Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Martinec

Evaluation of Usefulness of Debugging Tools

Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký

Study programme: Software Systems
Specialization: Dependable Systems
Prague 2015

Replace this page by assignment copy

Acknowledgement

I would like to thank people that supported me during work on this thesis. These are:

- Twenty university students of the local operating system course who spent non-trivial amount of their efforts to report information about their debugging activities. This study would not have been possible without their voluntary assistance and I am convinced that our efforts yielded results that are worth it.
- People from Department of Distributed and Dependable Systems, because they enabled me to interfere with their course and supported me with their ideas and feedback. My special thanks belong to my advisor Mgr. Martin Děcký who helped to decide whether I finish this work or not one year ago.
- An excellent life coach Martin Pošta who assisted me with finding meaning in continuing with this work and helped me to become able to move forward without absolutely any external push or motivation.

Název práce: Vyhodnocování užitečnosti ladících nástrojů
Autor: Tomáš Martinec
Katedra: Katedra Distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: Mgr. Martin Děcký

Abstrakt:

Ladění je časově velmi náročná aktivita programátorů. Přestože počet návrhů ladících nástrojů je velký, tak počet nástrojů, které jsou přijaty lidmi z praxe a používány při vývoji software je menší než by se dalo očekávat. Spousta lidí věří, že jedna z příčin nastalé situace spočívá v tom, že je obtížné odhadnout, jestli se úsilí nutné pro implementaci nově navržených nástrojů nebo přístupů vyplatí.

Prvním cílem této práce je navrhnout metodologii pro vyhodnocování užitečnosti ladících nástrojů. Abychom ukázali příklad použití navržené metodologie, tak jsme uskutečnili studii užitečnosti běžných ladících nástrojů pro vývoj operačního systému. Druhým cílem této práce je prozkoumat a popsat další aspekty procesu, jak programátoři ladí software.

Klíčová slova: Ladění, Empirická studie, Vyhodnocení užitečnosti

Title: Evaluation of Usefulness of Debugging Tools
Author: Tomáš Martinec
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Martin Děcký

Abstract:

Debugging is a very time-consuming activity for programmers. Although the number of proposed debugging tools is large, the number of tools that are actually adopted by practitioners and used during development of software is less than one may expect. Many believe that one reason for the situation is that it is hard to estimate whether the implementation efforts of proposed debugging tools or approaches are worth the gain.

The first goal of this thesis is to propose a methodology for the evaluation of usefulness of debugging tools. To provide an exemplary usage of the methodology, a study of usefulness of typical debugging tools for development of operating systems is conducted. Secondly, the thesis also explores and documents further aspects of how programmers debug software.

Keywords: Debugging, Empirical study, Usefulness evaluation

Contents

1. Recommendation for usage

Anyone who uses any output of this work is strongly suggested to provide feedback how it was useful. You can use the email fyzmat@gmail.com for this purpose.

The rationale behind this is that the author will get more feedback of how correct and meaningful this work is, and it prevents you to spend your time by studying something that may not be very serious for you. Although following this suggestion could become uncomfortable, I believe that it will help you to use your energy efficiently on things that matter to you.

Sometimes life is not really easy. Thank gods for that.

2. Introduction

Programmers spend significant amount of their time by locating and fixing bugs in their programs. Many professionals believe that finding the root cause of an issue is in most cases much more difficult and time-consuming than the actual fix of the problem[1]. Therefore, a large amount of work has been dedicated to design tools that would help programmers to identify the root cause of a faulty behavior. In order to provide an idea of what has been researched so far, we enlist some references to proposed or researched debugging tools in the section 5.

Often, the usefulness of the proposed tools is evaluated only by a discussion or by a few case studies. In our opinion, having just this not a very strong evaluation makes one uncertain whether the proposal can bring significant amount of benefit and whether it is worth the implementation efforts. Furthermore, without the feedback of usefulness for these tools, the research effort invested into them can address issues that are not very relevant or practical in real-world situations.

From the industrial point of view, there already exist commercial debugging tools with very advanced debugging possibilities. For example, the company Lauterbach offers a solution of hardware assisted debugging that provides highly advanced features such as: reverse execution, symbol-based tracing and debugging on both operating system and user space application levels, support for multiple operating system and virtualization, features for timing and performance analysis and integrated tools for processing and visualizing the measured data. The management of a company that develops low-level software may be confronted with the question whether buying such an advanced debugging product will be worth the purchase, or how many licences should be bought.

This thesis is aimed to address the questions about usefulness of debugging tools by its first goal of designing a methodology that evaluates usefulness of debugging tools. We tried that methodology out on debugging tools that are available to programmers of operating systems.

As we had an easy opportunity to collect much more data about debugging than just for purposes of tool usefulness evaluation, we decided that the second goal of this thesis will be an exploration of the process how people debug computer programs. For example, we focused on mapping the relationship between debugging time, the complexity of the debugging scenario, what was the root cause of bug, or whether a debugging tool had been used. The detailed description of what data were monitored and explored is in the section 3.3.3.

2.1. Contribution

In this work, we provide the following list of contributions:

- In 3.1 we discuss some thoughts on usefulness evaluation for generic tools. Anybody who needs to interpret results of a similar evaluation (or even who needs to design one) may find value in those thoughts.
- In section 3 we describe our methodology for evaluating usefulness of debugging tools in low-level programming environment. Its uniqueness lies in the fact that the data collection was done in a group of skilled programmers for hundred hours of programming work per each programmer. We are not aware of any published study that would monitor how people debug for so long while maintaining the details of collected data so large. Further pros and cons of our methods are mentioned in

[1] Interestingly, we were not able to locate any published evidence that would support the claim.

section 5. We believe that our methodology can server other researchers at least as an inspiration.

- We performed a usefulness evaluation of tools that were available to students of an operating system course at MFF UK[2]. The tools consisted of all the commonly available debugging tools and of some unusual debugging tools. Furthermore, we focused on comparing the usefulness of a command-line debugger (GDB) with usefulness of a GUI debugger (based on the Eclipse CDT plugin). After reviewing the collected data we reached to a rather surprising conclusion that both kinds of debuggers gave no advantage of faster debugging than the other alternative. See 4.1.2 for more details.
- During this exploratory study we collected multiple kinds of data. Therefore, in section 4.2 we provide an analysis that covers a spectrum of debugging aspects.
- We focused some of our analysis to uncover areas that would be worth of further research. The results are in the section 4.3.
- All the collected data are stored in the attached CD in the form of exported SQL database, and the attachment 3 describes the structure of the data. We did so, because some researchers could be interested in those data.

2.2. How to use this work

If you are a programmer and you intend to just have a quick overview of this thesis, we would suggest you to go right into the result section 4 or check the table of contents for aspects of debugging that interests you. If you are in a position of software company management you may find value in the thoughts on how to evaluate usefulness of tools in general (section 3.1), and in the results sections 4.1 and 4.2. Then, when any result catches your attention and you would like to make serious decisions based on that result, we strongly recommend you to read the methods section 3, understand the specifics of the environment where the data were collected from, and become aware of weak points of our methods. That should help you to interpret our results most realistically and adapt our conclusions to your specifics.

If you are a researcher you may find value in the whole work. Specifically, section 4.3 of the results is likely to get your interest, the methods 3 section can serve as an inspiration for your research, and the 5 section provides a basic set of references to the related research.

[2] Charles University in Prague, Faculty of Mathematics and Physics

3. Methods

We start describing ideas behind our methodology from a rather generic discussion about usefulness evaluation, and then we describe how we applied the general approaches for the specific environment of our study.

3.1. How much useful is a screw-driver or a hammer for people? ...thoughts on evaluating usefulness of generic tools

Although it may sound simple, we would like to pinpoint how the usefulness of tools can be perceived by people. This can help us to be able to interpret better how the results of any usefulness evaluation reflect the reality. The first distinction is whether we perceive the usefulness as absolute or relative:

We define absolute utility of a tool as the benefit of using the tool minus costs of inventing, obtaining and maintenance of the tool.

We define relative utility of a tool against some other tool (or set of tools) by comparision whether the tool is better or worse than some other alternative, or by expression how much is the tool better or worse.

The strong point of the absolute measure of utility is that it fits very well for the purposes of cost/benefit analysis, which is a popular method for making rational decisions. Sometimes, it may be difficult to estimate the benefits of using the tool. For example, the method that was used in (1) can be to some help here too. The method aims to evaluate what are the consequences of living in low-trust environment in terms of money. Basically, the evaluator keeps asking evidence questions (e.g. *How often does it happen?* or *Who does that job?*) and impact questions (e.g. *What are the consequences of not having that possibility?*) until he reaches the costs or benefits in terms of money.

When the estimation of absolute measure of utility is not possible or not reasonable to be done, we may become satisfied with the relative measure. The main benefit of this measure is that it can be often estimated just by simple observations and intuition or, more scientifically, by using statistical tests. When interpreting such relative comparision of usefulness we should be aware how well does the alternative match with our situation, or whether the comparision includes both the costs and benefits of the tools.

Regardless of whether the utility is measured absolutely or relatively we have identified the following factors that, in our belief, have influence on the usefulness of tools:

- The tool is actually being used. The justification behind this factor lies in the common-sense assumption that the less often a tool is used the less benefit it generates.
- The tool helps when it is being used.
- The tool is comfortable, improves the work satisfaction, or it makes people less tired.

Clearly, these factors are to a large degree independent. We can have a tool for very generic purposes (such as a kitchen knife) and a tool for rather special purposes (such as a bread slicing machine), which may be used much less often than the generic tool, but in some specialized scenarios it helps much more. Therefore, we may consider both of these tools to be useful regardless of what factor they fulfil better.

One could be tempted to construct a mathematically formal utility function out of these largely independent variables that would be used to compare the tools according to their usefulness. We abandoned that approach as in our area of interest we found no strong benefit of having the tools sorted by their usefulness. For other areas such as management-like or political decisions (e.g. *Which tool to buy?*) we suggest to use the absolute measure of utility if possible, and if not, use the multi-criteria analysis (2) to the identified factors of usefulness.

In the following text, we will transfer the ideas of generic tools evaluation into the specific area of debugging tools. For each generic factor of usefulness, we determine a set of values that tools can have and criteria for assigning these values. The values and criteria are described in the table 1, table 2 and table 3. We believe that this way of presenting results is better suited for informative purposes of this thesis rather than presenting raw numbers.

Tool usage frequency	Criteria
Often	The tool was used for 10% or more of investigated issues
Sometimes	The tool was used for 2% or more up to 10% of investigated issues
Rarely	The tool was used for less than 2% of investigated issues

Table 1: Informative values of how often is a debugging tool used

Tool helpfulness	Criteria
Very helpful	More than 60% of tool usages were perceived as very helpful
Questionably helpful	More than 60% of tool usages were perceived that the tool did not help at all
Somewhat helpful	The remaining case

Table 2: Informative values of how was the usefulness of a debugging tool perceived by participants

The helpfulness of the tool usage was perceived and recorded directly by participants. They were choosing between the values *Helped a lot*, *Helped a little* and *Did not help*. The more exact meaning of these values is located in the section 3.3.3.7.

Tool specialization	Criteria
Specialized	There is a debugging intent that is being performed in at least 80% of cases by the tool
Generic	The remaining case

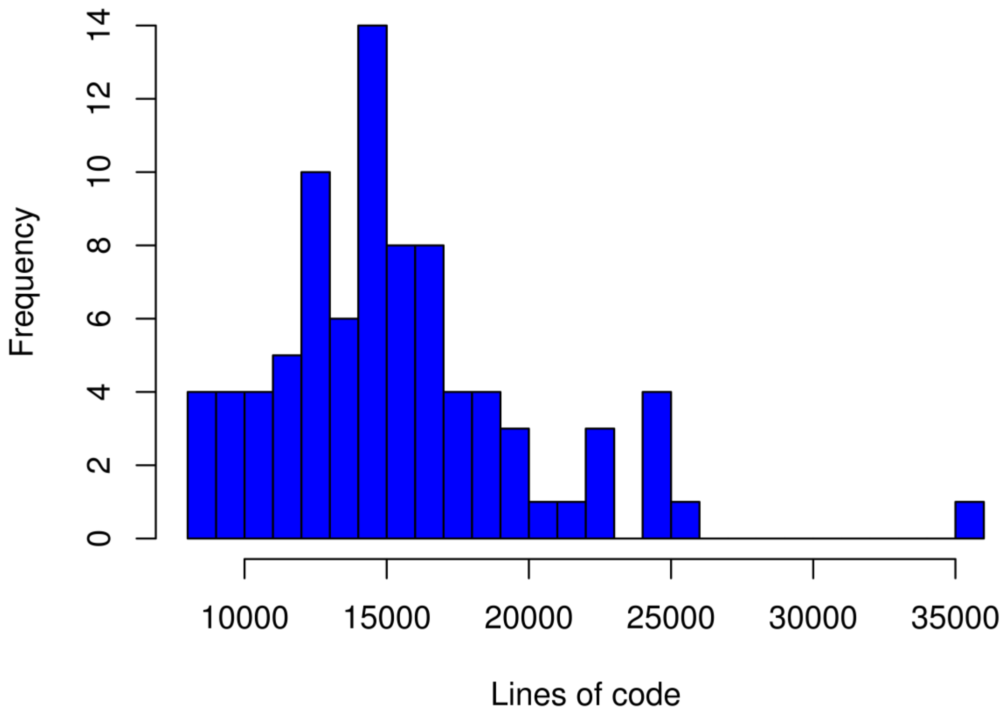
Table 3: Informative values of how specialized was a debugging tool

We introduced the factor of specialization in the table 3 to avoid evaluating a tool with specialized usages as less useful than it is in reality. Note that the result of the proposed criteria is highly dependent on the way of grouping the debugging tools. For example, one could group all the kinds of debuggers together, or one could decide that there will be two groups of debuggers (GUI debuggers and command-line debuggers). In the first case there will be much higher chance that debuggers will be evaluated as a specialized tool for, let's say, finding out the value of variable than in the latter case. Therefore, our way of grouping should be reviewed to check if it still fits the purpose the evaluation will be used for. In this exploratory study we cannot know what specific purpose this usefulness evaluation is aimed for, so we chose one decent alternative.

The last identified factor that is related to the impact on work-satisfaction, tool comfort and (mental) energy required to use the tool is not examined in this work. We believe that these aspects cannot be sufficiently studied just by self-observations of participants and that some assistance of researchers would be needed during experiments. Thus, we think that monitoring these aspects would require us to commit more resources to this work than is reasonable in our situation.

3.2. Environment of data collection

The participants of this study are students of a course of operating systems at MFF UK. During the course the students are supposed to get more familiar with concepts of operating systems and improve their low-level programming skills. Therefore, this study is closely related to the area of low-level programming. Many students perceive the course as one of the most difficult programming courses of the faculty. That is because of demanding amount of work and environment where debugging is unusually hard. The typical amount of time that each student performs programming tasks moves between 120 and 350 hours[3]. The students work usually in teams of three (or much less often in two or four) members. In order to illustrate to the size of the project we provide the picture 1, which summarizes the lines of code that the resulting software has had so far with 14686 lines of code as the median value.



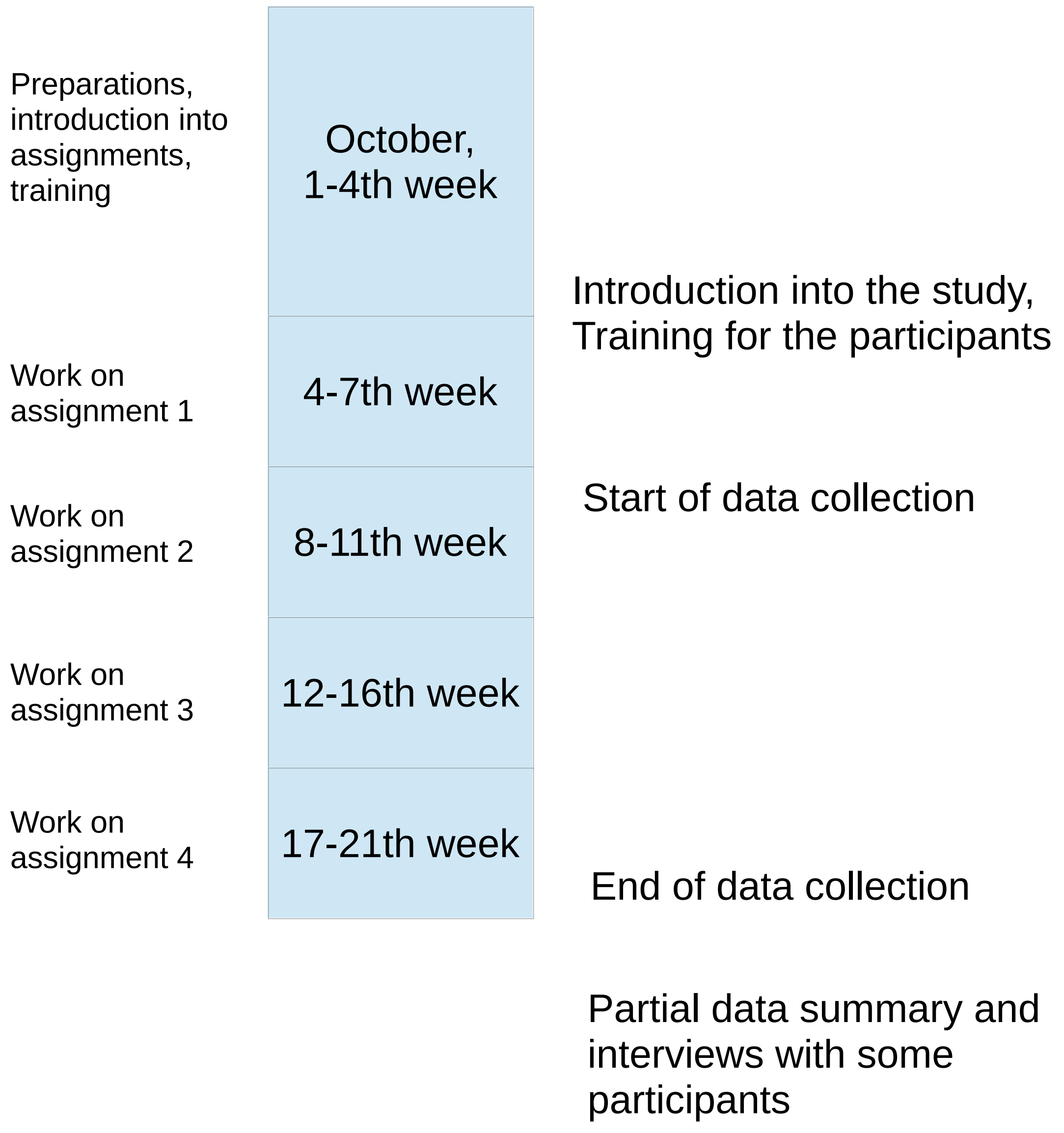
Picture 1: Size of the whole project for various teams in LOC

The students must do four assignments in order to pass the course successfully. After implementing all the assignments, the students end up with a minimalistic operating

[3] We cannot explain why the variance is so high and we noticed that even a technically highly skilled student reported 351 hours of implementation efforts, one possible explanation is that they aimed for higher quality.

system that satisfies a simplified version of POSIX API in the areas of threading, processes, memory allocation, and synchronization. The implemented code is executed on a virtual machine called MSIM. MSIM emulates a simple computer that is based on a MIPS R4000 processor. For a detailed description of the programming assignment and the course see the attachment 1.

The high difficulty and benefit in understanding operating systems of this course is well-known among students and they have the option to avoid this course. Therefore, most attending students are highly motivated to master the topic. Additionally, weak students in programming attend this course only rarely and some attending students have even a few years of professional programming experience. Thus, we believe that the students/programmers attending to this course represent real-world low-level programmers as much as is possible in academic environment. We consider this important, because cooperation with these students makes our results much more applicable to the realistic situations where skilled programmers are employed.



Picture 2: Time schedule of the whole project and events related to this study

3.3. Data collection

3.3.1. Periods of data collection

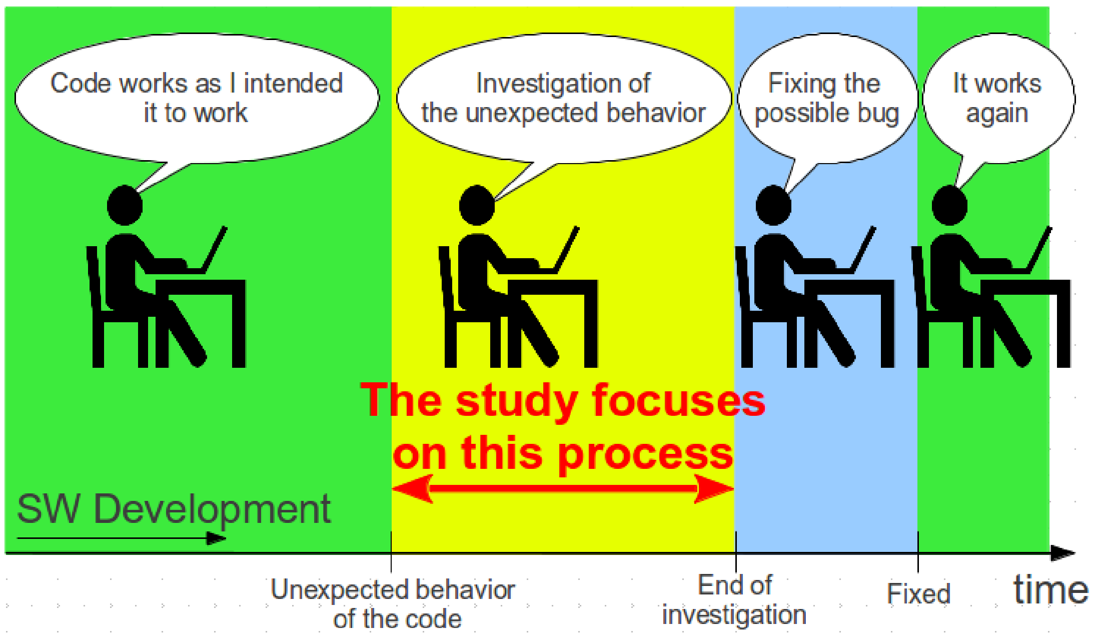
In order to collect as much data as our resources allowed, we performed the study during the winter school terms of the years 2011, 2012 and 2013.

Picture 2 shows the schedule of a single term with important events and deadlines marked. During the years 2011 and 2012 the participants recorded complex information about their debugging activities, because we aimed to explore many kinds of data. The detailed description of the recorded data is located in the section 3.3.3. In the year 2013 the data collection was restricted just to obtain information about mapping of what debugging intents programmers have and what debugging tools they use to perform those intents. The table 4 contains how many students participated in the study. The participation was voluntary and the participated students were given a small bonus during evaluation of their assignments.

Year of the study	Count of participants	Focus of collection
2011	9	Generic data
2012	6	Generic data
2013	5	Debugging intentions

Table 4: Count of participants and focus of this study during years

3.3.2. How the data were collected


Code works as I intended it to work
Investigation of the unexpected behavior
Fixing the possible bug
It works again
The study focuses on this process
SW Development
Unexpected behavior of the code
End of investigation
Fixed
time

Picture 3: Specification of what process this study aims to explore[4]

This study aims to explore the process of how programmers debug computer programs. To define this process more exactly we consider debugging as actions that are performed when the programming code does not behave as the programmer expects and ends by explaining the unexpected behavior or by abandoning the efforts. Some people

[4] We reused and adapted the image of programmer by Hadi Davodpour from the Noun Project, which is published under the Creative Commons 3.0 license.

may not see an exact fit with their perception of debugging and our definition, because our definition, for example, allows that there is no actual bug in the software and the programmer can still debug it. We choose not to refine the terminology further as we found no strong reason in doing so for the context of this work.

To give some better feeling of what we consider debugging, we give three examples:

- The programmer executed a piece of code with a belief that it should behave in some way and it actually behaves in another way. We consider this activity to be debugging.
- Somebody else reports to the programmer that the program behaves in a faulty way and the programmer starts to reason about it. We consider this activity to be debugging.
- The programmer is searching for bugs in the code without any attempt to execute the code. We do not consider this activity to be debugging.

In this study the participants recorded information about every debugging activity they encountered as they were working on their regular assignments. The records were filled into a prepared web interface every time the participants finished investigation of any unexpected behavior of their code.

3.3.3. Description of collected data

In the following text we explain what kind of data was collected for each debugging activity.

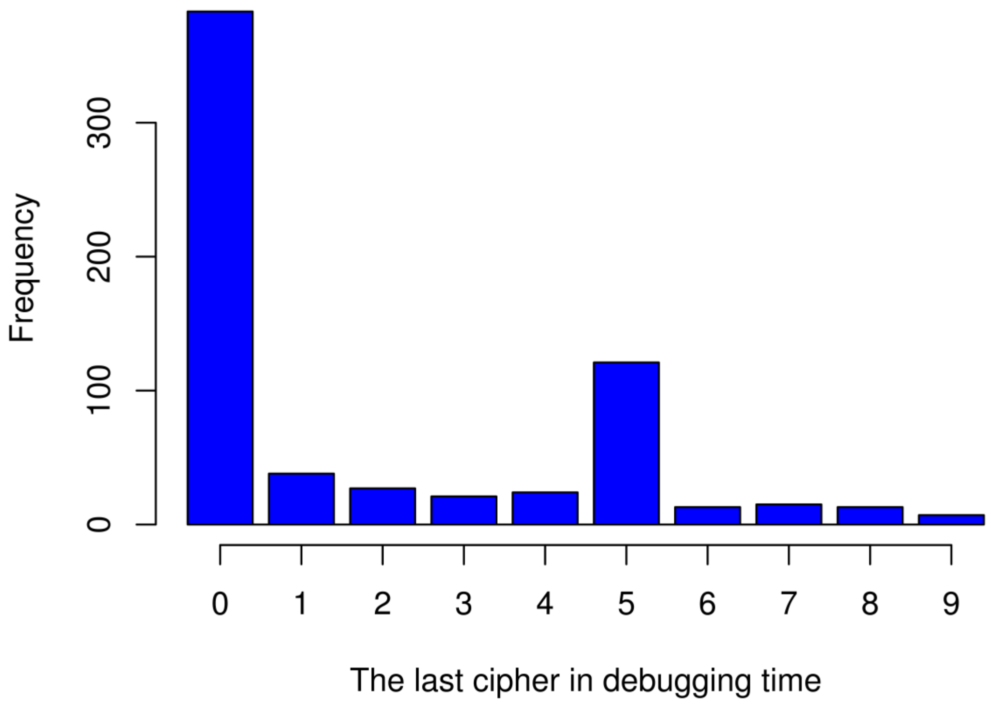
3.3.3.1. Project development time

This is a single time value that means how much effort each participant did on fulfilling the assignments. It includes all the development activities such as implementation, debugging, communication, handling emails or writing documentation. We recommended to participants to update this value every day when something was done.

3.3.3.2. Debugging time

When we use the term debugging time in this work we mean the time of investigation of an unexpected behavior. Maintaining this information for the whole process of a long investigation in precise manner could be too demanding for participants, so they were instructed to maintain high precision for short investigations and they were allowed to have 10 minutes error for investigation longer than 2 hours. In this uncontrolled experiment we also expected the participants to do rounding of these time values and we were not mentioning anything about how they are supposed to round. For example, we believed that participants will very likely round 28 minutes to 30 minutes. The main motivation behind keeping the methodology in this way was that in our opinion it was not reasonable to tie up the participants by many strict rules for these uncontrolled observations. We think that they would be more likely to stop their contribution to this study or the rules hard for maintenance could have a very strong effect on the experiment itself. What we did in order to increase the precision was encouraging the participants to note the time when they started to investigate some unexpected behavior.

The histogram in the picture 4 suggests that the debugging time values were indeed rounded very often.



Picture 4: How often was the debugging time rounded

3.3.3.3. Complexity of the debugging scenario

We wanted to obtain information of how difficult a particular debugging scenario is. Therefore, we instructed the participants to record their opinion of the difficulty of the debugged issue. To define the object of such observations less vaguely we expressed the complexity as follows:

We consider a complexity of the debugging scenario as the amount of thinking that is needed to understand the situation and for the analysis of the problem. It does not necessarily have to reflect the debugging time of the issue. The table 5 lists the values and provides examples of debugging scenarios with different complexities.

Complexity	Description and an example
Trivial	Requires only little or less thinking. Checking code that implements a straight-forward idea. No large pointer manipulations. No complex conditions. No recursion. Fixing a wrong return value in a function that loads configuration.
Easy	Requires some thinking. Insertion to a link list.
Medium	Making charts or notes starts to be useful. Insertion to an AVL tree. Searching for a bug in the operating system page tables code.
Hard	Requires an analysis or the programmer thinks a lot. Searching for a race condition that cannot be reproduced easily.

Table 5: Description of complexity of debugging scenarios

This distinction of complexities was introduced and explained to the participants. It should have provided them the boundaries that would help them decide which complexity to choose.

During the course of the study we started to have concerns whether the recorded data correspond correctly to its supposed meaning. The evaluated data in 4.2.7 suggest that the recorded complexity almost linearly depend on the debugging time. One explanation for this is that the participants tended to perceive the complexity of the scenario according to the time needed for investigation. In such a case we would measure something else than we originally wanted. As we do not have evidence to justify this concern, we must treat the meaning of these recorded data more generally as a not very specifically defined difficulty of a debugging scenario. In order to capture the data with the intended meaning we would need to modify the methodology of the data collection. Likely, the experiment would have to be more controlled.

Also note that for the reasons explained by the previous paragraph we use terms complexity and difficulty interchangeably in this work.

3.3.3.4. Feelings from the debugging activity

During designing of this study we decided to collect data about how the amount of job-related hardship corresponds to work satisfaction of low-level programmers. To increase objectivity the participants were instructed to reflect their feelings in the following situation:

Suppose that today is some another day in the future. You arrive to your work, do some programming for an hour or two, exchange a few interesting news with your colleagues and then you encounter the similar problem just like what you have just solved. How would you feel?

The reasoning behind making the participants imagine the described situation is that for our observations we wanted to minimize the impact of the psychological peak-end rule (3)[5] that could be combined with the pleasure gained by discovering the explanation of the unexpected behavior. We thought that this effect would be only short-lasting (no more than a week in duration), so we aimed to set the time of reflection into distinct future in order to minimize the likely positive effect of task achievement. Therefore, we promised ourselves that we would achieve a better level of objectivity.

During final interviews with participants we found that they often did not manage to reflect their feelings in the intended way. They explained what they recorded by using sentences like *I reported the positive feeling, because I have just finished a difficult task successfully*. Therefore, we believe that the effect of goal achievement was significantly influential regardless of our effort to minimize it and we must treat the collected data as it would have a more generic and vague meaning of work-satisfaction. Similarly as for the complexity of a debugging scenario, we believe that collecting this data with a more refined meaning would require changes in methodology.

3.3.3.5. Way of detection

As we were preparing this study, we found no published evidence about the ways how programmers become aware of bugs and, more importantly, how frequent these bug

[5] The so called peak-end rule is a psychological pattern of how people tend to memorize intensive experience (both pleasant and unpleasant). The rule claims that the most relevant for later judgement are the moments of the peak intensity and the end of the experience rather than the average of all the moments. For our case that means that the end of the experience (i.e. finishing a difficult task) is very likely to have strong influence on work-satisfaction of the programmers, because the success could eliminate the effect of most unpleasant experience.

detections happen. Therefore, we included collection of these data into the study as we believed that it may become relevant for some management-like decisions in companies such as designing software quality-assurance processes.

The complete list of identified ways of detection is in the attachment 5. The participants were supposed to choose one way of detection from this list for every record of investigation. Furthermore, the participants recorded additional information for each record, which captured further characteristics of the detection (such as *The bug was detected during debugging of another bug*). The web interface contained predefined checkboxes that allowed the participant to record the additional information. The list of these options is in the attachment 5 too.

3.3.3.6. Root cause

We collected the data about root causes of bugs in order to map how the individual root causes are related with debugging time and how frequent they are. (4) mentions that the selection and categorization of root causes is typically done with regards to the purpose of the study rather than by choosing an existing well-established categorization. As the purposes of our study are exploratory, we just desired the categorization to be comprehensive well enough. We got inspired from (5) and adapted it to fit better the environment of low-level programming (e.g. we added *Violation of ABI rules* root cause). The root causes are grouped into three main categories *Wrong logic or design*, *Wrong implementation* and *Other*, and we created subcategories for each of these categories in order to make orientation in the root causes faster for the participants.

Participants were instructed to distinguish between the two main categories according to the following rules. If the issue was caused by incorrect thinking they should select the root cause from the *Wrong logic or design* category. If the issue was correctly designed or thought, but incorrectly implemented they should choose from the category *Wrong implementation*. The main category *Other* contains root causes that hardly fit these rules such as *Not a bug* root cause.

The list of identified root causes is identified in the attachment 6.

3.3.3.7. Used methods and tools

The participants were supposed to record what methods (i.e. debugging approaches) and debugging tools they used to investigate the unexpected behavior. For each such usage they selected what amount of usefulness the method or tool brought. The table 6 lists the criteria for selecting the usefulness. We specified this table in order to maintain a reasonable level of objectivity of these data and to provide guidance to participants how to choose the amount of usefulness. The participants were asked to use their own judgement in blurred cases.

Usefulness	Criteria for selecting
Did not help	The usage brought no useful information for the investigation or it lead to an inconclusive dead end.
Helped a little	Anything between the other two values. The usage brought some minor information that was to some use. For example, where not to search for the error.
Helped a lot	The usage helped significantly with the investigation. For example, it led directly to explaining the unexpected behavior, or the programmer would hardly investigate the problem without the tool.

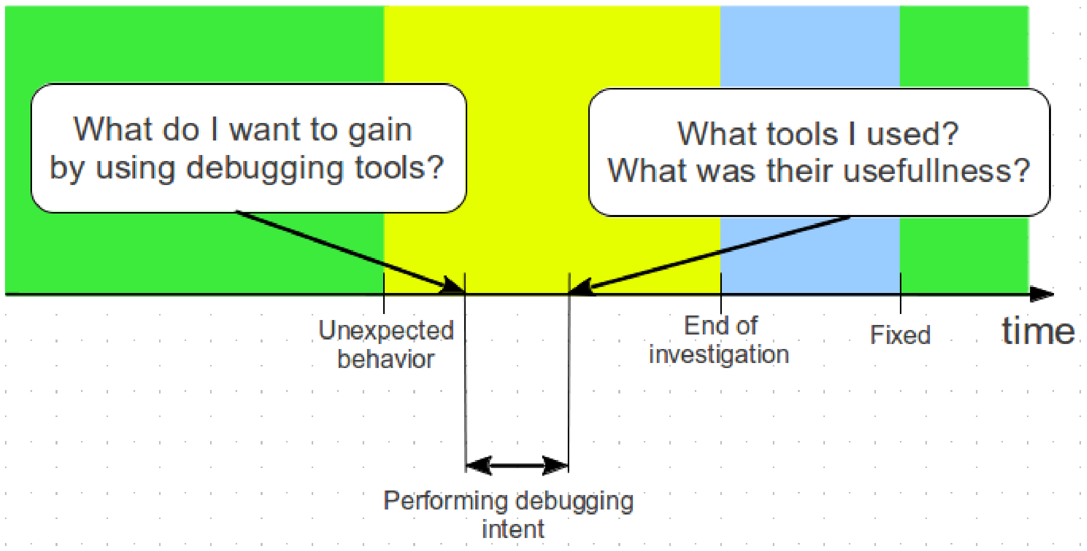
Table 6: Values of usefulness for a particular tool usage and criteria for selection

The full list of identified methods and debugging tools is located in the attachment 7.

3.3.4. The last year of the study

In 2013 we changed the methodology significantly to obtain a different kind of data for tool usefulness evaluation. We wanted to map how frequent the debugging intents of the programmers are (such as *I want to know the value of some variable*) and how useful the available debugging tools are to fulfil those intents.

What changed was the moment when the data were supposed to be recorded. We instructed the participant to record the intent for using a debugging tool after each its usage. They participants recorded the debugging intent they were trying to perform, and they selected the debugging tools that were used to achieve the intent with usefulness for that particular usage. This is illustrated on picture 5, which was shown to the participants too.


What do I want to gain by using debugging tools?
What tools I used? What was their usefullness?
Unexpected behavior
End of investigation
Fixed
time
Performing debugging intent

Picture 5: Specification of what process this study aimed to explore in 2013

Also note that in this year we stopped collecting the data described by 3.3.3, because we wanted to focus on obtaining just the data about the debugging intents. In 3.6.4 we explain why collecting just one thing matters.

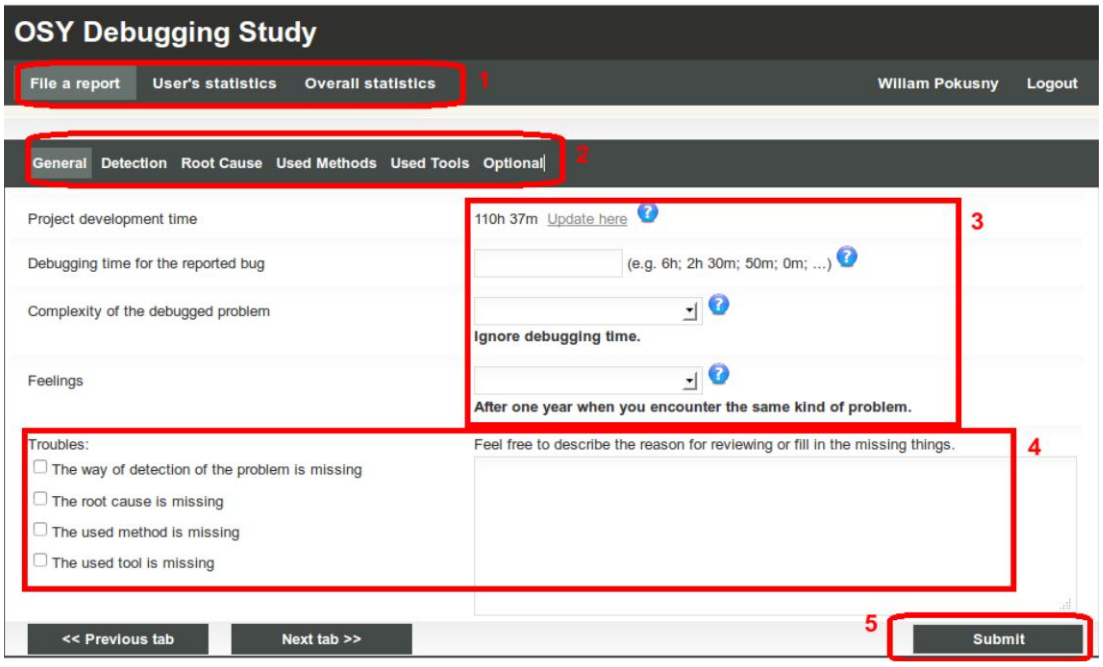
3.4. Training of the participants

The participants received training to help them with the assignments and to instruct them how to fill the records for the study. Furthermore, for both purposes they were introduced how to use the GNU Debugger (GDB), MSIM Graphical debugger (i.e. eclipse-based extension of GDB for MSIM) and some binutils (e.g. objdump).

We asked the participants to provide us feedback of the methodology of the study and to inform us if any way of detection, root cause, method or debugging tool was missing in our study. We can recommend this practice, because during the course of study we received several suggestions and warnings about the validity of the obtained observations. Some suggestions were incorporated in the study early enough (e.g. a missing tool or debugging intent) and the warnings about flaws in our methods that we were not successful to cope with are mentioned here in the text (e.g. in 3.6.4 how the large amount of debugging reports caused many interruptions of regular programming work).

3.5. Web interface for data collection

In the following we will go through the most relevant elements of our web interface that was used to record data. We will mention our motivation behind the design of some elements and the experience with the design.


OSY Debugging Study

File a report | User's statistics | Overall statistics — Wiliam Pokusny | Logout

General | Detection | Root Cause | Used Methods | Used Tools | Optional

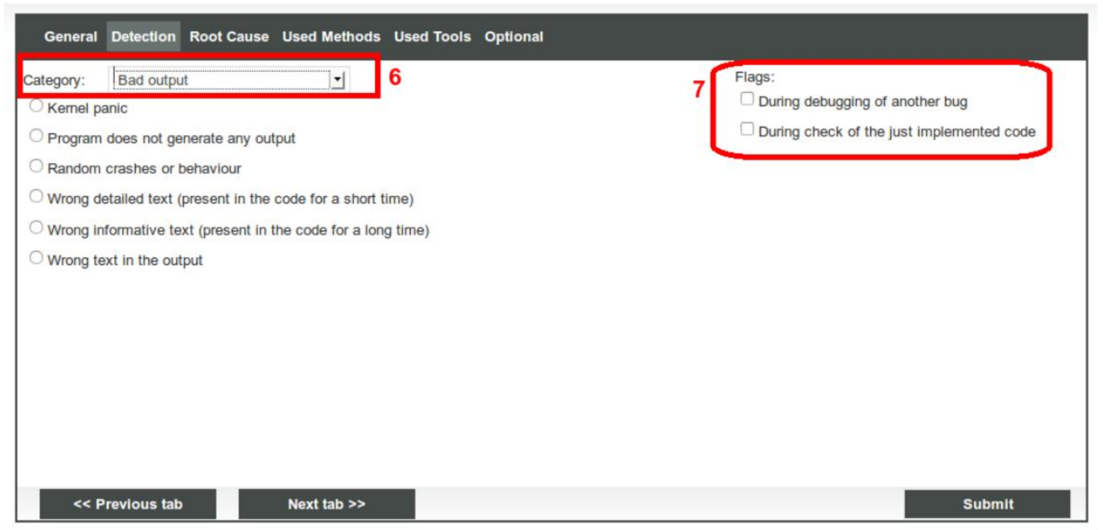
Project development time	110h 37m Update here
Debugging time for the reported bug	(e.g. 6h; 2h 30m; 50m; 0m; ...)
Complexity of the debugged problem	Ignore debugging time.
Feelings	After one year when you encounter the same kind of problem.

Troubles:
- ☐ The way of detection of the problem is missing
- ☐ The root cause is missing
- ☐ The used method is missing
- ☐ The used tool is missing

Feel free to describe the reason for reviewing or fill in the missing things.

<< Previous tab | Next tab >> | Submit

Picture 6: The main page of the web interface that was used for data collection

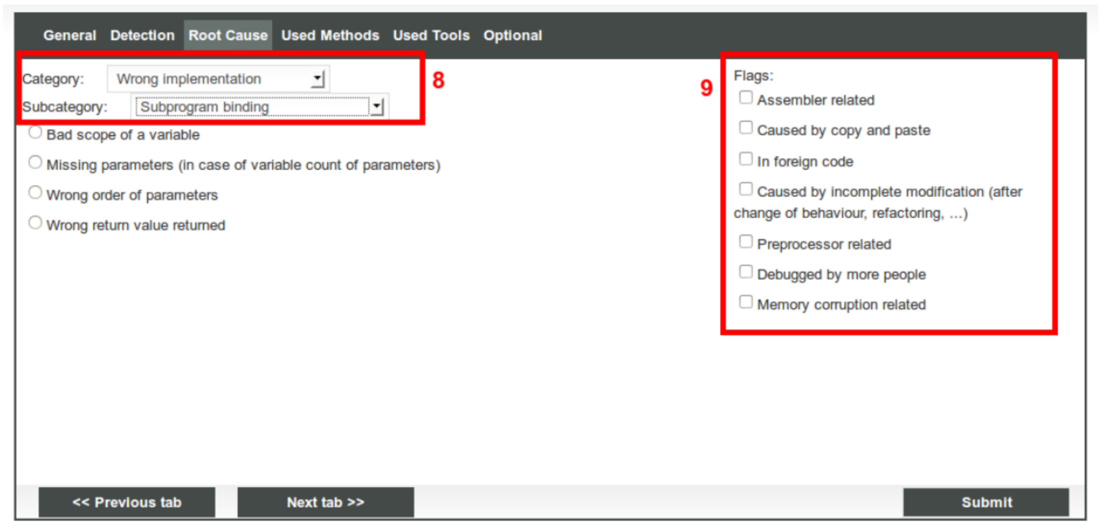

General | Detection | Root Cause | Used Methods | Used Tools | Optional

Category: Bad output
- ○ Kernel panic
- ○ Program does not generate any output
- ○ Random crashes or behaviour
- ○ Wrong detailed text (present in the code for a short time)
- ○ Wrong informative text (present in the code for a long time)
- ○ Wrong text in the output

Flags:
- ☐ During debugging of another bug
- ☐ During check of the just implemented code

<< Previous tab | Next tab >> | Submit

Picture 7: The tab for reporting information related to the way of detection of the unexpected behavior



Category: Wrong implementation

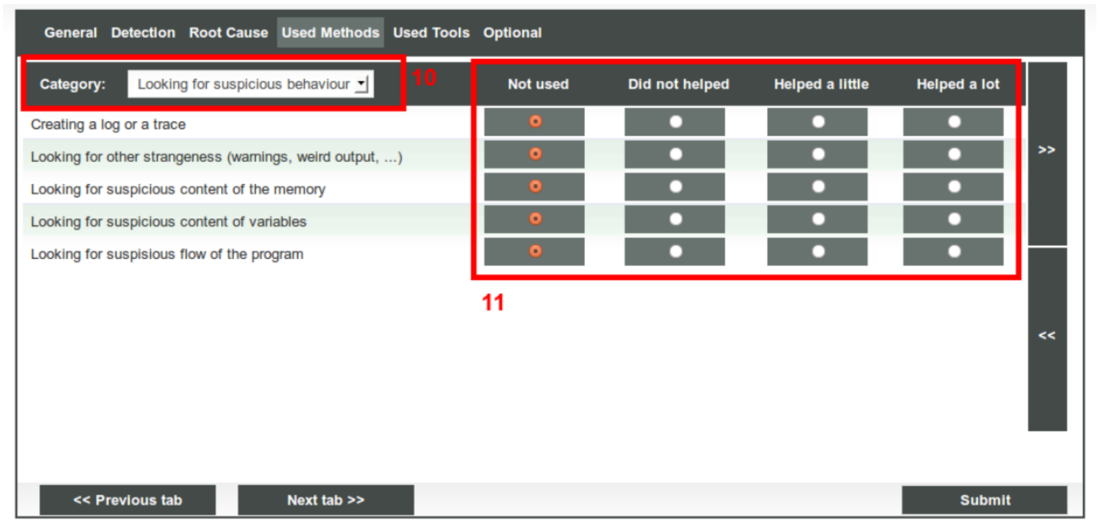
Subcategory: Subprogram binding

- ○ Bad scope of a variable
- ○ Missing parameters (in case of variable count of parameters)
- ○ Wrong order of parameters
- ○ Wrong return value returned

Flags:

- ☐ Assembler related
- ☐ Caused by copy and paste
- ☐ In foreign code
- ☐ Caused by incomplete modification (after change of behaviour, refactoring, ...)
- ☐ Preprocessor related
- ☐ Debugged by more people
- ☐ Memory corruption related

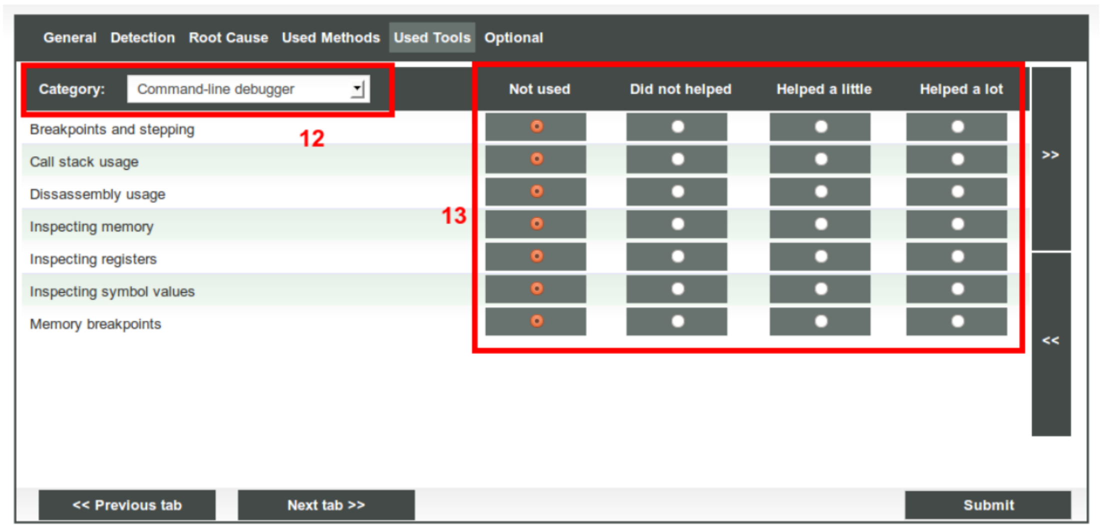
Picture 8: The tab for reporting information related to the root cause of the unexpected behavior



Category: Looking for suspicious behaviour

	Not used	Did not helped	Helped a little	Helped a lot
Creating a log or a trace	●	○	○	○
Looking for other strangeness (warnings, weird output, ...)	●	○	○	○
Looking for suspicious content of the memory	●	○	○	○
Looking for suspicious content of variables	●	○	○	○
Looking for suspisious flow of the program	●	○	○	○

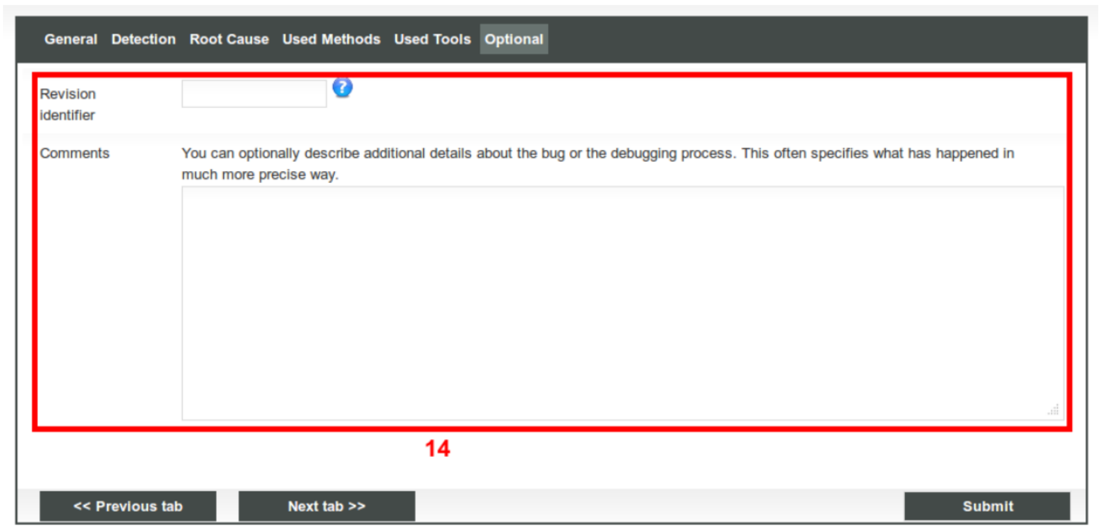
Picture 9: The tab for reporting information related to the used methods for investigation of the unexpected behavior



Category: Command-line debugger

	Not used	Did not helped	Helped a little	Helped a lot
Breakpoints and stepping	✓			
Call stack usage	✓			
Dissassembly usage	✓			
Inspecting memory	✓			
Inspecting registers	✓			
Inspecting symbol values	✓			
Memory breakpoints	✓			

Picture 10: The tab for reporting information related to the used tools for investigation of the unexpected behavior

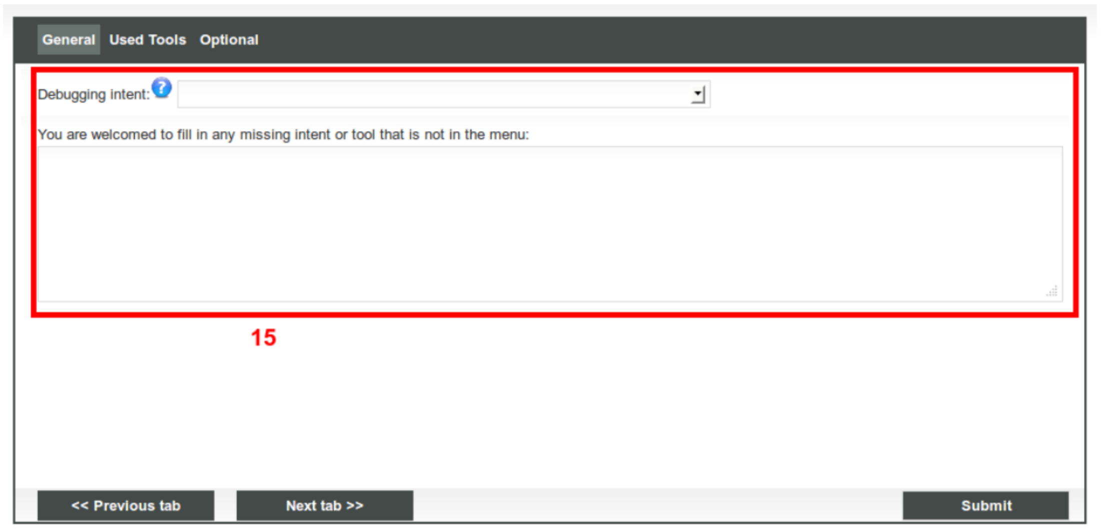


Revision identifier

Comments

You can optionally describe additional details about the bug or the debugging process. This often specifies what has happened in much more precise way.

Picture 11: The tab for filling optional information about the report


General Used Tools Optional

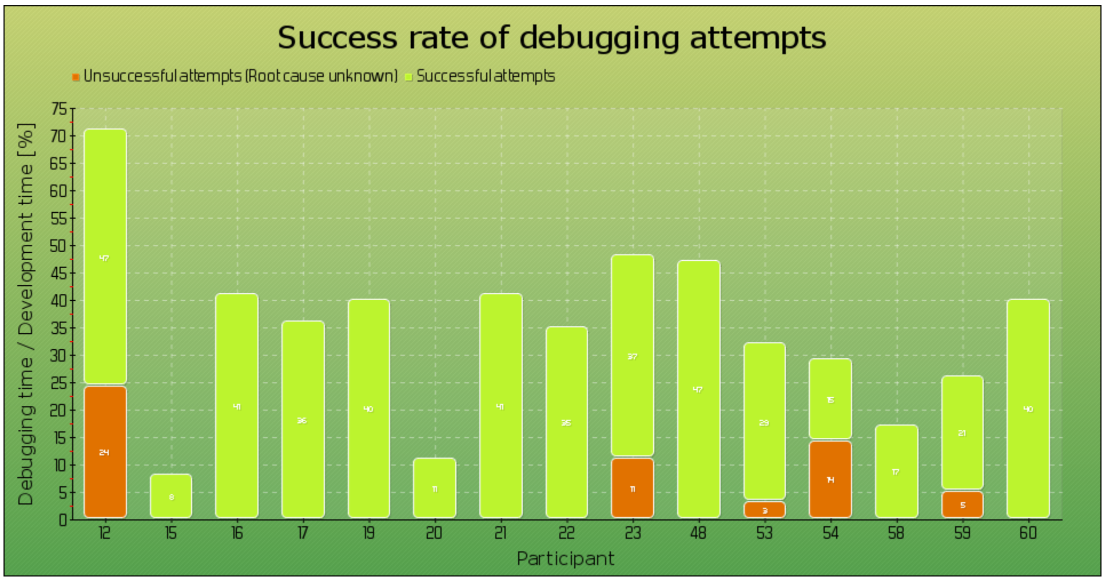
Debugging intent:

You are welcomed to fill in any missing intent or tool that is not in the menu:

15

<< Previous tab Next tab >> Submit

Picture 12: The main tab of the study in the last year of the study



Picture 13: One of the graphs in the web interface; this one shows how much debugging resulted in abandoning the investigation

We refer to the numbered red frames in pictures on pages 14-17:

- Frame 1 - the main menu of the web interface. We made significant effort (60 hours of work) to display summarizing statistics during the course of study in an attractive way, which was supposed to make the study more attractive in overall. For example, see picture 13 for the having an idea of what level of the look-and-feel was achieved. After the first year of the study we concluded that the efforts required to maintain and develop further nice-looking summaries of data are unreasonable for the study of our scale. The only exception was creating a view that enabled each participant to see all his reported data, which helped a lot with communication whenever there was an unusual debugging report.
- Frame 2 - tabs of the debugging report form. During the first two years of the study we collected several various kinds of data (see 3.3.3). Filling all the data for each debugging record could become too demanding for participants, so we were focused

to make the user interface as much comfortable as it was reasonable to do. The different kinds of data were placed on different tabs and the participants could navigate quickly through these tabs via clicking on the tab title or via the previous and next buttons.

- Frame 3 - generic information about debugging report. The participants filled information that was described in 3.3.3.1, 3.3.3.2, 3.3.3.3 and 3.3.3.4. For making sure that the instruction will not be forgotten we created floating help with the instructions about meaning of the fields and how they are supposed to be filled.
- Frame 4 - space for feedback and suggestions. This was mainly supposed to be used to prevent missing items in categorization of root causes, ways of detection and so on... During the course of the study we received about 10 improving suggestions and we reviewed 29 out of 662 debugging records. The interface allowed to submit an incomplete record only if this field had some content.
- Frame 5 - the submit button. The interface prevented the participants to submit invalid data and in such a case it printed verbose and usable information about what part of the form are filled wrongly. This was designed in order to save participants as much time as possible with troubleshooting what is wrong. In the correct case the web interface printed a message that the record has been submitted successfully.
- Frame 6 - the combobox for selection of the way how the unexpected behavior was detected (as described in 3.3.3.5). The combobox itself contains just a category and participants should choose the specific way of detection by choosing the appropriate radio button below. The radio buttons changed according to the selected category. This was designed to allow the participants quickly browse through all the options.
- Frame 8 - comboboxes for selection of the way what is the root cause of the unexpected behavior (as described in 3.3.3.6). As our hierarchy of root causes is large we used two comboboxes. The first one for the top-level category (*Wrong implementation*, *Wrong logic or design*, *Other*) and the second for subcategories. This specific way of user interface had its role. Some root causes are very similar (for example *Wrong implementation* → *Wrong program flow* → *Wrong order of commands* and *Wrong logic or design* → *Data structures and algorithms* → *Sequence of actions designed in a wrong way*) and we wanted participants to decide to which top-level category the root cause belongs before selection of the specific root cause. Thus, we prevented the participants to select some similar root cause from a wrong category.
- Frames 7 and 9 - flags related to the way how the unexpected behavior was detected, or to the root cause. We desired to obtain some boolean-typed information for each report. Therefore, we introduced a set of checkboxes to the relevant tabs.
- Frames 10 and 12 - comboboxes for selection of the category for the used debugging methods or tools (as described in 3.3.3.7). Selecting a different category changes the list of possibilities. This was designed to fit the list of options into a single tab, so the participants would not get slowed down by scrolling.
- Frames 11 and 13 - arrays of radio buttons that indicate whether a particular debugging method or tool was used. The default option has the value *Not used*, because the participants usually will want to select only a few options and leave the majority of options in the *Not used* state.
- Frame 14 - the optional tab for filling details whenever doing the report required further discussion with the researchers. This helped a lot with effective troubleshooting of unusual situations.

- Frame 15 - the main tab of the last year of the study (2013). The content was minimalistic. The main user interface element on this tab was the combobox for selecting the debugging intent for using the debugging tool.

3.6. Further thoughts on our methodology

In the following subsections we will look on our methods from other viewpoints.

3.6.1. Realistic environment

We designed our study to collect the data from as much realistic environment as was possible in our conditions. This brings both pros and cons. One strong advantage is that programmers record data from real-world situations. Therefore, we consider the validity of our data to be resistant to the effect of collecting the data from an unrealistically designed controlled experiment, which usually happened, for example, in the area of evaluating usefulness of automated debugging tools (6). See the section 5 for further comments on this.

The second advantage is that we aimed to test the usefulness of debugging tools when all commonly debugging tools were available and participants had a completely free will, which tool to use. Thus, in our study we do not compare just one tool with one possible alternative, but rather we evaluate tools in the competition of other available tools. We designed this with the assumption that skilled IT professionals will always choose the tool that is most useful for the job. Although originally, we did not plan to systematically monitor how is this assumption correct, we managed to get some observations that suggest that this assumption is not so strongly held in reality. We present these observations in the results section 4.3.8.

3.6.2. Exploratory study versus aimed study

The exploratory design of this study also brings some pros and cons. The advantage is that the study can cover much more aspects of debugging than a more focused study, and it can more efficiently pinpoint areas worth of further research. On the other hand, we do not have a very specific goal (such as main questions and related hypotheses to prove) and therefore, we risked that the study will yield results with only little value. Thus, for the worst-case situation we expected that our results would contribute at least to some of the following points:

- We would obtain new data on aspects that have been already researched.
- We would obtain data that will confirm something that people know intuitively, but it has not been scientifically proven so far.
- We would obtain data that will map something that people cannot know intuitively.
- We would warn other researchers about unexpected flaws in our methodology, so they could be at least aware of them since the beginning of their studies.
- We would pinpoint research areas that are, in our opinion, mostly worth of further research efforts.

Ideally, we would be glad to provide some results with higher value such as:

- We would obtain data that will invalidate something that people know intuitively.
- Based on our data we would propose some way of improvement on existing tools, methods or approaches.

3.6.3. Uncontrolled experiment

There are two major reasons why we chosen to have the experiment in an uncontrolled way (i.e. the participants collect the data without presence of the researching observer). Having this study in a more controlled way would be much more demanding on our resources and it could violate privacy of participants unacceptably. The main disadvantage of this approach is that we have only little means to check whether the participants filled the data in the desired way. For example, some debugging reports could be omitted or the data could be filled inconsistently (such as that two almost identical bugs are recorded in a different way).

3.6.4. Amount of collected data

According to the feedback of participants, doing a single report about data of 3.3.3 took approximately 3 minutes when the participants got experienced with the reporting web interface. One participant explicitly mentioned that besides those 3 minutes the effect of interrupting his work brought much bigger inconvenience for him and that it reduces his productivity.

Therefore, the method of making the participants record their experience could influence our experiment by reducing their productivity. Thus, when using this approach of collecting the data this factor should be taken into consideration. For our study we believe that it is acceptable to make the participants collect a larger amount of data, because our goals are more exploratory than narrowly focused.

3.6.5. Interviews

We interviewed five participants when they finished their project. Because this study was long-term and exploratory oriented we did not have a plan or criteria how to do the interviews. The main goal of the interviews was asking the relevant participants about their experience of results that had been identified up to that time. Therefore, we could more easily interpret the results and formulate hypotheses about the data more accurately. These interviews also helped us to become more aware about flaws of our methods.

3.6.6. Hypothesis for testing

Regardless that the study was designed to be mainly exploratory, we actually did have three hypotheses that we wanted to statistically test:

1. Test that using a GUI debugger leads to faster bug investigation than using a console debugger or just debugging messages.
2. Test that design-time bugs (errors in logical reasoning) are more time-consuming for investigation that bugs that are caused by wrong implementation of correct ideas.
3. Test that the bugs that are related to assembler are more time-consuming for investigation than bugs that are not related to assembler.

3.6.7. Time resources needed for doing this study

Researchers may be interested how much time-demanding is it to perform a similar study. Based on our experience, we present the resource estimation in the table 7.

Activity	Resource estimation and comments
Studying related work	120 man-hours
Design of methods	40 man-hours
Implementation of web interface and its maintenance	160 man-hours
Participant training and communication with them	1 man-hour for each participant and 30 man-hours for researchers. We cooperated with 20 participants.
Work of participants	In average 3 man-hours per participant. This does not include the effect of interrupting their programming activities, which we expect to be much more relevant to them.
Data analysis	140 man-hours. As this is an exploratory study many uninteresting views on the collected data are expected.
Writing the report	120 man-hours
Sum	In total 610 man-hours for researchers and 60 man-hours for participants.

Table 7: Human resources estimation for performing a study like this one

4. Results and interpretation

We organize the results into three parts: the usefulness evaluation as described in 3.1, further aspects of debugging, and data that could be valuable more-likely just to researchers.

The location of the data and the evaluation script is described in the attachment 2. In order to allow a highly detailed view on the way how we processed the data we put R snippets into the relevant places of presented data analysis. These snippets point out the reader into the *evaluation.R* script that is located in the attached CD, so they can inspect details of our evaluation or do their own evaluation. The snippets look in this way:

```
this is an R snippet of the evaluation.R script
```

4.1. Usefulness evaluation of debugging tools

4.1.1. Evaluation of all the available tools

Task

Evaluate the usefulness of tools that were available to students of the operating system course according to the methods from section 3.1.

Data analysis

The table 8 summarizes the collected data of monitoring how often were debugging tools used and how often were the usages helpful. These data consist of 662 debugging records.

Tool	Frequency [%]	Helped a lot [%]	No help [%]
Breakpoints and stepping, GUI	14.1	59.0	17.4
Call stack usage, GUI	0.0	NA	NA
Disassembly usage, GUI	5.8	64.4	15.3
TLB window usage, GUI	1.0	10.0	50.0
Memory breakpoints, GUI	2.7	39.3	32.1
Memory view usage, GUI	0.7	14.3	71.4
Registry window usage, GUI	1.5	13.3	73.3
Variables window usage, GUI	1.5	6.7	66.7
SVN (log, history)	6.6	36.8	30.9
Objdump	1.0	70.0	30.0
Text processing tools	3.1	50.0	15.6
Excel, R, own script, ...	3.7	47.4	5.3
Instruction-level stepping and breakpoints, MSIM	0.2	50.0	0.0
Memory dump, MSIM	2.4	64.0	16.0
Inspecting registers, MSIM	0.7	28.6	57.1
Special instructions of MSIM	2.4	44.0	40.0
Memory breakpoints, MSIM	4.2	48.8	18,6
Execution trace, MSIM	1.1	54.5	36.4
Breakpoints and stepping, GDB	4.6	46.8	23.4
Call stack usage, GDB	8.2	33.3	33.3
Disassembly usage, GDB	2.7	14.3	46.4
Memory breakpoints, GDB	2.0	45.0	40.0
Inspecting memory, GDB	1.4	28.6	42.9
Inspecting registers, GDB	3.0	35.5	35.5
Inspecting symbol values, GDB	1.4	28.6	64.3
Own functions (in code) for debugging	3.4	31.4	28.6
Own debugging programs	14.2	69.7	9.7
Reading documentation	1.7	88.2	5.9
Web search (google, forums, ...)	3.0	61.3	9.7
Questioning community	1.2	33.3	8.3
Tools for static analysis	0.7	71.4	14.3

Table 8: Frequency of debugging tool usages and their perceived usefulness

In the last run of the study when we focused on studying debugging intents, we found 4 tools that had a specialized usage. The table 9 summarizes the relevant data. The results are based on 138 records for 22 intents.

Tool with a specialized usage	Intent	Reported usages
SVN (log, history)	Investigate what has changed recently	1
Objdump	Search for a symbol name from address	1
Own functions (in code) for debugging	Investigate what is the program doing right now	1
Reading documentation	Search of error code meaning	2

Table 9: Tools with a specialized usage

Interpretation

According to the proposed methodology, we interpret the data in the table 10.

Tool	Frequency of using	Usefulness of a usage	Specialization
Breakpoints and stepping, GUI	Often	Somewhat helpful	Generic
Call stack usage, GUI	Rarely	NA	NA
Disassembly usage, GUI	Sometimes	Very helpful	Generic
TLB window usage, GUI	Rarely	Somewhat helpful	Generic
Memory breakpoints, GUI	Sometimes	Somewhat helpful	Generic
Memory view usage, GUI	Rarely	Questionably helpful	Generic
Registry window usage, GUI	Rarely	Questionably helpful	Generic
Variables window usage, GUI	Rarely	Questionably helpful	Generic
SVN (log, history)	Sometimes	Somewhat helpful	Specialized
Objdump	Rarely	Very helpful	Specialized
Text processing tools	Sometimes	Somewhat helpful	Generic
Excel, R, own script, ...	Sometimes	Somewhat helpful	Generic
Instruction-level stepping and breakpoints, MSIM	Rarely	Somewhat helpful	Generic
Memory dump, MSIM	Sometimes	Very helpful	Generic
Inspecting registers, MSIM	Rarely	Somewhat helpful	Generic
Special instructions of MSIM	Sometimes	Somewhat helpful	Generic
Memory breakpoints, MSIM	Sometimes	Somewhat helpful	Generic
Execution trace, MSIM	Rarely	Somewhat helpful	Generic
Breakpoints and stepping, GDB	Sometimes	Somewhat helpful	Generic
Call stack usage, GDB	Sometimes	Somewhat helpful	Generic
Disassembly usage, GDB	Sometimes	Somewhat helpful	Generic
Memory breakpoints, GDB	Rarely	Somewhat helpful	Generic
Inspecting memory, GDB	Rarely	Somewhat helpful	Generic
Inspecting registers, GDB	Sometimes	Somewhat helpful	Generic
Inspecting symbol values, GDB	Rarely	Questionably helpful	Generic
Own functions (in code) for debugging	Sometimes	Somewhat helpful	Specialized
Own debugging programs	Often	Very helpful	Generic
Reading documentation	Rarely	Very helpful	Specialized
Web search (google, forums, ...)	Sometimes	Very helpful	Generic
Questioning community	Rarely	Somewhat helpful	Generic
Tools for static analysis	Rarely	Very helpful	Generic

Table 10: Evaluation of debugging tool usefulness

One interesting finding is that the GUI debugger was used mainly for its ability to put breakpoints and do stepping, view disassembled code and put memory breakpoints. More interestingly, even such a common feature as viewing the call stack was never reported for the GUI debugger. We see some possible explanations. The first is that the participants omitted to record usage of the call stack view. The second explanation is that the task of implementing the core of an operating system really generates very few situations where the call stack view in the GUI debugger would be useful. Or users of a GUI debugger need the call stack much less often than users of GDB, because they are much more often aware about the current location of the program execution.

The second thing worth of mentioning here is that the amount of data for evaluation which tools have a specialized usage is low in our opinion. And because some intents are performed much more often (see the table 18) than other intents, many intents has only from one to a few records. Therefore, tools that were recorded for the intents with low amount of records easily satisfy our definition, and thus we consider the validity of the specialized column at high risk and provide the summary mostly just for orientation purposes.

4.1.2. Comparision of a GUI debugger, GDB and printing messages

Question

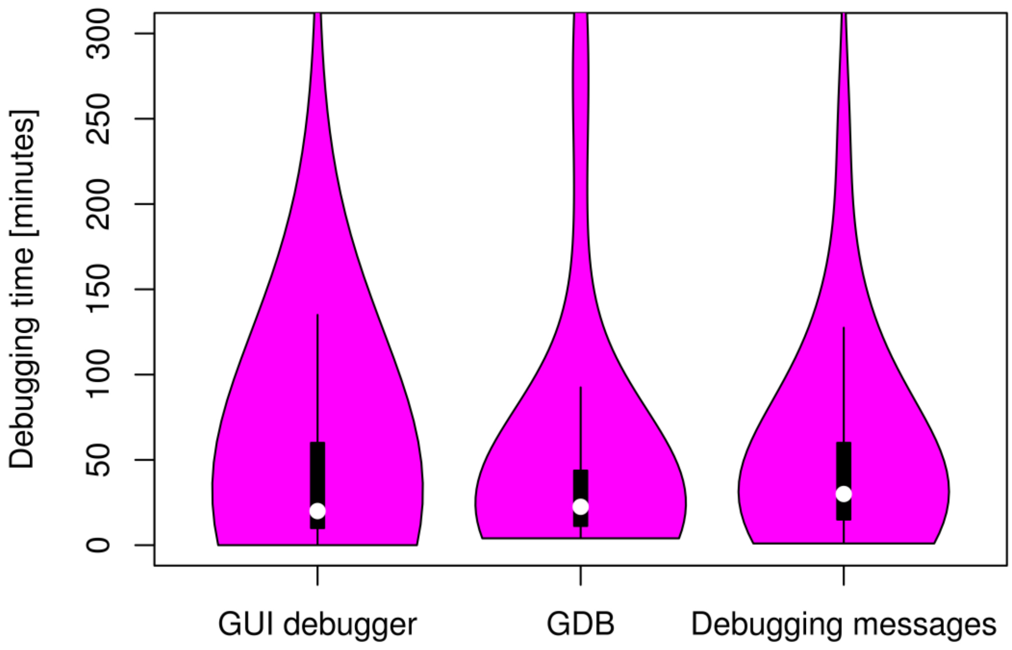
Sometimes low-level programmers have the possibility to use a graphical debugger, a command-line debugger or they can debug by printing debug messages. Which way is the fastest? This question relates to the hypothesis 1 of the section 3.6.6.

Data analysis

We took debugging issues that were investigated only by either a graphical debugger, by the GDB command-line debugger or by printing debug messages. During the data checks we discovered that some participants had a strong preference of a single debugging tool. Therefore, we took into account only a limited number of records from those participants in order to normalize the influence of their personal debugging style. The comparision is presented in the table 11 and the picture 14.

Debugging time [min]	GUI debugger 7 participants 42 records	GDB 8 participants 24 records	Debugging messages 13 participants 57 records
Minimum	0	4	1
1st Quartile	10	15	15
Median	20	25	30
Mean	72.3	60.8	68.8
3rd Quartile	60	45	60
Maximum	1200	480	600

Table 11: Debugging time statistics for comparing how fast is debugging with GUI debugging, GDB debugger or just debugging prints



Picture 14: Distributions of debugging time for comparing how fast is debugging with GUI debugging, GDB debugger or just debugging prints

Testing whether the means differ statistically:

```
> t.test(GUIOnlyDebuggedTimes, GDBOnlyDebuggedTimes)
p-value = 0.66

> t.test(GUIOnlyDebuggedTimes, PrintingMessagesOnlyDebuggedTimes)
p-value = 0.90

> t.test(GDBOnlyDebuggedTimes, PrintingMessagesOnlyDebuggedTimes)
p-value = 0.71
```

The difference in means of investigation time with using the GUI debugger, GDB or just debugging messages is not statistically significant.

Answer

Based on our data we see no major difference of how the choice between a graphical debugger, the GDB command-line debugger or debugging messages affects time of debugging.

The only minor observation is that the graphical debugger is more suitable to investigate issues that are fast for resolution (approximately up to 20 minutes). On the other hand the data indicates that using a command-line debugger is better for issues that take a lot of time for resolution (approximately from 45 minutes).

Hypotheses

Although researching this trend more deeply is outside the scope of this study, we can at least formulate a hypothesis that explains this trend: Using a graphical debugger is more comfortable (meaning that the users see the whole source code and are able to navigate easily and quickly) than the other debugging means, which helps programmers to investigate simple debugging scenarios faster than with the other means. On the other hand the programmers tend to think less intensively while using a graphical debugger, so resolving difficult debugging scenarios takes them more time than with using other means.

4.2. Further aspects on debugging

In this section we present data about various other aspects of debugging that we were able to collect. In order to make these results as much practically oriented as possible, we begin with a question that addresses real-world issues and provide more context of the question. Furthermore, we maintain objectivity by separation of facts from our interpretation, opinions and beliefs. Also note that among many evaluated views on the data we present only those that we consider valuable (see 3.6.2 for our criteria).

4.2.1. What portion of development is spent by debugging

Question

How much time should project managers expect to be spent on debugging during development?

Context

Project managers in many industries use the technique of Gantt diagram (7) and the method of critical path for planning of the work for their colleagues. In software engineering this technique has a very weak point - it is very hard to estimate working time for single programming tasks and working packages. For example, throughout our professional experience it has been common to provide estimations that were two or three times less than the actual amount of performed work. We can remember even some tasks that were originally estimated to a week of work and ended up after three months of efforts. These errors in estimates lead to bad project planning and the consequences are often stressful for everybody involved.

Data analysis

We take into consideration the amount of time that each participant was investigating the unexpected behavior and the total time that he spent on the project. Then, we compute what part in percents did he spend by debugging and we are interested in the mean value of these percents.

95% one-sided confidence interval of the mean value is [0%; 40.0%].

Answer

Based on the data we have we can help project managers if they were able to obtain an estimate of other development efforts without debugging activities, because our data suggest that debugging takes less than 40% of development time in average.

From our results we propose an improvement of the depicted planning approach: Extend the estimation of work without debugging activities by the upper bound of the

95% confidence interval (i.e. 40% in our case) and you will get an estimation for the whole programming activity including the debugging efforts. We believe that this advice will, in average, give better estimates than those based on pure intuition. Longer chains of activities (more than 4) will in our opinion reduce deviations from the average. If you will experiment with this proposal please check the following:

- Be aware that the activity you are applying this suggestion to should be similar to the activities we measured. It should be some kind of coding in low-level programming language, ideally development of an operating system.
- Take better care on the critical path and critical sub paths of the Gantt diagram, because an error in estimation on activities on these paths may have worse consequences.
- Collect data from your environment to refine the precision of the estimates.
- Do provide us feedback how our proposal worked.

4.2.2. Worst-case estimation of debugging time for a single issue

Question

This question and the next question 4.2.3 focus on estimation of investigation time for a single debugging activity. How much time can I expect to debug an issue that is likely to be very hard to analyze?

Context

The customer reported a problem in our operating system and the initial analysis and symptoms suggest that the issue will be very hard for debugging. The customer needs to understand urgently where the problem is located and your boss is expected to give him some realistic worst-case time of getting the problem investigated. In what time can I tell my boss (with 95% probability) that the root cause will be found?

Data analysis

```
> complexity = read.csv("complexity-and-feelings.csv", header = TRUE)
> veryHardDifficultyTimes <- complexity[complexity[,2] == 4,1]
> quantile(veryHardDifficultyTimes / 60, 0.95)
```

The 95% quantile is 12.6 hours of investigation. Searching for the root cause was abandoned in 2.1% of cases.

Answer

You can tell your boss that the root cause will be investigated with reasonable certainty after 12.6 work hours. Only one issue of twenty will take longer. Also note that some small amount of issues (2.1%) was left unresolved, so the full investigation would take longer in those cases.

We consider this estimation suited best for tasks related to development of an operating system where simulation on QEMU is possible.

4.2.3. Most probable estimation of debugging time for a single issue

Given my feelings of the bug difficulty, what time can I expect to be debugging the issue?

Context

I ask, because many times when it is late in the day my wife calls me asking at what time I will be home. She will prepare the dinner to be hot at that time. The only thing I can base my estimation on is a more or less vague feeling from bug's symptoms and the environment (multi-threaded scenario, assembler code, etc.) where I will search for the bug. Very often my estimation is very wrong, the dinner gets cold and my wife is somewhat disappointed when I return home. On the other hand, I have a strong need to have the task finished when I leave the work. It would be very helpful if I could estimate the debugging time much better.

Data analysis

Similarly to the previous question, we take some relevant quantiles of investigation time just for bugs with easy difficulty. The table 12 presents the distribution.

```
> easyDifficultyTimes <- complexity[complexity[,2] == 2,1]
> quantile(easyDifficultyTimes , c(0.25,0.5,0.75,0.95))
```

Quantile	Debugging time of easy bugs [min]
25%	9
50%	20
75%	31
95%	120

Table 12: Investigation time for bugs that were perceived as easy to investigate

Data for other difficulties have similarly wide or wider distribution.

Answer

Suppose that the issue does not seem trivial, but it still seems simple. From the data we can say that there is a 25% probability that the issue will be analyzed in 9 minutes, a 25% probability that it will take from 9 minutes to 20 minutes, a 25% that it will take from 20 minutes to 31 minutes, a 20% probability that it will take from 31 minutes to 120 minutes, and a 5% probability that you may not be able to finish the task today, so you will return very late. Furthermore, this data cover only the investigation of the problem and not the fixing efforts.

Suppose that the lunch will get cold (and your wife annoyed) in twenty minutes. Then, it is clear that you cannot give a precise estimation with reasonable certainty, because there will be still at least 50% chance that your estimation will be wrong regardless of what it will be. The recommended practice for these situations is an uncompromising search for an alternative that fulfils both needs. A special course or training of creative thinking and communicating people's needs can be very helpful in such times.

4.2.4. Debugging time for design and implementation errors

Question

Is there any statistically significant difference between time that is required for investigating flaws in design and the time that is required to investigate errors in implementation? This question relates to the hypothesis 2 from the section 3.6.6.

Data analysis

We compare the mean values of debugging time for bugs caused by wrong design and by wrong implementation.

```
> designErrorsTime <- records[records["CategoryId"] == 1, "Debugging_Time"]
> implementErrorsTime <- records[records["CategoryId"] == 2, "Debugging_Time"]
> t.test(designErrorsTime, implementErrorsTime)

p-value = 0.02239
95 percent confidence interval:
<3.574012; 46.632950>
mean of designErrorsTime 85.04206
mean of implementErrorsTime 59.93857
```

Answer

The difference is statistically significant. Flaws or bugs in design of program logic are more time-consuming for investigation than bugs in correctly thought, but wrongly implemented ideas. On the other hand, our data do not suggest that the difference is larger than twice the lesser value.

4.2.5. Debugging time of different programmers

Question

What are the differences between how long individual programmers perform debugging activities across the whole project?

Data analysis

The table 13 summarizes the debugging efforts of each participant and how many issues did he encounter.

Participant	Development time [h]	Debugging time [h]	Development time / debugging time [%]	Investigation count
12[6]	66	47	71.3	95
15	110	9	8.3	18
16	144	59	41.0	18
17	96	35	36.7	18
19	42	18	41.7	9
20	72	8	11.5	6
21	61	25	41.3	23
22	125	45	35.9	50
23	180	87	48.5	64
48	351	167	47.4	49
53	323	102	31.6	109
54	136	40	29.3	14
58	155	27	17.7	78
59	209	55	26.1	48
60	205	84	40.9	63

Table 13: Debugging efforts of individual participants

The average ratio of debugging time is 35.2% with standard deviation 15.8%.

Participants 15, 19 and 20 had a surprisingly low debugging time or the count of reported issues. Based on their feedback we assume two reasons for that:

- They focused much more on documentation than the other team members.

[6] From the feedback of participant 12, we are concerned that the high amount of his debugging reports had significant negative impact on his programming performance, because he filled a somewhat large amount of data in shorter period of time than the others. This is a weak point of our methodology as discussed in 3.6.4.

- Because of some limitations, they were forced to develop larger parts of software without executing them at all. Therefore, the majority of their debugging efforts were done in the integration phase, which was in their case only one or two days long.

Answer

Developers spend from about 10% and up to 45% of their time by debugging activities. One participant spent more than 70% of his work time by debugging and he abandoned the project for that year.

Note that we do not consider this percentage to be a good indicator of individual programmer's performance, because it does not tell anything about the development conditions and the kinds of task of the individual programmer.

4.2.6. Debugging and work satisfaction

Question

How is the debugging activity related to the mood and work satisfaction of the professionals? From our professional experience we know a method[7] how a human-resources person can relatively quickly evaluate the work satisfaction of employees. It works by the following way:

Write down how many hours per week you spend in your job by activities that are exciting or very pleasant, pleasant, neutral or OK, unpleasant, and very unpleasant. The methodology suggests that a highly satisfying job is the one that has more than 60% of exciting, very pleasant or pleasant activities.

Context

This problem interests us because of the fact that many people are discouraged by pursuing technical carrier is, in our opinion, at least partially caused by characteristics of those professions. Technically demanding professions often require intensive problem-solving activity that is related to machines, dysfunctional software or other problems that does not in its core incorporate human relationships and communication. Typically, even highly skilled technicians are confronted with situations that something technical does not work as expected and they try to fix it for many hours. From our life experience, this situations cause large amount of frustration regardless whether the profession is a builder, a car mechanic, a wireman, an IT support, an electrotechnician, or a programmer. One typical reaction what people do in these situations is that they throw various curses around themselves, because probably, it helps them to make their frustration easier. Anyway, we believe that the ability to handle this kind of frustration is one of determining factor whether a person will be a skilful technician.

Therefore, we think that investigating topics related to this question could provide very valuable insights for educators of computer science or technical professions in general.

[7] There is large amount of research and approaches related to the job satisfaction of people, see (20) for a summary. We were not able to find the exact origin of the described method, but in our opinion it seems to be a derivative of the Minessota Satisfaction Questonaire. In our history we heard feedback of several people whose job satisfaction was evaluated by this method. They appreciated that it helped them to reflect what activities are unpleasant or stressful for them and it provided them some guidance on how they ideal workload should look like.

Data analysis

The table 14 contains the summary of our data. The numbers in the feelings columns are computed as sum of debugging time of issues with the particular feelings divided by the whole debugging time of the participant. Because the debugging is assumed to be the most flustrating activity of this project, we focused on the periods of unpleasant or very unpleasant debugging. Therefore, the last column contains what part of the whole project the particular user had been debugging unpleasant or very unpleasant issues.

Participant	Feelings [% of debugging time]				Contribution to negative work satisfaction on the project [%]
	Positive	Nothing remarkable	Unpleasant	Very unpleasant	
12	1.2	17.6	33.5	47.6	57.8
15	36.2	50.1	13.7	0.0	1.1
16	9.3	17.8	18.6	54.2	29.9
17	21.3	40.5	29.8	8.5	14.1
19	74.3	22.9	2.9	0.0	1.2
20	0.0	69.7	30.3	0.0	3.5
21	1.0	21.6	33.8	43.7	32.0
22	5.8	66.4	27.9	0.0	10.0
23	8.4	53.2	11.7	26.7	18.6
48	24.8	6.5	24.4	44.2	32.6
53	1.9	25.6	17.2	55.3	22.9
54	24.2	17.9	22.7	35.3	17.0
58	1.3	60.1	37.2	1.5	6.8
59	0.1	35.1	18.2	46.7	16.9
60	5.2	21.1	41.6	32.2	30.2
All	12.1	28.0	24.1	35.7	21.1

Table 14: Percentage of debugging time grouped by feelings

Answer

Our methodology was not aimed to collect data about feelings of participants or mood across the whole project, so we can at least interpret the data partially from the debugging part. In our opinion, participants 12[8], 16, 21, 48 and 60 could welcome some changes in their work activities, because the amount of negative issues they experienced was getting over 30% of their work time.

4.2.7. Debugging time and perceived complexity

Question

How does the perceived difficulty of bugs correspond to the debugging time?

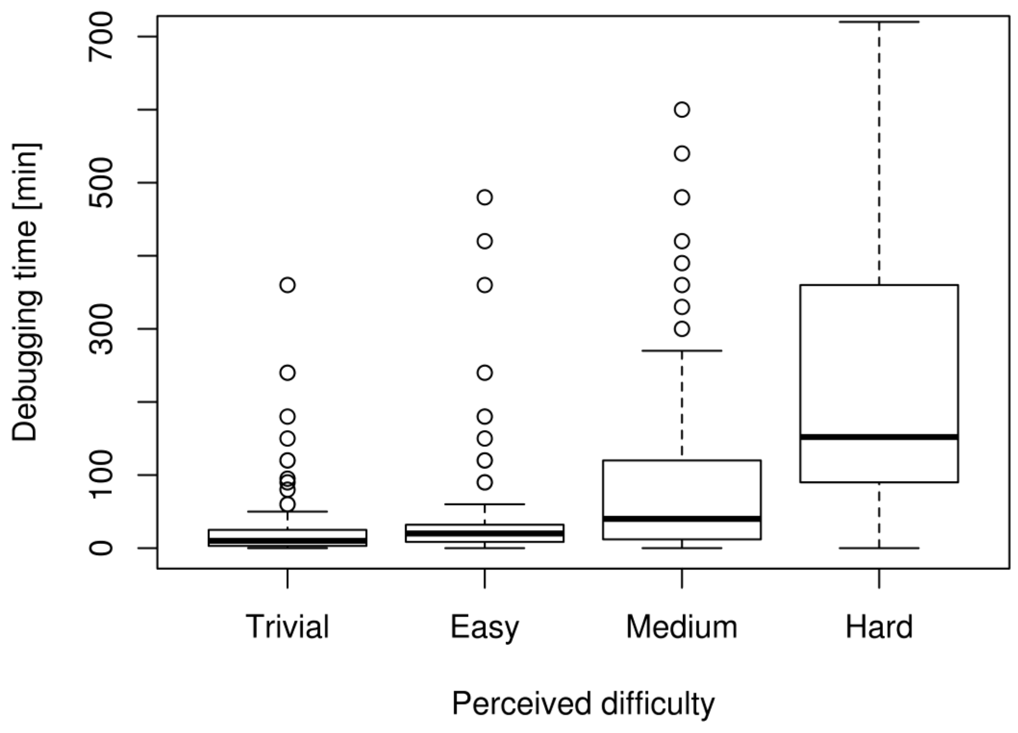
Context

This could give evidence on how is the perception of debugging issue difficulty connected with debugging time, which could possibly improve time estimations.

[8] The participant 12 did not finish the tasks that year.

Data analysis

The picture 15 shows how is the debugging time related to the perceived difficulty and the table 15 summarizes the median values.



Picture 15: Debugging time grouped by perceived difficulty

Difficulty	Median of debugging time [min]
Trivial	10
Easy	20
Medium	40
Hard	152

Table 15: Medians of debugging time for issues with different perceived difficulty

Answer

Our data suggest that programmers perceive in such a way that with each increase of difficulty level the debugging time tends to increase two or three times.

Note that our observations do not tell anything about causality because of our methods of data collection. More specifically, we cannot claim that a more complex debugging scenario implies more debugging time, or that more debugging time implies that the programmers perceive the debugging scenario as more difficult. We just provided evidence about the relationship of these two variables.

4.2.8. Debugging time in different life cycles of a bug

Question

Many people believe that it is much more costly to fix a bug at the end of the project than fixing the same bug during the early stage of the project. How does our data support that claim?

Context

Many professionals believe that this claim is correct in reality and even our professional experience supports it. On the other hand we are aware of only one publication (8) about its validity:

"A significant related insight is that the cost of fixing or reworking software is much smaller (by factor of 50 to 200) in the earlier phases of the software life cycle than in the later phases."

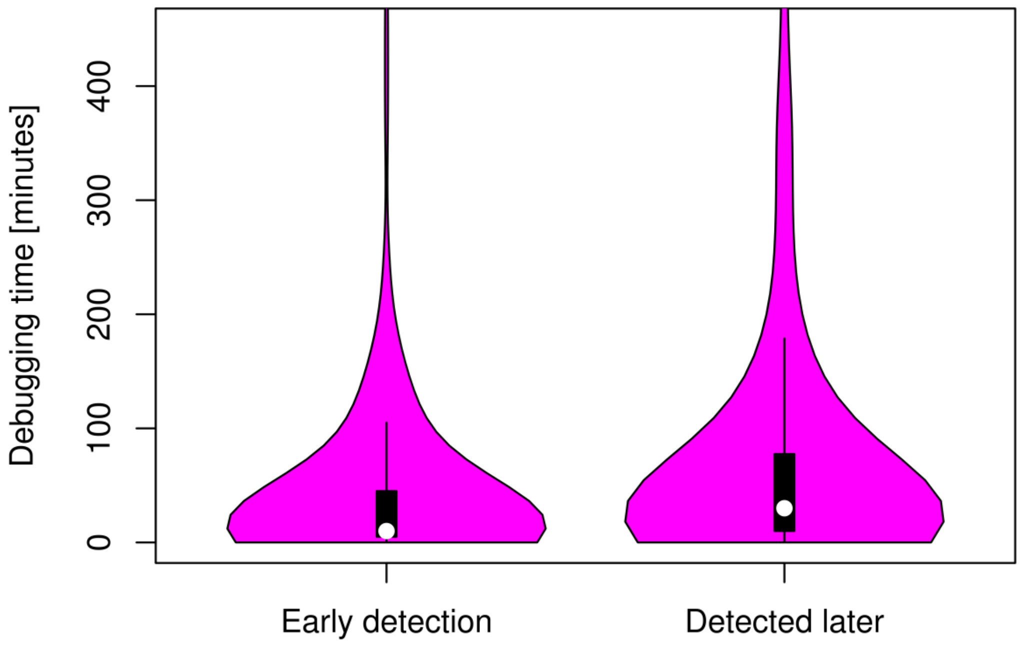
This evidence is over two decades old, so one may not see as very strong, because it may be outdated. Therefore, we perceive publishing further newer data as a valuable contribution.

Data analysis

We compared reports of bugs that were investigated during the first checks of the just implemented code and reports of issues that were detected later. In the picture 16 and table 16 the first kind of issues is referred as issues that were detected early and the latter kind is referred as issues that were detected later. The data are composed from 218 reports of the early detected issues and 444 issues that were detected later.

Debugging time [min]	Early detection	Detected later
Minimum	0	0
1st Quartile	5	10
Median	10	30
Mean	47.2	78.3
3rd Quartile	45	77.5
Maximum	1200	1800

Table 16: Debugging time of issues from different bug life cycles



Picture 16: Debugging time of issues from different bug life cycles

```
>t.test(early[,2], later[,2])
p-value = 0.0059
```

The difference of mean debugging time is statistically significant.

Answer

We confirm that bugs that are detected very soon during the development are resolved approximately twice faster that bugs that are detected later in the project development.

Hypotheses

The scope of our study allowed us only to compare only issues that were raised before any actual usage of the developed software. Therefore, we expect the difference to be even larger for bugs that would be detected after the release of the developed software.

4.2.9. Debugging time of bugs related to assembler

Question

It quite intuitive that bugs that are related to code that is written in assembler takes much more time for debugging than other bugs. Does the collected data provide statistically significant evidence about this claim? This question is related to the hypothesis 3 of the section 3.6.6.

Data analysis

We compare the mean values of debugging time for bugs that were related to assembler with other bugs.

```
> t.test(assemblerDebuggingTimes, nonassemblerDebuggingTimes)
p-value = 0.059
mean of assemblerDebuggingTimes = 177 minutes
mean of nonassemblerDebuggingTimes = 67 minutes
```

Answer

The difference in mean of debugging time between bugs that are related to assembler and the bugs that are not related to assembler is not statistically significant.

Hypotheses

We interpret this result as that there exist other kinds of bugs that are not related to the assembler language and they still increase debugging time similarly as the assembler related ones.

4.2.10. Debugging time and other aspects

Question

During the study the participants indicated for each investigated issue whether the issue was related to assembler, cause by a copy-and-paste activity, located in foreign code, caused by incomplete modification (better explained in the attachment 6), related to the C preprocessor, debugged by more people, and related to a memory corruption. How frequent are these cases and what is their impact on debugging time?

Data analysis

We constructed a linear model of how the named aspects affected debugging time.

```
> model <- lm(Debugging_Time ~ Flag_Assembler_Related +
Flag_Caused_By_Copy_And_Paste + Flag_In_Foreign_Code +
Flag_Incomplete_Modification + Flag_Preprocessor_Related +
Flag_Debugged_By_More_People + Flag_Memory_Corruption_Related, data=dat)
```

The table 17 summarizes the model coefficients, their relevance, and contains the frequency of the issues with the aspect.

Aspect	Coefficient [min]	p value	Frequency [%]
Assembler related	108.6	9.5E-006	5.1
Caused by copy and paste	55.6	0.017	5.7
In foreign code	90.3	6.4E-007	10.3
Incomplete modification		0.434	13.0
Preprocessor related		0.475	1.1
Debugged by more people	66.7	3.1E-005	14.5
Memory corruption related	63.4	4.8E-005	14.4

Table 17: Linear model for different aspects of bugs

Answer

The only observed aspects of investigated issues that do not affect investigation time are bugs caused by an incomplete modification or by the C preprocessor. The other observed aspects increase the investigation time significantly. The specific effect of increase can be seen in the table 17 in the coefficient column.

Hypotheses

As for the frequencies, we believe that assembler and memory corruption related issues will be less common during development of higher-level software than an operating system. The frequencies of *debugged by more people* and *bug located in the foreign code* can change with different structure of programming teams. Note that our data was taken from teams of two or three programmers.

4.2.11. Debugging time and copy-and paste bugs

Question

Some people claim that copying chunks of code without proper modifications of the copied code is one of the most common sources of programming errors, and that programmers do copy-and-pasting in order to save their time, but they lose more time by creating more bugs. How correct is this in reality?

Data analysis

5.7% of reported bugs were caused by the copy-and-paste activity. The linear model from table 17 suggests that a copy-and-paste bug adds 55 minutes to the debugging time.

Answer

The data shows much less frequency of this kind of bugs than we expected. We can say that the effect of each copy-and-paste bug can be thought as a 55 minutes increase in debugging time in average. Also note that for the full cost and benefit analysis of copy-and-pasting we believe that other factors should be taken into account: copy-and-paste operations may save time, improve the work-satisfaction and reduce mental exhaustion of the developers. Monitoring these other factors is out of the scope of this work.

Hypotheses

We believe that the low frequency of copy-and-paste errors is caused by two factors:

- Participants were very often experienced and skilled programmers. They are aware of this kind of bugs and pay special attention to prevent errors when they copy code, or they use programming practices that avoid copy and paste activity.
- The fact that participants explicitly filled the information that the bug was caused by code copying trained them to avoid these errors.

4.3. More theoretical aspects of debugging

In this section we provide data about aspects of debugging that are not aimed to address real-world issues so much as those in the section 4.2. We present these data in order to support further research efforts.

4.3.1. Debugging intents and their frequency

Question

Naturally, when the programmers use a debugging tool they do that with some intent. What are the intents for using debugging tools and what is their frequency? This information could be beneficial for designers of debugging tools, because it maps common debugging intents and tells how often they are.

Data analysis

In the table 18 we summarize all the identified debugging intents and their frequencies. This summary is backed by 62 collected records.

Debugging intent	Relative frequency [%]
Investigate value of variable	11.3
Investigate whether some code was executed	0.0
Investigate function return value	4.8
Check that some code is reached	6.5
Investigate what has changed recently	1.6
Attempt to reproduce the problem	3.2
Investigate what is the program doing right now	1.6
Investigate CPU registers	6.5
Execute just a subset of the code	0.0
Search of error code meaning	3.2
Investigate execution flow in the code	25.8
Search for a symbol address	0.0
Check whether the code matches to the compiled instructions	4.8
Investigate the consistency of the memory	6.5
Search for the place where a variable is changed	0.0
Check statistical parameters of a (pseudo) random events	0.0
Search for an error in the stdout	0.0
Use the tools of formal verifications	0.0
Investigate TLB mapping	9.7
Search for a symbol name from address	1.6
Search for a place where the program crashed (NULL pointer dereference, ...)	11.3
Execute tests to check whether they still pass	1.6

Table 18: Identified debugging intents and their frequencies

Answer

Unfortunately, the period of collecting data about debugging intents was not long enough to collect many reports. Therefore, we consider the presented summary to be most suitable just for orientation purposes. Furthermore, the *Investigate TLB mapping* debugging intent is highly specific just for our programming task, so for more typical programming tasks we are confident that this debugging intent would be very rare or even less common.

4.3.2. Root causes and bug frequency

Question

What kinds of root causes occur the most often? In summary, what kinds of root causes are investigated for the longest period of time?

Data analysis

The table 19 contains frequencies and sum of debugging time of each root cause identified by our methods in 3.3.3.6.

Root cause category	Count of bugs	Relative count of bugs [%]	Sum of debugging time of the bugs [min]	Relative portion of debugging time against the whole [%]
Wrong design assumption	131	19.8	11240	23.2
Forgotten code	116	17.5	5571	11.5
Used wrong entity	76	11.5	4021	8.3
Wrong expression	44	6.6	1876	3.9
Memory	43	6.5	3186	6.6
Synchronization	43	6.5	3955	8.2
All other	41	6.2	5862	12.1
Data structures and algorithms	40	6.0	3004	6.2
Wrong program flow	27	4.1	942	1.9
Wrong (coding related) assumption	23	3.5	1320	2.7
Initialization	22	3.3	1678	3.5
Extra code	15	2.3	430	0.9
Subprogram binding	11	1.7	567	1.2
Assembler specific	10	1.5	3005	6.2
C specific	7	1.1	469	1.0
Dynamic data structures	5	0.8	270	0.6
Other	3	0.5	260	0.5
Value corruption	3	0.5	290	0.6
Finalization	2	0.3	510	1.1

Table 19: Frequencies of different root causes and associated debugging time

Correlation coefficient between root cause frequency and the summed debugging time for those root causes is 0.88.

Answer

The most common root causes are from flaws in design or thinking (19.8%), pieces of code that were forgotten to be written (17.5%) and usage of wrong entity (11.5%). Regarding the amount of time spent on debugging each kind of bug we see that more frequent root causes tend to contribute more to the debugging time. The largest exception to this trend are *All other* root causes with the dominating root cause *Not a bug*, which is not so common, but it takes unproportionally long debugging time.

4.3.3. Root causes and debugging time

Question

Where is a very meaningful place for researchers to concentrate their efforts if they want to make debugging faster?

Context

The difference between this question and the question 4.3.2 is that it is concerned on the direct cost of debugging (developer's time), which may be a more relevant aspect in fault-tolerant environment where other consequences of software bugs are not very unpleasant.

Data analysis

We refer to the table 19. The *Wrong design assumption* root cause was behind 23.2% of debugging activities, the *All other* root cause was behind 12.1% and the *Forgotten code* was behind 11.5% of debugging activities.

Answer

The data suggest that one very meaningful place for improvement is helping programmers with these kinds of situations:

- Assisting them with realizing all the corner cases, which should help with the *Wrong design assumption* root causes.
- Making sure that they understand correctly how the external entities of the software (such as hardware or a third party library) are supposed to be used, which should again help with the *Wrong design assumption* root causes.
- Assisting them to avoid overlooking or forgetting parts of implementation, which should help them with the *Forgotten code* root causes. Unfortunately, during this study we did not obtain data that show how the parts of implementation got missing. These data could provide a more specific guidance and obtaining them can be a further direction of this research.

Hypotheses

This study was not designed to observe the participants so closely to map the reasons why debugging activities caused by these three root causes were so time-consuming, so we at least provide our hypothesis:

The reason why the *Wrong design assumptions* and *All other* kinds of bugs took 35.3% of all the debugging activities seems to lay in the fact that the bugs both happen often and each one is often very time-consuming by itself. We guess that the most consuming part of debugging these bugs is realizing how things are supposed to work correctly. The reason why the *Forgotten code* bugs took 11.5% of all the debugging activities seems to be just because they had been occurring very often.

4.3.4. Root causes and the project phase of their detection

Which kinds of bugs have tendencies to remain in the code longer?

Context

Many bugs in commercial software are unpleasantly costly, so the programmers may strongly desire to have as few bugs as possible in the released software. Answering this question could provide evidence if some bug root causes tend more to remain undetected.

Data analysis

The problem with measuring the count of undetected bugs is that they cannot be detected and so they cannot be measured directly. Therefore, we work under assumption that bugs that are detected but present in the code for a long time have similar properties as undetected bugs. Thus, for the context of 4.3.4 we specify what we consider a bug that is present in the code for a long period of time for the context of this study.

Most typically, programmers at least check their software right after they implement a runnable and checkable part of it. We call bugs that are detected in this phase as bugs detected early in the development. What happens next with the quality assurance efforts varies on the kind of organization and project management. In the case of this study, the students used a testsuite and their work was accepted when their code passed the testsuite. We call bugs caught by the testsuite as detected by tests. As the students are supposed to build on their previous code for longer time (mostly more than 100 work hours, see the table 13 for exact numbers), they could detect further bugs. We call these bugs as bugs detected in the later stage of development.

Note that in enterprise environment one could expect additional possible stages when a bug can be detected – for example during quality assurance processes or the actual usage of the software. Therefore, for obtaining more realistic data from other environment our methods would have to be adjusted or reworked.

The table 20 shows the occurrences of various kinds of bugs during early, testing and later development stages.

Root cause category	Detected early	Detected by tests	Detected later	Detected later relatively [%]
Initialization	12	1	9	41
Subprogram binding	3	3	5	45
Data structures and algorithms	12	9	19	48
Wrong design assumption	36	23	72	55
Finalization	2	0	0	0
Used wrong entity	37	10	29	38
Other	2	0	1	33
Forgotten code	37	15	64	55
Dynamic data structures	2	2	1	20
Memory	14	5	24	56
Wrong expression	15	5	24	55
Synchronization	10	6	27	63
C specific	3	0	4	57
Assembler specific	4	1	5	50
Value corruption	0	2	1	33
Wrong program flow	10	5	12	44
Extra code	6	1	8	53
Wrong (coding related) assumption	6	3	14	61
Sum	212(34%)	91(14.6%)	319(51.4%)	

Table 20: Count of bugs grouped by the stage of their detection

Answer

The relative occurrence of bugs that survive in the code the early development phase ranges between 40% and 65% (with few exceptions below 40%). Therefore, we see no very strong connection on how the kind of root cause affects the bug's ability to remain undetected. The most undetectable bugs seem to be those caused by errors in synchronization or by assumptions that some code constructs work in a different manner.

The other interesting finding is that the correctness checks that were done soon after the implementation together with tests were able to detect less than 50% of the bugs. To be more confident about validity of this claim, we would need a more controlled experiment, because we cannot guarantee that no bugs from the early detection phase were omitted.

4.3.5. Debugging with and without debugging tools

Question

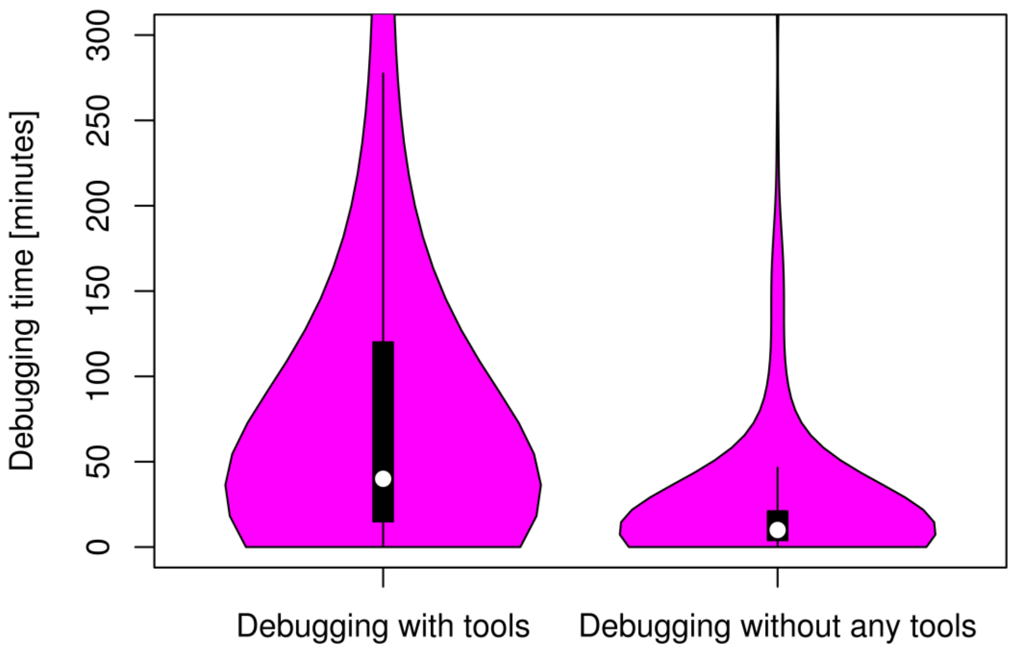
What are the specifics of bugs that programmers investigate without debugging tools? How often does that happen?

Context

Debugging without any debugging tools may indicate a lack of coverage of available debugging tools. This information can provide some guidance where to focus research effort to address inconvenient issues.

Data analysis

In picture 17 and table 21 we compare 253 bug reports that were investigated without any debugging tools to 409 bug reports that were investigated with one or more debugging tools.



Picture 17: How is a usage of a debugging tool connected with debugging time

Debugging time [min]	Debugging with tools	Debugging without tools
Minimum	0	0
1st Quartile	15	4
Median	40	10
Mean	100	29.8
3rd Quartile	120	21
Maximum	1800	720

Table 21: How is a usage of a debugging tool connected with debugging time

Here we compare the mean values:

```
> t.test(debugging_time_with_tools, debugging_time_without_tools)
p-value = 8.0e-13
```

The difference in mean values is statistically significant.

Answer

The collected data shows a rather surprising result that bugs that were debugged without any debugging tools were investigated much faster than those investigated with debugging tools support. When the programmers used a debugging tool they had been debugging in average about 4 times longer than in the cases when they did not use any debugging tool.

Hypotheses

This observation can have a number of possible explanations. One of them is that programmers tend to think much more intensively when they do not have any programming means available (which happens sometimes during development of an operating system), and therefore they debug much faster. In our study we do not have data to explore this explanation.

Another explanation would be that we have a flaw in our methods, because the amount of recorded data could be so large that sometimes the participants could skip parts of the bug report. We think that a specialized more controlled experiment is needed to check this possibility.

The next questions investigate some other possible explanations of our observation.

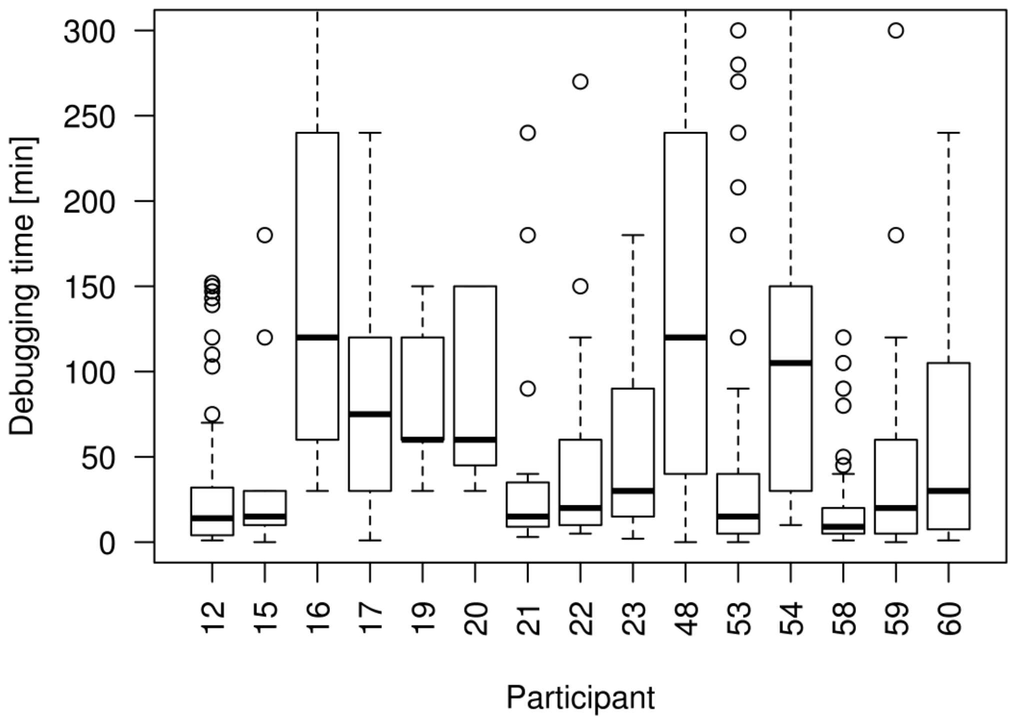
4.3.6. Participants preference on using debugging tools

Question

The finding of the previous question 4.3.5 could be explained by the fact that participants strongly preferred and used only one way of debugging and the fact that debugging time differs significantly for different programmers (as presented in table 13). Did the best performers (i.e. in this context participants who debugged the issues in a very fast way) debug without any debugging tools?

Data analysis

The picture 18 shows how fast each participant debugged his bugs.



Picture 18: Debugging times of each participant

In table 22 we summarize how often each participant used debugging tools.

Participant	Reports with a tool [%]	Reports without any tool [%]	Reports count
12	24	76	95
15	44	56	18
16	17	83	18
17	78	22	18
19	89	11	9
20	50	50	6
21	91	9	23
22	96	4	50
23	83	17	64
48	86	14	49
53	37	63	109
54	100	0	14
58	78	22	78
59	65	35	48
60	63	37	63

Table 22: How often each participant used debugging tools

The fastest participants were 12, 15, 21, 53 and 58. The sum of their records where no debugging tools were used is 170. That is 67% of all these records.

Answer

One third of the fastest participants reported two thirds of the investigations that belong to the group of in average 4 times faster investigations than the other reports. Therefore, our data suggest that the fast debugging programmers tend not to use debugging tools.

On the other hand, we are not strongly convinced about this conclusion, because our observation could be caused by the fact that some participants omitted reports of very quickly investigated issues. We would need a controlled experiment to become more confident on how to interpret the finding of this question.

4.3.7. Usage of debugging tools and bug complexity

Question

Another explanation of finding in the question 4.3.5 is that more difficult errors are more often investigated with debugging support. Thus, this could explain the difference of the debugging times distributions in picture 17, because harder bugs take much more time to investigate. Does the data confirm that?

Data analysis

The table 23 shows how many errors were investigated with and without debugging tools for each complexity.

Complexity	With debugging tools	Without any debugging tool
Trivial	82	99
Easy	132	92
Medium	131	51
Hard	64	11

Table 23: How perceived difficulty of bugs is related to whether a debugging tool was used

Answer

We see that in difficult debugging scenarios people tend to use some debugging support much more often. From data of 4.2.7, we know that debugging an issue with easy perceived complexity tends to be two times faster than debugging an issue with medium difficulty and that tends to be three times faster than debugging an issue with hard difficulty. Therefore, we conclude that the questioned explanation is at least one of the factors why issues debugged without debugging tools are investigated faster.

4.3.8. Selection from available debugging tools

Question

The common sense tells that programmers use the debugging tool that is the most useful for the particular debugging scenario. How much is the common sense the common practice?

Context

Answering this question could provide valuable input for researchers that would like to make a similar study as ours, or for innovators that would like to introduce new tools and practices into their teams or companies.

Data analysis

We made five interviews with the participants and asked the participants how their debugging preferences evolved over time. All of them used debugging tools that had been most usual for them before the project. One participant openly admitted that he did not even try out other debugging means despite the fact that he was encouraged to do so.

Answer

From the feedback of participants and our professional experience we doubt the expectation that people will automatically use the best available tool. Often people tend to use what worked best for them in the history and are reluctant to new things. We consider this observation interesting, because the participants were, in our opinion, very adaptable professionals in the information technology world, which is generally considered highly innovative and very open to changes.

5. Related work

We have structured the related work for our thesis into three types, which we will describe in the following three subsections.

5.1. Studies evaluating usefulness of debugging techniques and tools

This is the most relevant kind of related work as it provides experience of how to do evaluations of debugging tools, and it stresses out the importance of these studies. (6) is an evaluation of techniques of automated debugging, which is being researched for decades. It reviewed the usefulness evaluation activities that had been published and concludes that:

"... and most programs studied are trivial, often involving less than 100 lines of code."

Therefore, (6) evaluated the usefulness of automated debugging techniques with programmers and on two programs of 2403 and 4408 LOC, and it came with a finding that the evaluated techniques helped only to expert programmers. Furthermore, it identified the reasons for the unexpectedly low usefulness in neglecting human-related aspects during design of the techniques. For example, it was observed that programmers value more explanations than recommendations and the techniques are based on recommendations. Interestingly, we believe that the same threat to validity can be applied for (6) as well, because the evaluated techniques may be much more helpful for very large programs with 1 million or more LOC than they are for programs of several thousand LOC. Anyway, (6) supports our opinion that performing more usefulness evaluations of researched debugging tools or approaches is likely to make current research efforts more impactful.

(9) proposes an alternative way of how to evaluate and research aspects of debugging. This framework and methodology has already been successfully used for obtaining the inspiration to design the tool (10), which we will mention later.

We see the main differences between (9) and our work in two points. The first is that we use a different categorization of root causes of software errors. Our main aim was to compare which of thinking-based or implementation-based bugs are more time-consuming for investigation. (9) tries to capture the process how bugs are created by defining four layers where defects can appear. These layers are named *Specification*, *Programmer*, *Programming System* and *Program*. The bug is created if the defects on each layer are connected. For these layers (9) adapts categorization of human errors from (11).

The second difference is in the way of data collection. (9) records the debugging activities of participants while they are thinking aloud. The work also contains the best practices of how to facilitate a think-aloud experiment without being intrusive or inconvenient to participants. In our opinion, the main advantage of our approach is that we can collect more data with much less resources. On the other hand, we can collect much less thoughts of participants, so our root cause analysis may not go very deep into human psychology as (9) goes.

5.2. Design of new debugging tools and their evaluation

During decades computer scientists have developed many debugging approaches and tools, and here we would consider beneficial to review and map how much has this research been based on HCI findings and in what degree has the usefulness been evaluated. Doing such a review is outside of the scope of this work and therefore, we just present a few papers with various level of evaluation and HCI-evidence based support.

In recent years (10) introduced the Whyline tool that enables developers to get answers on why and why not questions. The tool was based on the preceding HCI research (9). The evaluation was done on two groups of ten programmers who were mostly experienced. The first group used breakpoints and usual tools for debugging and the second group used Whyline. The debugged software had 150 000 LOC and it was unknown to the participants. The designers of the evaluation study introduced two realistic bugs into the software that both occurred in the past. In our opinion, the first bug was not very difficult for investigation and the second one was more demanding. The participants received training how to use debugging tools that they were going to use. From the methodology perspective we see an interesting point of instructing the participants to focus on speed rather than on correctness. This is likely to make the results of different participant more comparable as, in our opinion, some programmers may refuse to hand over their work until they are very sure of correctness, which would slow them down a lot. We see this approach reasonable for situations where the programmers are not familiar with the code and their time for investigation is limited. The evaluation results were based on the comparision of how often were the bugs investigated successfully and how fast that was done. Furthermore, Whyline users provided their (very positive) feedback.

Many programmers would welcome the possibility to control the execution of their code in the reversed order, which is usually technically demanding to implement, and therefore such a debugging facility is not typically available. (12) describes an experimental debugger that makes the backward debugging possible. The motivation behind the work is supported just by a discussion that pinpoints several reasons why the reverse execution would be useful. The work presents no evaluation of the usefulness and thus, we perceive it as a representative of papers that are focused primarily on technical aspects.

(13) presents an interesting tool called DARWIN that is related to automated debugging. The tool takes two versions of the debugged software, the newest one faulty and some older one correct, and the input on which the faulty version fails. The output of the tools is a list of places where the bug is likely introduced in the newest version. The reasoning behind this design is that many software projects are covered by a testsuite, which is executed regularly. Adding DARWIN to the testsuite would allow the developers to immediately see suggestions what could be the root cause of each failed testcase. In the presented evaluation the authors try out the tool by themselves on localizing a few realistic bugs in libPNG, miniweb-apache and savant-apache projects.

The DebugAdvisor (14) is a tool for search of similar bug reports. Programmers can use it when they encounter a bug to check whether there was a similar reported and solved issue in the past. The motivation for having such a tool is based on a study (15) that was done in Microsoft's Windows Serviceability group. The evaluation efforts were, in our opinion, extensive and they were performed directly in the field. The search service indexed 2.36 million records (bug reports, attachments, logs, etc), had 129 users and 628 queries in one month. The feedback was received 208 times with 78% searched results viewed as helpful. Furthermore, the authors tried to resolve 20 active bugs by using the DebugAdvisor. Three bugs were solved immediately, and in other 12 cases the DebugAdvisor's output was perceived as useful.

5.3. Studies of debugging or software empirical research in general

As this work is not supposed to provide a comprehensive review of what has been researched so far, we mention only papers that we found most interesting or relevant. For obtaining a broader perspective on empirical research of software engineering check these papers, because they sometimes contain a more detailed review and further links.

A summary of debugging-related research is provided by (4). It is mainly concerned about educational perspective of software development, but regardless it can be used for orientation of what had been done in research of debugging up to 2008. Designers of experimental studies like this one may find valuable suggestions from (16) about how to do studies in software engineering.

(9) summarizes categorizations of software bugs that has been used in the history and the work argues that:

"To fully understand how the interaction between a programmer and a programming system can lead to software errors, we need a more general discussion of the underlying cognitive mechanisms of human error."

Then it proposes its own typology of software errors that is based on research of how people make errors in general. What we find especially strong about the proposed typology is that it interconnects topics of the cognitive psychology and debugging. In that way, it allows researchers to reason about the aspects of debugging in a deeper way than we have seen in most related work. In (17) a study on the differences of how efficient and less efficient programmers navigate in the source code was done. From our point of view, the work is interesting for its methodology and way of data analysis. Five programmers were recorded when they were focused on their programming task. Researchers created transcripts of their work that contained relevant events for the study and made conclusions from that. The work is inspired by (18), which is a useful review of methods for doing qualitative empirical research in software engineering.

The rise of large open source projects in the last two decades provided researchers with databases of bug tracking systems, which added one more possibility how to do empirical research. One example of such an empirical study of the databases is (19). We see a very strong side of investigating these databases that the amount of reported bugs is many times more than 100 000, which is very suitable for quantitative analysis. The weak point of this approach is, in our opinion, that the databases are not made for research primarily, so the detail of recorded data does not have to be sufficient to study many research questions.

6. Conclusion

In this work we designed a methodology of how to evaluate usefulness of debugging tools. The tools were evaluated in the environment of operating systems development and the results were presented in the section 4.1. Furthermore, we used the chance to perform an exploratory study on other aspects of how programmers debug an operating system.

We see the most significant contributions of our work in the following points:

- We presented that it does not matter whether the programmers use a graphical debugger, a command-line debugging, or just debugging messages, because the difference of mean debugging time is not statistically significant for all these three choices.
- We proposed a methodology of how to evaluate usefulness of debugging tools and applied it in an environment of operating system development.
- The motivation behind design of our evaluation and other exploratory methods was discussed. That may be valuable to anybody who wishes to evaluate usefulness of tools in general or who wishes to make another similar study.
- During the first two runs of the operating system course, we collected 662 somewhat large debugging reports (each report took about 3 minutes to fill in) and in the third run we obtained 62 reports about debugging intents. We provide these data in the form of csv and SQL database on the attached CD.

Bibliography

1. **Covey, Stephen Richards.** The 8th Habit: From Effectiveness to Greatness. London : Simon & Schuster UK Ltd, 2006, Appendix 4.

2. **Figueira, José, Greco, Salvatore and Ehrgott, Matthias.** *Multiple Criteria Decision Analysis: State of the Art Surveys.* s.l. : Springer Science+Business Media, Inc., 2005. ISBN 0-387-23067-X.

3. **Kahneman, Daniel, et al.** When More Pain Is Preferred to Less: Adding a Better End. *Psychological Science.* 1993, Vol. 4, 6, pp. 401-405.

4. **McCauley, Reneþe, et al.** Debugging: a review of the literature from an educational perspective. *Computer Science Education.* 2008, Vol. 18, 2, pp. 67-92.

5. **Beizer, Boris.** *Software testing techniques.* 2nd ed. s.l. : Intl Thomson Computer Pr (T), 1990. ISBN 1850328803.

6. **Parnin, Chris and Orso, Alessandro.** Are Automated Debugging Techniques Actually Helping Programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* 2011, pp. 199-209.

7. **Wallace, Clark and Gantt, Henry L.** *The Gantt chart, A working tool of management.* New York : Ronald Press, 1923.

8. **Boehm, Barry W. and Papaccio, Philip N.** Understanding nad Controlling Software Costs. *Transactions on Software Engineering.* 1988, Vol. 14, 10.

9. **Ko, Andrew J. and Myers, Brad A.** A frameworkand methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing.* 2005, Vol. 16, pp. 41-84.

10. —. Debugging reinvented: asking and answering why and why not questions about program behavior. *International Conference on Software Engineering.* 2008, pp. 301-310.

11. **Reason, James.** *Human Error.* Cambridge, UK : Cambridge University Press, 1990. ISBN 978-0521314190.

12. **Agrawal, Hiralal, DeMillo, Richard A. and Spafford, Eugene H.** An execution backtracking approach to program debugging. *IEEE Software.* 1991, pp. 21-26.

13. **Qi, Dawei, et al.** DARWIN: An Approach for Debugging Evolving Programs. European Software Engineering Conference and Foundations of Software Engineering. 2009.

14. **Ashok, B., et al.** DebugAdvisor: A Recommender System for Debugging. European Software Engineering Conference and Foundations of Software Engineering. 2009.

15. **Budge, S., et al.** Global software servicing: Observational experiences at Microsoft. *IEEE International Conference on Global Software Engineering.* 2008.

16. **Basili, Victor R., Selby, Richard W. and Hutchens, David H.** Experimentation in Software Engineering. *Transactions on Software Engineering.* 1986, Vol. 12, 7.

17. **Robillard, Martin P., Coelho, Wesley and Murphy, Gail C.** How Effective Developers Investigate Source Code: An Exploratory Study. *Transactions on Software Engineering.* 2004, Vol. 30, 12.

18. **Seaman, C.B.** Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering.* 1999, Vol. 25, 4, pp. 557-572.

19. **Zhou, Bo, Neamtiu, Iulian and Gupta, Rajiv.** A cross-platform analysis of bugs and bug-fixing in open source projects: desktop vs. Android vs. iOS. *International Conference on Evaluation and Assessment in Software Engineering.* 2015.

20. **Aziri, Brikend.** Job Satisfaction: A Literature Review. *Management Research and Practice.* 2011, Vol. 3, 4, pp. 77-86.

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Meaning
MFF UK	Faculty of Mathematics and Physics, Charles University, Prague
LOC	Lines of code
GUI Debugger	Graphical user interface debugger
GDB	GNU debugger
HCI	Human-computer interaction
MSIM	Simulator of a machine based on a MIPS family processor, http://d3s.mff.cuni.cz/~holub/sw/msim/

Attachments

1 Assignments of the operating system course

The exact and full assignment is located on the attached CD in Czech. The participants were supposed to implement three basic assignments and choose one extended assignment from three possibilities. In the following we summarize the assignments briefly:

- Basic assignment 1 – implementation of operating system core. The students were allowed to use some initial implementation (the operating system Kalisto) that shielded them from implementing most assembler-related code and had some requirements of the first assignment partially implemented. The core was supposed to have support for:
  - basic IO operations (formatted prints, ...)
  - basic debugging means (kernel panic, ...)
  - service of interrupts and exceptions
  - a simple memory allocator
  - threads
  - a simple scheduler
  - synchronization primitives (mutex)
  - timers
- Basic assignment 2 – Support virtual memory via the mechanism of page tables and TLB handling.
- Basic assignment 3 – Implement user space and the system call mechanism. Implement the init process and the runtime library. The runtime library is supposed to provide API for IO operations, debugging, dynamic memory allocation, threads and synchronization primitives.
- Extended assignment 1 – So far all the assignments were supposed to be executed on a single CPU machine. This extended assignment instructs the students to support multiple CPUs. The synchronization primitives should include spinlocks, semaphores, read-write locks and condition variables. Furthermore, non-blocking lists are supposed to be implemented and their performance compared against their blocking variant.
- Extended assignment 2 – Improve the performance of the basic assignment 2 by choosing proper data structures. Extend API for work with the virtual memory address space. Implement more heap allocators that use the allocation strategies first fit, next fit, best fit and worst fit. Furthermore, implement a multi-threaded performance test for the heap allocators and compare their performance.
- Extended assignment 3 – Implement a simple read-only driver for disk operations. Add API for work with processes. The init process should allow running other processes from the attached virtual disk device.

2 Content of the attached CD

This is the directories in the CD:

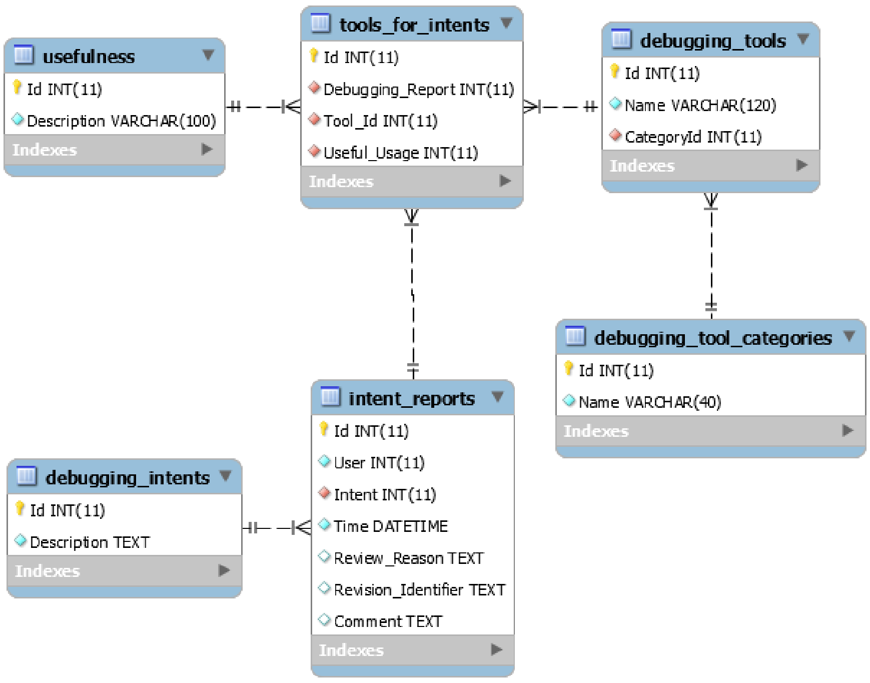
- data – exported mysql data in the form of SQL commands; these SQL tables and data can be imported to your SQL database
- database-structure - ER diagram pro Mysql Workbench
- assignment – full and exact assignment for the students in Czech language
- processing – R scripts and some processed data
- web – the source code of the created web interface

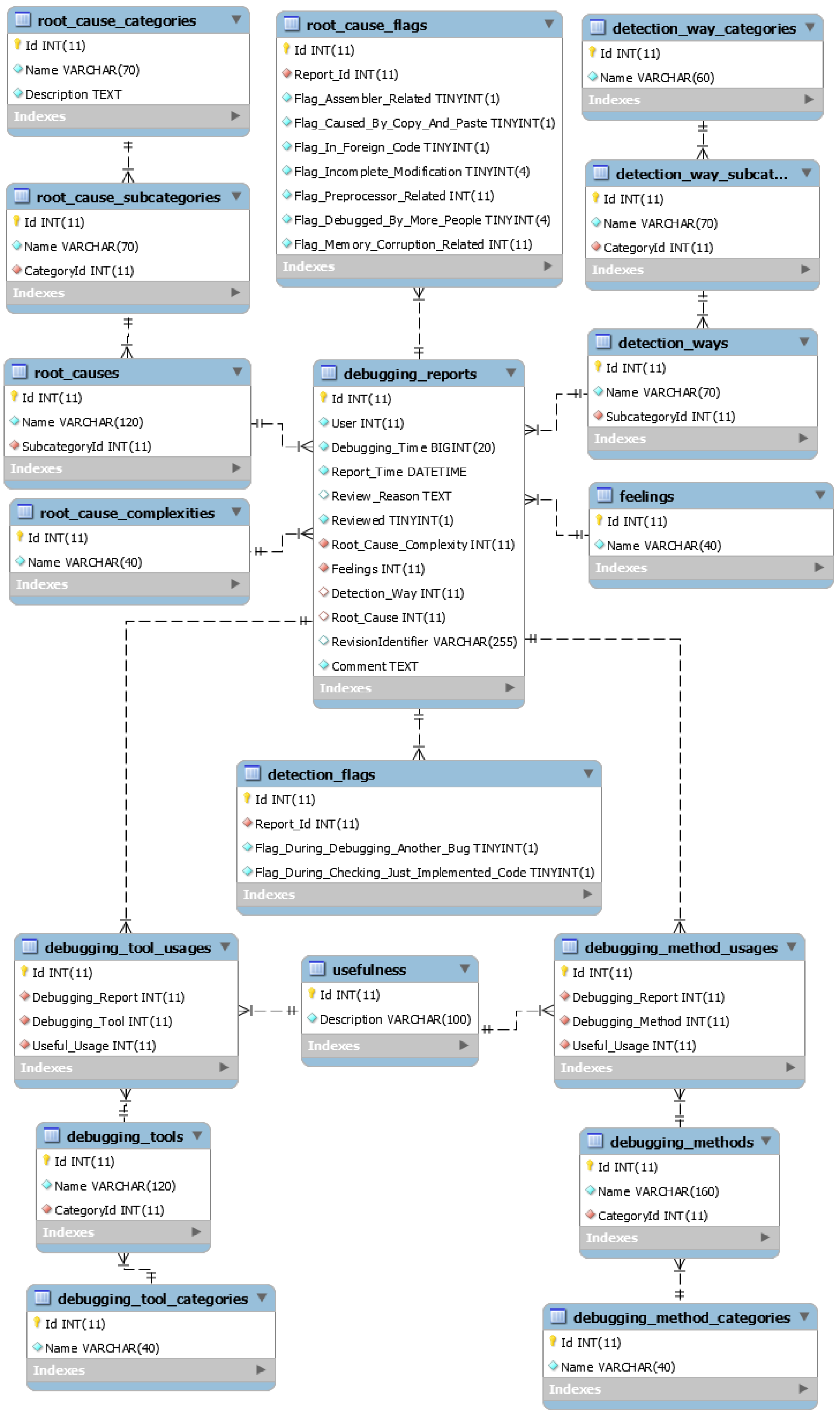
3 Structure of SQL database

In the following pictures we present the ER diagram of the mysql database that was used for storage of collected data. It is not easy to show the detailed relationship about individual tables, therefore we suggest you to open the data model in the mysql MySQL Workbench application to see more details.

The core tables are debugging˙reports for the first years of the study and intent˙reports for the last year of the study. Each record of these tables corresponds to one debugging report. The linkage with other kinds of data about the debugging report can be seen in the MySQL Workbench.

The participants are defined in a special table called users. Note that we defined some supervisor users and some users for debugging and testing purposes of the web interface. Therefore, the database contains some reports that should be filtered out if the data would be used for further analysis. The group of users that contain relevant data is named *Standard users*.




root_cause_categories
Id INT(11)
Name VARCHAR(70)
Description TEXT
Indexes
root_cause_flags
Id INT(11)
Report_Id INT(11)
Flag_Assembler_Related TINYINT(1)
Flag_Caused_By_Copy_And_Paste TINYINT(1)
Flag_In_Foreign_Code TINYINT(1)
Flag_Incomplete_Modification TINYINT(4)
Flag_Preprocessor_Related INT(11)
Flag_Debugged_By_More_People TINYINT(4)
Flag_Memory_Corruption_Related INT(11)
Indexes
detection_way_categories
Id INT(11)
Name VARCHAR(60)
Indexes
root_cause_subcategories
Id INT(11)
Name VARCHAR(70)
CategoryId INT(11)
Indexes
detection_way_subcat...
Id INT(11)
Name VARCHAR(70)
CategoryId INT(11)
Indexes
root_causes
Id INT(11)
Name VARCHAR(120)
SubcategoryId INT(11)
Indexes
debugging_reports
Id INT(11)
User INT(11)
Debugging_Time BIGINT(20)
Report_Time DATETIME
Review_Reason TEXT
Reviewed TINYINT(1)
Root_Cause_Complexity INT(11)
Feelings INT(11)
Detection_Way INT(11)
Root_Cause INT(11)
RevisionIdentifier VARCHAR(255)
Comment TEXT
Indexes
detection_ways
Id INT(11)
Name VARCHAR(70)
SubcategoryId INT(11)
Indexes
root_cause_complexities
Id INT(11)
Name VARCHAR(40)
Indexes
feelings
Id INT(11)
Name VARCHAR(40)
Indexes
detection_flags
Id INT(11)
Report_Id INT(11)
Flag_During_Debugging_Another_Bug TINYINT(1)
Flag_During_Checking_Just_Implemented_Code TINYINT(1)
Indexes
debugging_tool_usages
Id INT(11)
Debugging_Report INT(11)
Debugging_Tool INT(11)
Useful_Usage INT(11)
Indexes
usefulness
Id INT(11)
Description VARCHAR(100)
Indexes
debugging_method_usages
Id INT(11)
Debugging_Report INT(11)
Debugging_Method INT(11)
Useful_Usage INT(11)
Indexes
debugging_tools
Id INT(11)
Name VARCHAR(120)
CategoryId INT(11)
Indexes
debugging_methods
Id INT(11)
Name VARCHAR(160)
CategoryId INT(11)
Indexes
debugging_tool_categories
Id INT(11)
Name VARCHAR(40)
Indexes
debugging_method_categories
Id INT(11)
Name VARCHAR(40)
Indexes

4 Data evaluation and the processing script

The file *evaluation.R* contains R snippets that were used to evaluate the data. Our best practice was running the R in the *sql-data* directory and copy-pasting the pieces of R code into the R console. The script is not in the best shape, so you can take it as an inspiration of what data was processed and a detailed documentation how it was processed.

The CD also contains other scripts and files like *usefulness-evaluator.R*, which are there as a blind branch of our analysis. We keep them there mainly as backup if we would like to come back later to the ideas examined by those files.

5 Ways of detection

The following tables list the ways of bug detection that we identified and recorded in our study.

Way of detection Category Bad output	Comment
Wrong text in the output	
Kernel panic	
Wrong informative text (present in the code for a long time)	Typically debugging or diagnostic prints that get committed into the project repository
Wrong detailed text (present in the code for a short time)	
Program does not generate any output	
Random crashes or behavior	

Way of detection Category Formal methods	Comment
Report of static analysis	

Way of detection Category *Other*	omment
Program is weirdly slow	
Program is weirdly fast	

Way of detection Category *Other*	Comment
Report from another person	
During reading of the code	
Compiler warning	
Program never stops	
Test failed	
Assertion fired	
Just by thinking	
By using the debugger	
Environment crash (simulator, ...)	
Random crashes or behavior	

Each bug report had the following list of flags that enabled us to monitor other further aspects of way how the bug was detected.

Flags for the way of detection	Comment
During debugging of another bug	
During checking just implemented code	The bug was detected when the programmer checked a piece of code that had been just implemented

6 Root causes

The following tables list the root causes that we identified and recorded in our study.

Root cause Category Wrong logic or design, Data structures and algorithms	Comment
Misunderstanding properties of the structure or algorithm	
Misunderstanding how the structure or algorithm should be implemented	
Sequence of actions (algorithm) designed in a wrong way	

Root cause Category Wrong logic or design, Wrong design assumption	Comment
Wrong assumption about the user (or usage)	
Wrong assumption about the assignment	
Wrong assumption how software (function, library, ...) works	
Wrong assumption how hardware works	
Unconsidered corner case	
Unconsidered consequences of design decision	

Root cause Category Wrong logic or design, Synchronization	Comment
Violation of a critical section	Logic allows multiple threads to enter the critical section
Deadlock	
Livelock	
Missing synchronization	
Synchronization designed in another wrong way	

Root cause Category Wrong implementation, Initialization	Comment
Initialization is never done	
Initialization is sometimes not done	
Initialization is done in a wrong way	
Memory for a variable is not allocated	
Resource (handle, id, ...) is not allocated	

Root cause Category Wrong implementation, Subprogram binding	Comment
Bad scope of a variable	
Missing parameters (in case of variable count of parameters)	
Wrong order of parameters	
Wrong return value returned	

Root cause Category Wrong implementation, Finalization	Comment
Variable is not freed	
Resource (handle, id, ...) is not released	

Root cause Category Wrong implementation, Used wrong entity	Comment
Wrong subprogram called	
Wrong variable read	
Wrong constant used	
Wrong variable written	
Used another wrong entity	
Wrong multiple variables	

Root cause Category Wrong implementation, Other	Comment
Wrong options of compiler	
Correct implementation is unknown	
Misunderstanding of programming language	

Root cause Category Wrong implementation, Forgotten code	Comment
Missing assignment	
Missing commands	
Missing call of a subprogram	
Missing sub expression	
Missing check of error value (returned NULL, -1, ...)	

Root cause Category Wrong implementation, Dynamic data structures	Comment
Index out of bounds	
Index in bounds, but wrong	
Forbidden modification	
Wrong type conversion	

Root cause Category Wrong implementation, Dynamic data structures	Comment
Usage of freed memory	
Write to read-only memory	
Heap corruption	
Stack corruption	
Stack overflow	
Other memory corrupted	
Wrong address space used	
Unaligned address accessed	

Root cause Category Wrong implementation, Memory	Comment
Usage of freed memory	
Write to read-only memory	
Heap corruption	
Stack corruption	
Stack overflow	
Other memory corrupted	
Wrong address space used	
Unaligned address accessed	

Root cause Category Wrong implementation, Wrong expression	Comment
Wrong logic operator	
Wrong arithmetic's operator	
Sub expressions evaluated in a wrong order	
Wrong relation or comparison operator	

Root cause Category Wrong implementation, C specific	Comment
Missing volatile keyword	
Wrong written macro	

Root cause Category Wrong implementation, Assembler specific	Comment
Ignored behavior of the branch delay slot	
Unexpected compiler optimilization	
Violation of ABI rules	
Wrong directives for compiler (set .noreorder, ...)	

Root cause Category Wrong implementation, Value corruption	Comment
Integer overflowed or underflowed	
Mixed signed and unsigned integer	

Root cause Category Wrong implementation, Wrong program flow	Comment
Wrong condition in an if command	
Wrong condition for the end of a loop	
Unhandled case in a switch command	
Wrong propagation in a switch command	
Wrong cases in a switch command	
Wrong count of loop iterations +1	
Wrong count of loop iterations by more than 1	
Wrong order of commands	

Root cause Category Wrong implementation, Extra code	Comment
Extra assignment	
Extra commands	
Extra call of a subprogram	
Extra sub expression	

Root cause Category Wrong implementation, Wrong (coding related) assumption	Comment
Wrong assumption how library (or module) works	
Wrong assumption how function works	
Wrong assumption how language construct works	
Wrong assumption how environment works	

Root cause Category All other	Comment
Root cause not known	
Error from outside (simulator, compiler, ...)	
Not a bug	

Each bug report had the following list of flags that enabled us to monitor other further aspects of the root cause.

Flags for root causes	Comment
Assembler related	Does not have to be a bug in the assembler code. For example, the investigation included obtaining detailed information about stack operations.
Caused by copy and paste	
In foreign code	
Incomplete modification	The performed modification in the code was not done in every needed place.
Preprocessor related	
Debugged by more people	
Memory corruption related	

7 Methods and debugging tools

The following tables list the debugging tools and methods that we identified and recorded in our study.

Debugging tool Category *Other*	Comment
SVN (log, history)	
Objdump	
Own functions (in code) for debugging	Including debugging prints
Own debugging programs	Own scripts, stack analyzer, ...

Debugging tool Category GUI Debugger	Comment
Breakpoints and stepping	
Call stack window	
Disassembly window	
TLB window	This was implemented specifically into the GUI debugger, it is not a commonly available (and purposeful) feature
Memory breakpoints	
Memory view	
Registry window	
Variables window usage	

Debugging tool Category Evaluation of debugging data	Comment
Text processing (grep, sed, search in a text editor, ...)	
Excel, R, own script, ...	
Disassembly window	

Debugging tool Category MSIM debugging support	Comment
Instruction-level stepping and breakpoints	
Memory dump	
Inspecting registers	
Special instructions of MSIM	
Memory breakpoints	
Execution trace	

Debugging tool Category *GDB*	Comment
Instruction-level stepping and breakpoints	
Memory dump	
Inspecting registers	
Special instructions of MSIM	
Memory breakpoints	
Execution trace	

Debugging tool Category Obtaining information	Comment
Reading documentation	
Web search (google, forums, ...)	
Questioning community	

Debugging tool Category Formal methods and verification	Comment
Tools for static analysis	Any tool that does code analysis before run-time

Debugging method Category Looking for suspicious behavior	Comment
Looking for suspicious content of variables	
Looking for suspicious content of the memory	
Looking for suspicious flow of the program	
Creating a log or a trace	

Debugging method Category Summarizing the problem	Comment
Discussing the problem (written or spoken)	
Documenting the problem	
Drawing charts or diagrams	

Debugging method Category Making the localization easier	Comment
Making the bad behavior reproducible	
Reducing the input of the program	
Refactoring code	

Debugging method Category *Other*	Comment
Minimizing the input of the program	
Going through a chain of bad variables	
Re-reading the source code	
Searching for the problematic part of the code	
Disabling parts of code	
Generating hypotheses and checking them	
Changing the program (what-if approach)	
Writing a test	
Investigating probable location of the bug	
Changing build process (rebuilding, O2 -> O0, ...)	