is not satisfied, however, explaining that he is not "still not very happy with it yet," and that he is not sure how well it is working. He is aware that his understanding of how the open source library he is working with is still developing. He is cautious, describing the library he has found as "not the right way, a better way". This caution extends into the software, causing Joaquim to hold back on promoting the function where the bug was located into the general API he is designing.

The details of Joaquim's story are less emotionally vivid than those given by Evan. He, like Valentin, has spent a number of years developing software in commercial and academic environments, and of the three developers, has spent the most time in this organisation. However his account included less evidence of overt strategy than that of Valentin, perhaps because his story was collected in the midst of his work on the project. The details he recounts suggest

that at times even experienced developers make simple mistakes and settle for solutions with an intuitive sense, rather than a rational understanding that they are flawed.

Though most strongly exemplified in Joaquim's story, evidence for this theme was also provided by Evan and Valentin. It also surfaced in accounts given by James, Richard and Marisa. These developers did not recount incidents so directly connected with writing software but, like Joaquim, were interviewed in the midst of longstanding work on a project.

Compromise was related in terms of an intuitive sense of how well software is functioning, a sense which at times contradicts cues given by the software. Joaquim has achieved a working solution in his design, but he is dissatisfied and still feels that something isn't right, that it may not be working well enough. Evan has no information that things aren't working, but is still wary. Everything is working now "touch wood".

Closely related to this, informants conveyed that they had settled for a sub-optimal solution in aesthetic terms. Evan has gotten his application framework to run, but is aware that it is "pretty dirty". Joaquim described working with event handling in web-browsers first as "not nice," and later explained a particular event handling scenario as a "bit of a disaster". James describes his refactoring work as "good enough", but notes that he would have liked it to be a "a little prettier".

Compromising is informed by past events, and shapes future expectation. Evan expects that he will have similar kinds of problems when he promotes his software to a different environment. Valentin reported that he expected to see occurrences of the rendering error based on past occurrences. As he described it, "this [the first occurrence] prepared me for that". Richard doesn't think he has achieved the perfect data model, but isn't sure what exactly might be wrong. Even though his project is behind schedule, he feels that they have "moved too quickly" and as a result are going to have to settle for a less than "ideal" model.

Valentin suggests that intuition can dissuade developers from compromising, describing an unused alternative solution as "very ugly," something to be used only as a "last resort". Similarly, Marisa recounts nearly giving up, before trying again to find a solution for a respected project partner:

> "I didn't want to go and say... It's impossible. Trying to find something that was maybe not everything that they wanted but better...You don't give up."

6 Limitations

Stories were collected from a single organisation, and may not represent software development in different sectors, or in organisations with different work practices. As these stories were gathered retrospectively, it is possible that details were forgotten or distorted[3].

Indeed, the accounts given yielded rich evidence for thrashing tolerating and compromising, but cannot fully explain them. Was Evan's story of thrashing solely the result of his lack of experience? Was Valentin's story an unambiguous example of an expert programmer at work? Joaquim's story suggests that there may be more to both stories. Of the three, his is the only story collected while he was in the midst of a long development arc. He is the developer with the most experience in the organisation and yet the bug he fixed was trivial, and might have been interpreted as a novice error. These facts and the way he describes settling *for now* suggest that the perspective developers hold when stories are collected is important. Gathered early in a development or learning process, errors may lend themselves to novice explanations. Gathered completely after the fact, decisions related to things that go wrong may be reported as strategic. However when they are gathered in the midst of work, a murky middle area emerges that deserves closer examination.

All of the themes suggest that errors influence and inform development work, however the theme of compromising in particular indicates that developers use intuition to identify and manage flaws in the software they are building. This sense is described in terms of doubt,

confidence, and satisfaction. However, it is not clear to what extent or in what ways they use this sense while work is happening.

Retrospective elicitation also cannot explain how individual perspective might obscure details of how thrashing, tolerating and compromising co-occur in a single development process. Joaquim's original implementation took less than an hour. The bug may have resulted in part from an earlier, hasty decision to get it working. Similarly, though Valentin reports having consciously implemented partial, pragmatic solutions, this strategy crystallised in response to interactions with project partners and managers in which the priority of the issue was emphasised to him. This suggests that even though the story was recounted as strategic, there may have been moments in which he settled for rather than tolerated interim solutions. Neither Valentin nor Joaquim report periods of thrashing like those reported by Evan. However, Joaquim does report a moment of confusion about the source of the bug and Valentin reports being surprised by a manifestation of the issue in an area of the software he had forgotten about.

7 Conclusions

The critical decision method was used in this study to elicit rich accounts of issues faced in one kind of software development. The analysis examined things that go wrong beyond bug fixing, and emphasised the perspective of the developer toward problems. This analytical perspective has yielded three insights. First, it provides a detailed look at what happens when a developer thrashes, and suggests that thrashing may hold value for the developers who engage in it. It includes evidence of expert behaviour that is consistent with related studies, but extends the applicability of such findings to development activities other than bug fixing. Finally, by considering development activity more generally, a nuanced view of how flaws are intuitively understood and managed over time emerges. This view might be explored in future research by examining how a developer's perspective toward particular choices changes in the course of work on a project.

8 Acknowledgements

We thank the software developers who shared their experiences with us. This research is supported in part by ERC grant AdvG 291652-ASAP (Nuseibeh) and SFI grant 10/CE/I1855 (Nuseibeh).

References

1. Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 91–120. Springer Boston, 2004.
2. Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.
3. B. Crandall, G.A. Klein, and R.R. Hoffman. *Working minds: A practitioner's guide to cognitive task analysis.* The MIT Press, 2006.
4. A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Reliable Software*, pages 327–336. ACM, 1975.
5. Jared T. Freeman, Thomas R. Riedl, Julian S. Weitzenfeld, Gary A. Klein, and John D. Musa. Instruction for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 271–282, London, UK, UK, 1991. Springer-Verlag.
6. M. Leszak, D.E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *The Journal of Systems & Software*, 61(3):173–187, 2002.
7. D. Perry. Where do most software flaws come from? In A. Oram and G. Wilson, editors, *Making Software: What Really Works, and Why We Believe It*, pages 453–494. O'Reilly Media, Inc., 2010.
8. D. Perry and C. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 48–67. Springer-Verlag, 1993.

9. D.E. Perry and W.M. Evangelist. An empirical study of software interface faults. pages 32–38, 1985.
10. Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults — an update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, volume II, pages 113–126, January 1987.
11. Brian Randell. On failures and faults. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Springer Berlin / Heidelberg, 2003.
12. Thomas R. Riedl, Julian S. Weitzenfeld, Jared T. Freeman, Gary A. Klein, and John D. Musa. What we have learned about software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 261–270, London, UK, UK, 1991. Springer-Verlag.
13. N.F. Schneidewind and H.-M. Hoffmann. An experiment in software error data collection and analysis. *Software Engineering, IEEE Transactions on*, SE-5(3):276 – 286, May 1979.
14. Julian S. Weitzenfeld, Thomas R. Riedl, Jared T. Freeman, Gary A. Klein, and John D. Musa. Knowledge elicitation for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 283–296, London, UK, UK, 1991. Springer-Verlag.
15. D. Zorich. *A survey of digital humanities centers in the united states.* Council on Library and Information Resources, 2008.