Debugging and Supporting Software Systems

By:

Warren Myers

Debugging and Supporting Software Systems

By:

Warren Myers

Online:
< http://cnx.org/content/col11350/1.2/ >

CONNEXIONS

Rice University, Houston, Texas

This selection and arrangement of content as a collection is copyrighted by Warren Myers. It is licensed under the Creative Commons Attribution 3.0 license (http://creativecommons.org/licenses/by/3.0/).
Collection structure revised: August 29, 2011
PDF generated: March 5, 2012
For copyright and attribution information for the modules contained in this collection, see p. 22.

Table of Contents

1 Troubleshooting Software Systems 1
2 Creating Effective Support Tickets 7
3 Submitting Bug Reports 11
4 Creating a Supportable and Maintainable System 17
Index 21
Attributions 22

Chapter 1

Troubleshooting Software Systems[1]

Introduction

One of the best trainers I ever had taught the incoming crop of support engineers at Opsware (of which I was a member) that Support is all about applying hatchets to problems to make them easier to handle - when someone is calling for help, they [typically] have a major problems that is impacting their job, and need a solution to it last week. The product we were being trained on came on 2 full DVD iso images (it has since grown to three, dual-layer DVD iso images). That's a **lot** of potential area for errors to occur - whether from bugs in the application, or user mistakes. After a while, you start to see patterns in incoming issues, which allows for quicker resolution of customer complaints - when you've seen the same problem pop up at a dozen locations, as soon as a fix is found for one of them, you can, most likely, apply that same solution to the next 11, and solve all of those problems "at once".

You will learn about a host of hatchets you can use to narrow-down problems from the initial symptom of "it doesn't work" or "it broke" to the root cause, or viable workarounds.

The techniques described can be applied to other areas as well, but the focus will be on software systems.

Overview

We will cover an array of hatchets:

- Stop, Drop, and Roll
- What Changed
- Logging Output / Log Files
- Effective Searching
- Debugging Tools
- User Error
- Post-Mortem Data Collection
- Pro-Active, Preventative Measures

1.1 Stop, Drop, and Roll

When encountering software issues, whether in the smallest of scripts, or in enterprise tools, is to Stop, Drop, and Roll. Yes, those same three words you learned from the fireman as a child for what to do if your clothes ever catch fire.

Famous last words in most cases are, "I know what I'm doing" - you may very well, but always guess first that you don't. This is not to insult your intelligence, but rather to remind you that everyone makes mistakes!

Things to do before blindly going on:

---

[1] This content is available online at <http://cnx.org/content/m40793/1.3/>.

- Take note of all error messages returned from the failed process
- See if the error is something you have seen before (such as "Permission denied")
- Make sure you are running as the correct user / with proper privileges
- Make sure you have enough space to continue the task

1.2 What Changed

Were you able to successfully accomplish the task at hand before? Did the script run successfully yesterday, but not today? Can someone else run this correctly and I can't?

If the answer to these, and similar questions, is "yes", then you need to find out what changed between that last time you did this and now.

Things that may have changed:

- Your user's permissions
- The contents of the script/tool
- Free space you have access to (ie, maybe you're nearing your quota)
- System changes (patches, updates, etc)
- Remote resources are inaccessible (maybe it relies on a file server that is down for maintenance)

If you can undo any of the changes, does the tool work again? For example, if you are nearing your quota space but you delete some files, will it then run? If so, maybe you need your quota expanded. When the network file server is back up, does it run correctly?

1.3 Logging Output / Log Files

If the first basic checks don't yield any useful data, it's time to start looking at the the output of the program. Does it create any log files when it runs? If so, are any error messages kicked-out when you run it? Can the log setting be turned up?

If the script does not generate its own log files, it's time to start generating them on your own. Many tools have commandline options that will increase verbosity - use them in conjunction with trapping the output using tools like `script` and `tee`.

1.3.1 script

One of the most useful tools to use when troubleshooting is `script`. From the `man` page for `script`: "Script makes a typescript of everything printed on your terminal". So, while it is running, everything you type and/or is displayed will also get recorded into a file set when you generate the `script` session. Such an output file could end up being invaluable to the support team for the product if you need to open a case, or to the developers of the tool if you need to create a Bug Report (Chapter 3).

1.3.2 tee

Another very useful tool is `tee`. The `man` page for `tee` is short, but gets right to the point: "read from standard input and write to standard output and files". In other words, if you pipe the tool you are running into `tee`, not only will it display on the screen, but it will be captured in the output file you have supplied as well.

For example, in addition to the following displaying the entire contents of the filesystem (that your user has permission to see) on screen, it will trap the output into a file called "filesystem.out" in your home directory: `ls -R / | tee ~/filesystem.out`

1.4 Effective Searching

Google (or Bing, Yahoo, DuckDuckGo, etc) is your friend when trying to find answers to technical problems. After checking the `man` pages for a given tool (if they exist), always take any error messages found to Google and see what may be found thereon. For example, an error kicked-out by Apache Tomcat is likely to be documented just by searching the error string on Google - and probably from the official documentation / known issues / publicly-viewable bug reports (eg http://tomcat.apache.org/tomcat-6.0-doc/api/org/apache/tomcat/jni/Error.html[2]).

Effective searching is far more than "let me Google that for you" - it is knowing both **what** to look for and **how** to evaluate the results. Weak searching is easy: type in what you're looking for and **hope**. Like going to eBay[3] and searching for "baseball card" - you'll get millions of results on just that query: not the most helpful thing in the world. Try "roger maris rookie baseball card", and your resultset drops to a manageable few dozen - those are results you can pick through effectively. Similarly, searching for "OutOfMemory" will yield millions of hits, whereas "bea weblogic outofmemory java error jvm 1.4[4] " may start to yield **useful** results in the top 2-5 links.

Becoming an Effective Searcher will translate over into a host of other arenas beyond mere troubleshooting: from finding photographs of an obscure HVAC part, to locating the best deals on travel, searching effectively is a skill that anyone can learn, hone, and benefit from in their daily lives.

Effective Searchers look at what they are trying to find, pick out the important parts (if it's an error message, the generic portion of the message, not the specific-to-the-instance portion), and start looking, refining as they go.

To become an Effective Searcher does take some practice, but there is no better time than the present to start. Learn your favorite search engine's advanced features. For example, with Google I can search just on apache.org by adding `site:apache.org` to my search. Quoting the text of what you want to find will also tend to push references of it higher.

A drawback to using search engines, though, is that since they try to index **everything**, you can get a lot of results that are other people asking more-or-less the same question you are trying to find the answer to. That's great if an answer was posted, but when the responses are a lot of "me too"-types, it can be frustrating.

Part of becoming an Effective Searcher is learning to identify authority[5] in resources found: "I had a paper to write several years ago on comparing AMD's x86-64 architecture and Intel's IA32 architecture for the companies' CPUs. Sources like Tom's Hardware Guide were helpful to see real-world comparisons between the competing products, but the true sources of authority on the products were AMD and Intel themselves. I printed large chunks of the manufacturer's technical documentation to backup conclusions I made in my paper... Authority of sources isn't assured by just one factor – author, publisher, host, length, etc – but rather by directly linking to the data used to produce the conclusions made by that source. No resource stands on its own as an authority on any topic. In order to establish credibility, any resouce must cite where their data came from – either through some kind of bibliography in the case of a paper, or experimental results, or that the resource is maintained by the people who designed and built what they're writing about... The means of determining authority needs to come down to the following factors: 1) is the article written in an intelligent form? 2) are the sources cited of an authoritative nature? 3) has the author written anything previously that can be considered authoritative? and 4) would someone who is a known expert in the field (perhaps a professor of the topic) agree that the source is not some crackpot?"

Another component in becoming and Effective Searcher is to learn the skill of skimming for important details - and ignoring everything else. If you don't, you'll end up like the person described in this XKCD comic[6] !

---

[2] http://tomcat.apache.org/tomcat-6.0-doc/api/org/apache/tomcat/jni/Error.html

[3] http://ebay.com

[4] http://google.com/search?bea+weblogic+outofmemory+java+error+jvm+1.4

[5] http://antipaucity.com/2006/10/23/authority-issues-online

[6] http://imgs.xkcd.com/comics/the_problem_with_wikipedia.png

Lastly, to become an Effective Searcher, don't be afraid to ask for help: there is likely someone sitting near you or available via instant message who can help you out, and would be happy to if you only ask them.

1.5 Debugging Tools

Linux typically has several useful debugging tools available out of the box. These include `gdb` (for the advanced user), `strace`, `ps`, `lsof`, `netstat`, `iostat`, `uptime`, and `top`. Many others exist as well, but these are the most-commonly utilized.

1.5.1 top

One of the quickest-to-use tools for a picture of the current state of a Linux system is `top` which displays current top processes running, system uptime, load average, CPU utilization, memory usage (real and swap), and other items.

1.5.2 uptime

For a quick view of the system uptime and load averages, run `uptime`.

1.5.3 iostat

`iostat` displays information about the current state of the disk I/O on the system

1.5.4 netstat

To see what network ports are currently open and listening, use `netstat`. For example, `netstat -an | grep 80` will display what is using port 80 (and 8080, and anything else that has '80' in its port number).

1.5.5 lsof

`lsof` will show what process is holding open a network port or file. To use "list open files" to see what process is holding port 80, run `lsof -i:80`

1.5.6 ps

To see a list of the process table, run `ps`. My favorite argument sequence is `aux` which gives lots of information back: `ps aux`

The similar call on a Solaris machine is: `ps -ef`

1.5.7 strace

For a fuller diagnosis of what a given process is doing, `strace` can be a lifesaver. It essentially wraps around the process in question (either by running `strace <program-name>`, or by attaching to a running process with `strace -p<pid>`

On Solaris, the similar tool is `truss`.

1.5.8 gdb

The GNU debugger, `gdb`, is a massively-useful tool in the right hands: tracking individual calls inside a program, setting breakpoints, etc: it should be learned by every developer, and known to advanced users.

1.6 User Error

"User error"[7] is among the most commonly-cited errors with software and systems: the operator did something the creators did not expect. To use a ubiquitous car analogy, it's "user error" if the driver hits the gas instead of the brake. One interesting article[8] makes the claim that there is [almost] no such thing as "user error", and that instead it should be the developers who make tools not resilient enough to handle **any** user (no, a car manufacturer can't make the gas act like the brake when you "meant to stop", but maybe software developers can make their products less error-prone, or at least have them give better errors when they do have a problem).

A spectrum of user-initiated errors:

- Typos (misspellings, fat-fingering, generally mistyping something)
- External environmental problems (eg unplugging a network cable)
- Clickos (ie, misclicks - akin to mistyping)
- Forgetfulness
- Etc

From personal observation, I would guess user error accounts for 70-80% of all errors seen.

1.7 Post-Mortem Data Collection

When something has gone so awry that it has violently crashed, or even taken out its host system, it's time for some post-mortem data collection - maybe even forensic analysis.

Core dumps, log files, and even images of whole drives can be investigated during a post-mortem analysis of problems seen: as your technical acumen grows, you'll be able to investigate more parts of these prior to escalating to the tool's support or development teams.

1.8 Pro-Active, Preventative Measures

Ideally, we would all live and work in a world where nothing ever failed, and everyone acted the way they are "supposed" to. Sadly, that world does not exist. So what can we do to help prevent issues in the first place, or respond more adeptly when they [inevitably] occur?

Some solutions are simple: add more memory to the system; increase swap space; verify storage quotas; make sure all the resources I need are available; etc. Many can be more complex.

If there is a set of "Known Issues" or release notes that come with a particular product, make sure you read and are aware of them: there is almost nothing more frustrating than finding out there is a known issue, but you didn't check the manuals first!

Asking "Why"

If you're on the administrative side of the technical world, and not just the end-user side, the other big thing to remember is to always ask "why". Why did it fail? Why did we miss the known issue? Why were we not notified a necessary resource was going to be down? Why was there no alert sent about resources nearing their limits? If you can ask (and answer) those, then you should be able to reduce the number of "why" questions you need to ask in the future - because hopefully you're solving problems before they arise.

1.8.1 "Future-proofing" - is it possible?

The idea of "Future-Proofing"[9] is to create an environment that can survive future developments without needing to be changed itself. A common example of this would be to look at the current and expected growth needs of the email infrastructure of an organization, and then size the mail servers to handle 15-25% more than the expected growth (ie 100 users today, adding 20% per year, size the environment today for 200 users in three years (173 expected, plus ∼15%). Or it could mean ensuring that data you are working with today in version 4.3 of some tool will be accessible when upgrading to 7.2 in 4 years.

---

[7] http://en.wikipedia.org/wiki/User_error

[8] http://electronicmuseum.org.uk/2009/06/26/there-is-no-pebcak/

[9] http://en.wikipedia.org/wiki/Future_proof

When relying on external vendors, guaranteeing your environment is future-proof may not be possible - they could decide to change database schemas, file formats, etc. Likewise, when relying on expected growth patterns, you may exceed those expectations (requiring additional licenses, hardware, etc), or you may not meet those plans, and have an unnecessarily oversized environment. Several mitigating strategies exist for these eventualities, but are beyond the scope of this lesson.

1.9 Closing Thoughts

You've completed this module, and so now you're ready to troubleshoot the most ornery problems in the most obscure corners of your system, right? Don't let me discourage you from that lofty goal: but the reality is that becoming a good troubleshooter takes time, practice, lots of exposure, practice, skimming skills, practice, and patience. Oh, and did I mention: practice!

Lots of professions require troubleshooting skills, and each has their own tricks and tips to follow: auto mechanics will check the OBDII and listen to a rattle; electricians look for wiring faults; doctors look at symptoms to come up with a diagnosis. Skills learned in one field may not always translate into another, but if you can learn the basics (which DO all transfer), then gleaning insights from others can only improve your own personal Bag O' Hatchets.

Chapter 2

Creating Effective Support Tickets[1]

Introduction
I spent quite a while working in an enterprise Tier 2/3 support role, and have had extensive experience interfacing with software systems support engineers both prior to and after that role in support. This guide is intended to provide an introduction to the most effective techniques for creating, managing, and closing a support case from both the supportee and supporter views. As a supportee opening a case, what are the salient things the individual who will respond to your case going to be looking for? And as a supporter trying to solve a problem, what would be the best thing(s) for your customer/reporter to provide to you?

While every application will have different specific needs, there are myriad overlaps between systems, and knowing how to interact effectively with the product's support team is crucial to the successful use of any environment.

> NOTE: If your vendor has a specific policy regarding ticket reporting and case creation, by all means follow their guidelines - if they vary from this guide, make sure you do what they ask, when they ask!

2.1 Identifying the problem

The first mandate of successfully navigating the support process is to properly identify the issue at hand. A solid knowledge of basic troubleshooting (Chapter 1) skills is helpful, but - especially the larger the tool - not only is domain expertise needed, individual components of a system may react in unexpected ways. For example, an error may manifest itself in a log file as a component getting an "OutOfMemory" message, when the real issue is that your JVM can't expand its memory usage from the system heap due to insufficient swap space[2] (a specific example that haunted me for over a week on one project).

Be sure to familiarize yourself with the system's log file locations: in all probability not only will they provide initial clues to the issue, but the support engineer will ask for them to be added to the case you create during the process. On a Linux/Unix system, this is typically in `/var/log/appname`.

2.2 Helpful case titles and descriptions

The most useful aspect of any support case is a helpful, descriptive title, and a clear explanation of the issues witnessed. The least useful title I can recall having seen was "problem". Not what the problem was, but just "problem". Try instead for something like "Cannot start tomcat6 on Ubuntu 10.10 x64" - this describes the basics of what you are trying to do, and where you are trying to do it!

Adding an initial description to our case title above might look like the following:

---

[1] This content is available online at <http://cnx.org/content/m40795/1.3/>.

[2] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4697804

> *After successfully running* `sudo apt-get install tomcat6` *on my Ubuntu 10.10 x64 server, Tomcat will not start.*
>
> *The following message is showing in* `/var/log/tomcat/stderr.out`*: ...*
>
> *Debugging steps taken: ...*

This shows the support engineer what you are seeing, and what you have already done - both of which you **will** be asked for!

Because you've shown at least some inclination towards being helpful, the support technician(s) are likely going to be more inclined to **want** to work with you to a resolution. In this specific example, you've identified some errors showing up in a specific log file - go ahead and add that log (or perhaps the whole log directory, if appropriate) to the case so that the engineer(s) you interact with can have a head start on their more advanced troubleshooting steps.

2.3 Communication

Effective solutions to support problems can sometimes take a long time - and not always because it's a "hard" problem to solve, but individual schedules, time zone differences, support engineer case load, local project demands, etc can all contribute to resolution times not being as optimal as everyone would like.

Remaining professional, even under pressure, is paramount to both sides staying calm, and keeping a specific case at the forefront. As a case opener, try to avoid yelling at the support personell you deal with: you are [likely] **NOT** the only customer with an issue, and they are a person, too. From the support engineer's perspective, telling your customer they are dumb, stupid, or silly is not going to win you brownie points. Think of it like being at a restaurant: if you are polite to the waitress, she is more likely to be polite back - it a virtuous circle. Alternatively, if you're rude to your waiter, he might be inclined to "forget" things, be sluggish in replying to requests, and all-around just not want to be around you. Professionalism, politeness, and general courtesy goes a very long way towards accelerating a case to resolution.

When support asks for more information, or for the case opener to run/do something and provide feedback, it is important to do so - it allows them to work towards an answer, and will [likely] shorten the overall life of the ticket. Just as important is for the assigned engineer to acknowledge when the customer has submitted something previously-requested - communication is a two-way street, and it shows that both sides want to see the issue resolved.

"Terse verbosity" (Chapter 3) is the term I like to use for communication in a support issue. Say as much as needs to be said, but no more; be descriptive (verbose), but to the point (terse). Neither side is looking for *War and Peace*. Both sides are looking for the relevant bits of information that will provide a solution. An example of terse verbosity is shown in the sample description above regarding Tomcat not starting on a server. The sentences could have been written as bullet points. While they are verbose, they are also terse.

> *Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." –Antoine de Saint-Exupery*

2.4 Resolution

The ultimate goal of any support inquiry is to achieve a resolution. Ideally, that resolution solves the problem found - perhaps via referencing documentation, providing a workaround, or fixing a configuration issue. Sometimes a resolution will be in the form of a bug report (Chapter 3). Other times it will be in the form of a "request for enhancement", or RFE[3] . And other times the resolution might be a paraphrase of "sorry, we can't do that".

---

[3] http://encyclopedia.thefreedictionary.com/Request+For+Enhancement

Achieving a resolution that is ideal for all parties is in the support engineer's best interest, as it will give the customer confidence in your abilities, the commitment of the vendor to their product, and enable the supportee to go about his task at hand in an effective manner. The resolution to our sample case above about Tomcat might be to add storage space (maybe it doesn't have enough temp space to launch itself), or maybe it's to install a newer version of the package, or perhaps it's a configuration issue that can be fixed with minimal changes to initialization parameters.

Regardless of what the resolution may be to a particular problem, it is important for all sides to remember that they are dealing with actual people - in today's digital lifestyle, it can be easy to be rude in an email, or ignore/delay responses to help requests. But being rude, or ignorning/delaying responses will only serve to alienate both parties - perhaps to the point of a resolution never being found, or the vendor-customer relationship to be permanently damaged. As a case opener, you are **the** face of your company to the support technician(s) who help you. Likewise, as the support case owner, you are **the** face of the company to the customer. Interactions in these situations are vitally important to maintaining the long-term health of the relationship between these two organizations.

Chapter 3

Submitting Bug Reports[1]

Introduction
I am deeply indebted to Simon Tatham's[2] **excellent** "How to Report Bugs Effectively"[3] - most of this is in the form of direct quotations, or a condensation of what he has so aptly written. I have taken some rewording liberties to expand the focus of his work, and remove personal references, as well. I appreciate Simon's permission to adapt his work here.

3.1 "It doesn't work"

Give the creators of the tool a little credit for some basic intelligence: if the program really didn't work at all, they would probably have noticed. Since they haven't noticed, it must be working for them. Therefore, either you are doing something differently from them, or your environment is different from theirs. They need information; providing this information is the purpose of a bug report. More information is almost always better than less.

Many programs, particularly free ones, publish their list of known bugs. Check this to see if the bug you've just found is already known or not. If it's already known, it isn't worth reporting again **as a new bug**. However, if you think you have more information than the report in the bug list, you might want to contact the developers anyway to add to the bug report. The new information you have may enable them to fix the bug more easily/sooner.

Specific developers / companies / groups have particular ways they like bugs to be reported. If the tool comes with its own set of bug-reporting guidelines, read and follow them!

3.2 "Show me", or "Show me how to show myself"

One of the very best ways you can report a bug is by showing it directly to the developer - either in person, or via a remote support session (like webex). Demonstrate the exact thing that goes wrong. Let them watch you run the software, watch how you interact with the software, and watch what the software does in response to your inputs.

They [should] know that software like the back of their hand: which parts they trust, and which parts are most likely to have faults. Intuitively, they know what to watch for. By the time the software does something obviously wrong, they may well have already noticed something subtly wrong earlier which might give them a clue. They can observe [almost] everything the system does during the test run, and they can pick out the important bits for themselves.

---

[1] This content is available online at <http://cnx.org/content/m40790/1.4/>.
[2] http://www.chiark.greenend.org.uk/~sgtatham
[3] http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

This may not be enough. They may decide they need more information, and ask you to show them the same thing again. They may ask you to talk them through the procedure, so that they can reproduce the bug for themselves as many times as they want. They might try varying the procedure a few times, to see whether the problem occurs in only one case or in a family of related cases. If you're unlucky, they may need to sit down for a couple of hours with a set of development tools and really start investigating. But the most important thing is to have the programmer looking at the computer when it goes wrong. Once they can see the problem happening, they can usually take it from there and start trying to fix it.

If you have to report a bug to a programmer who can't be present in person (or via remote support tools), then you want to ensure they can reproduce the problem. If they can't, they may have to assume that there's a user error.

Report **exactly** what you did leading up to the error. If it's a graphical program, report exactly which buttons you pressed, and in what order you pressed them. If it's a command-line program, capture precisely what command you typed - this is where tools like 'script' and 'tee' can again come in handy!

Make sure the bug report has all the input you can think of. If the program reads from a file, add a copy of the file. If the program talks to another computer over a network, while you can't send a copy of that device, you can report what kind of system it is, and (if you can) what software is running on it.

3.3 "Works for me. So what goes wrong?"

If you report a long list of inputs and actions, and the developer(s) attempt to recreate the problem, but nothing goes wrong, then you haven't given them enough information. Possibly the fault doesn't show up on every computer; your system and theirs may differ in some way. Possibly you have misunderstood what the program is supposed to do, and you are both looking at exactly the same display but you think it's wrong and they know it's right.

Therefore, you must also describe **what** happened. Report exactly what you saw. State why you think what you saw is wrong; better still, explain exactly what you expected to see. If you say something like "and then it went wrong", you have left out a lot of very important information. If you saw error messages, then report, carefully and precisely, what they were - they are important! At this stage, the developer is not trying to fix the problem: they're trying to find it. They need to know what has gone wrong, and those error messages are the system's best effort to tell you that. Write the errors down if you have no other easy way to remember them; it's not worth reporting the program generated an error unless you can also report what the error message was.

In particular, if the error message has numbers in it, do report them! Just because you can't see any meaning in them doesn't mean there isn't any. Numbers contain all kinds of information that can be interpreted by the developers, and they are likely to contain vital clues. Numbers in error messages are there because the computer is too confused to report the error in words, but is doing the best it can to get the important information to you somehow.

At this stage, the programmer is effectively doing detective work - they don't know what's happened, and they can't get close enough to watch it happening for themselves, so they are searching for clues that might give it away: error messages, incomprehensible strings of numbers, and even unexplained delays are all just as important as fingerprints at the scene of a crime. Keep them! If you are using a Unix-like environment, the program may have produced a core dump. Core dumps are a particularly good source of clues, so don't throw them away. Make sure you put in the bug report you have such dumps; the development team may want them added to the case. Be aware that the core file typically contains a record of the complete state of the program: any "secrets" involved (maybe the program was handling a personal message, or dealing with confidential data) may be contained in the core file.

3.4 "So then I tried ..."

If you're just starting out, or even if you're a seasoned professional, why did you violate the first fundamental rule of troubleshooting: Stop, Drop, and Roll (Chapter 1)?

If you've violated that first rule, you may very well be exacerbating the problem rather than ameliorating it. There are a lot of things you might do when an error or bug comes up. Many of them make the problem worse. For example, maybe you've deleted all your Word documents by mistake, but before calling in any expert help, you tried reinstalling Word, and then running a disk defrag. Neither of those will help recover the deleted files, and between them they will probably scramble the disk to the extent that no Undelete program in the world will been able to recover anything. There might be a chance if the system is left alone.

Users like this are like a mongoose backed into a corner: with its back to the wall and seeing certain death staring it in the face, it attacks frantically, because doing something has to be better than doing nothing. This is not well adapted to the type of problems computers produce. Instead of being a mongoose, be an antelope. When an antelope is confronted with something unexpected or frightening, it freezes. It stays absolutely still and tries not to attract any attention, while it stops and thinks and works out the best thing to do. (If antelopes had a technical support line, it would be telephoning it at this point.) Then, once it has decided what the safest thing to do is, it does it.

When something goes wrong, follow the first law of troubleshooting! Immediately stop doing anything. Don't touch any buttons at all. Look at the screen and notice everything out of the ordinary, and remember it or write it down. Then perhaps start cautiously pressing "OK" or "Cancel", whichever seems safest. Try to develop a reflex reaction - if a computer does anything unexpected, freeze. If you manage to get out of the problem, whether by closing down the affected program or by rebooting the computer, a good thing to do is to try to make it happen again. Programmers like problems that they can reproduce more than once. Happy programmers fix bugs faster and more efficiently.

3.5 "I think the tachyon modulation must be wrongly polarised"

It isn't only non-programmers who produce bad bug reports. Some of the worst bug reports ever written come from programmers, and even from good programmers. Take the example of the programmer who kept finding bugs in his own code and trying to fix them. Every so often he'd hit a bug he couldn't solve, and he'd ask a colleague over to help. "What's gone wrong?" they ask. He replies by stating his current opinion of what needed to be fixed.

This worked fine if his current opinion was right - it meant he'd already done half the work and together the problem could be solved. It's efficient and useful. But quite often he was wrong. Work could go on for some time trying to figure out why some particular part of the program was producing incorrect data, and eventually the discovery is made that it wasn't, that the investigating was in a perfectly good piece of code, and that the actual problem was somewhere else. There's wasted time.

Would you do that to a doctor? "Doctor, I need a prescription for Hydroyoyodyne." People know not to say that to a doctor: you describe the symptoms, the actual discomforts and aches and pains and rashes and fevers, and you let the doctor do the diagnosis of what the problem is and what to do about it. Otherwise the doctor dismisses you as a hypochondriac or crackpot, and quite rightly so. It's the same with programmers. Providing your own diagnosis might be helpful sometimes, but always state the symptoms. The diagnosis is an optional extra, and not an alternative to giving the symptoms. Equally, sending a modification to the code to fix the problem is a useful addition to a bug report but not an adequate substitute for one.

If you are asked for extra information, don't make it up! Don't skip any diagnostic steps you're asked to perform, because they will [hopefully] lead to a solution. Using your intelligence to help the programmer is fine. Even if your deductions are wrong, the developer should be grateful that you at least tried to make their life easier. But report the symptoms as well, or you may well make their life much more difficult instead.

3.6 "That's funny, it did it a moment ago"

Say "intermittent fault" to any developer, and watch their face fall. The easy problems are the ones where performing a simple sequence of actions will cause the failure to occur. They can then repeat those actions under closely observed test conditions and watch what happens in great detail. Sadly, too many problems simply don't work that way: there will be programs that fail once a week, or fail once in a blue moon, or never fail when you try them in front of the programmer but always fail when you have a deadline coming up.

Most intermittent faults are not truly intermittent. Most of them have some logic somewhere. Some might occur when the machine is running out of memory, some might occur when another program tries to modify a critical file at the wrong moment, and some might occur only in the first half of every hour!

If you can reproduce the bug but the programmer can't, it could very well be that their system and your system are different in some way and this difference is causing the problem. For example, a program whose window curled up into a little ball in the top left corner of the screen, and sat there and sulked. But it only did it on a user's 800x600 screen; it was fine on the developer's 1024x768 monitor.

The developers will want to know anything you can find out about the problem. Try it on another machine, perhaps. Try it twice or three times and see how often it fails. If it goes wrong when you're doing serious work but not when you're trying to demonstrate it, it might be long running times or large files that make it fall over. Try to remember as much detail as you can about what you were doing to it when it did fall over, and if you see any patterns, mention them. Anything you can provide has to be some help. Even if it's only probabilistic (such as "it tends to crash more often when Emacs is running"), it might not provide direct clues to the cause of the problem, but it might help the programmer reproduce it.

Most importantly, the programmer will want to be sure of whether they're dealing with a true intermittent fault or a machine-specific fault. They will want to know lots of details about your system, so they can work out how it differs from theirs. A lot of these details will depend on the particular tool, but one thing you should definitely be ready to provide is the version number. The version number of the program itself, and the version number of the operating system, and probably the version numbers of any other programs that are involved in the problem.

3.7 "So I loaded the disk on to my Windows ..."

Writing clearly is essential in a bug report. If the programmer can't tell what you meant, you might as well not have said anything. Be specific. If you can do the same thing two different ways, state which one you used. "I selected Load" might mean "I clicked on Load" or "I pressed Alt-L". Say which you did. Sometimes it matters. Be tersely verbose - explain everything you did, but use bullet points if possible, rather than prose form. Give more information rather than less. If you say too much, the programmer can ignore some of it. If you say too little, they have to come back and ask more questions.

Be careful of pronouns. Don't use words like "it", or references like "the window", when it's unclear what they mean. Consider this: "I started FooApp. It put up a warning window. I tried to close it and it crashed." It isn't clear what the user tried to close. Did they try to close the warning window, or the whole of FooApp? It makes a difference. Instead, you could say "I started FooApp, which put up a warning window. I tried to close the warning window, and FooApp crashed." This is longer and more repetitive, but also clearer and less easy to misunderstand.

Read what you wrote. Read the report back to yourself, and see if you think it's clear. If you have listed a sequence of actions which should produce the failure, try following them yourself, to see if you missed a step.

3.8 Summary

The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can't be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves. In case the first aim doesn't succeed, and they can't see it failing themselves, the second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down all error messages, especially if they have numbers in them.

When your computer does something unexpected, freeze. Stop, Drop, and Roll (Chapter 1). Do nothing until you're calm, and don't do anything that you think might be dangerous.

By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.

Be ready to provide extra information if the asked for it. If it isn't needed, it wouldn't be asked for. The developers aren't being deliberately awkward. Have version numbers at your fingertips, because they will probably be needed.

Write clearly. Say what you mean, and make sure it can't be misinterpreted.

Above all, be precise. Programmers like precision.

Chapter 4

Creating a Supportable and Maintainable System[1]

Introduction
The most important aspect of any systems project - whether an individual tool, a product release, or a site-specific implementation - is that it is supportable: the right kind of documentation, with the proper contents, can make that goal a reality. This guide draws on years of experience of development, support, and site implementations of projects ranging from simple scripts to enterprise-managing environments.

4.1 Types of Documentation

4.1.1 Developer-to-Support

Documentation from the development team to the support team is vital. Without it, any product will succumb to unsupportability, and die.

Ideal requirements from development:

- the documentation of the code (javadoc, doxygen, etc)
- functional specification (if it exists)
- flow chart/schema of the way the application works
- what can be backed-up vs what cannot (database(s), configuration files, etc)
- **details on the build process**
    - source code repository view/login information
    - ticketing system information
    - where to get current source
    - how to file bugs (Chapter 3) (they will happen)
    - deployment diagram
    - the "why" of how it works (ie design/implementation choices)
    - other software used
    - route to provide patches either to the source or to customers (ideally, both)
    - location(s) and format(s) of log file(s)
- **user manual**
    - how it works
    - features list

---

[1] This content is available online at <http://cnx.org/content/m40967/1.4/>.

  - system requirements (CPU usage, memory, disk space, etc)
  - specific configurations needed
  - command-line arguments/switches
  - screenshots of operations and output
  - user-customizable portions (eg there is a scripting component or available API)
  - start-up and shut-down procedures
- primary contacts for each component, aka the "escalation path"
- **testing/QA document**
  - where documentation is stored
  - default application usernames/password
  - details of server IP and default admin / oracle / websphere passwords
  - any support SQL/tools created by the development team (for analysis, loading data, etc)
  - all known issues with the current build
  - "unusual" dependencies (eg if the FQDN or simple hostname is changed, the application cannot start)
- encouragement for feedback from Support as to what else they want to see

Format
Almost equal in importance to the content of documentation is its format - the more open and accessible, the better. That may come in the form of wiki pages, html files, pure text, or PDFs. Whatever is chosen, though, needs to be easily and readily accessible to whomever needs to know - avoid proprietary formats like Microsoft Word.

> NOTE: "Support" may not exist for a given product as such - the "support" available may be in the form of a community of users with a mailing list or forum (such as is often the case with open source software[2]). The ideal documentation listed herein still applies, but its target audience will be a little different if it's not handled by a formal support group.

4.1.2 Field-to-Support

When products are implemented at a customer site, ideally the vendor's support team will receive a set of handover documentation to ease their lives. At one point in my career, I was involved with creating, and then maintaining/improving, the field-to-support hand-off document for the product I worked most heavily with. That document helped alleviate headaches experienced both by support getting a new customer, and future field work wherein changes were perpetrated on an existing environment via upgrades, extensions, etc.

Basic components of the field-to-support documentation

- customer name and contact information
- field representative(s) contact information
- platform(s) used in delivery (eg, Windows Server 2008 R2 for SQL Server host, RHEL 5.5 x64 for application, etc)
- hostname(s) of server(s) used in the deployment
- hardware specifications of the server(s) utilized (eg 8 2.4Ghz CPU cores, 16GB RAM, 73GB local storage, 300GB SAN storage, dual Gig-E (10.10.10.5 and 10.10.20.5))
- verification that prerequisite packages are installed (eg the out put of `rpm -qa` on a Linux system)
- customer sign-off on basic functionality
- notated list of non-tested / non-functional component(s) (and why they were not tested / don't care they don't work, etc)

---

[2] "Planning for Focused Modularity in Open-Source Software Development" <http://cnx.org/content/m12403/latest/>

- customer sign-off on any site-specific configurations or customizations
- copies of all customized configuration files
- copies of all field-developed add-ons / customizations

> NOTE: Of course, the specific individual components of any given field-to-support hand-off documentation would be modified for a given product.

4.1.3 Support-to-Others

Often enough, issues, bugs, and other "gotchas" are not found by developers or by the QA team - they are found (Chapter 2) by end-users of a given tool. Documenting those items back from the field so others can benefit, or so bugs (Chapter 3) can be resolved, is a great boon.

Most often, these common issues will be collated into a Frequently Asked Questions list (FAQ) or Knowledge Base (KB). FAQ and KB articles generally prove invaluable to many parties - other support engineers, customers, developers, management, sales, etc.

Sources for KB articles

Knowledge Base articles generally form from two primary sources - support tickets, and forums (internally or externally facing). Common issues can often be better solved by creating one good "how-to" or "workaround" article instead of having users ask the same thing over and over again (but worded differently each time). this saves support engineer time, customer time, and makes all parties involved happier.

Internal vs external KB articles

Depending on the product, there may be a wide array of information that customers "cannot know" - who wrote what, similar failures at other customers, who other customers are, etc. Likewise, some documentation available to aid in troubleshooting a problem may be in the form of saved chat transcripts, raw wiki journals, poorly-written notes, etc. It is up to the support engineer to cull both internal and external data into a form that an end-user can benefit from.

At one job I had, we had both internal and external sources: a wiki and a Plone[3] instance were used internally, along with IRC[4] ; externally we had a small-but-growing KB database. Most of us who worked in support at the time also had our own "crib notes"[5] of things we'd run into before that we drew on to answer new issues. All of these sources were routinely exercised to help an ailing customer with his problem *du jour*.

4.2 Extra Credits

Some of the materials in this guide have been adapted from content on StackOverflow.com[6] and ServerFault.com[7] .

Specifically the following:

- What are the core elements to include in Support Documentation?[8]
- Support / maintenance documentation for a development team[9]
- What documentation is helpful when supporting an application?[10]
- How do you document a network?[11]

---

[3] http://plone.org

[4] http://en.wikipedia.org/wiki/Internet_Relay_Chat

[5] http://en.wikipedia.org/wiki/Cheat_sheet

[6] http://stackoverflow.com/questions/tagged/support+documentation

[7] http://serverfault.com/questions/tagged/documentation

[8] http://stackoverflow.com/q/205374/4418

[9] http://stackoverflow.com/q/4723685/4418

[10] http://stackoverflow.com/q/2946520/4418

[11] http://serverfault.com/q/12378/2321

- Documentation As-A-Manual vs. Documentation As-A-Checklist[12]
- Complex software installation documentation procedures or tools[13]
- Getting started with documentation[14]
- How are you documenting your work, processes and environment?[15]
- How do you document windows server configurations?[16]
- How to deliver documentation for IT tools?[17]
- What level of documentation do you expect to be provided to you by developers?[18]
- I am about to create a site bible aka documentation of our web server?[19]
- How to document linux server configuration?[20]
- How do you document your Cisco configurations?[21]
- What to document in my network/domain?[22]
- IT Documentation Platforms[23]

---

[12] http://serverfault.com/q/25404/2321
[13] http://serverfault.com/q/18553/2321
[14] http://serverfault.com/q/46273/2321
[15] http://serverfault.com/q/3970/2321
[16] http://serverfault.com/q/43970/2321
[17] http://serverfault.com/q/38326/2321
[18] http://serverfault.com/q/3043/2321
[19] http://serverfault.com/q/169827/2321
[20] http://serverfault.com/q/150788/2321
[21] http://serverfault.com/q/44134/2321
[22] http://serverfault.com/q/170326/2321
[23] http://serverfault.com/q/228359/2321

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

A assistance, § 2(7)

B bug, § 3(11)
bugreport, § 3(11)
bugs, software, troubleshooting, debugging, problems,, § 1(1)

C caveat, § 4(17)
communication, § 2(7)

D development, § 4(17)
documentation, § 4(17)

H help, § 2(7)

M maintenance, § 4(17)

R report, § 3(11)

S software, § 2(7), § 3(11)
specification, § 4(17)
support, § 2(7), § 4(17)
system, § 4(17)

T techniques, § 1(1)
troubleshooting, § 3(11)

V vendor, § 2(7)

Attributions

Collection: *Debugging and Supporting Software Systems*
Edited by: Warren Myers
URL: http://cnx.org/content/col11350/1.2/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Troubleshooting Software Systems"
By: Warren Myers
URL: http://cnx.org/content/m40793/1.3/
Pages: 1-6
Copyright: Warren Myers
License: http://creativecommons.org/licenses/by/3.0/

Module: "Creating Effective Support Tickets"
By: Warren Myers
URL: http://cnx.org/content/m40795/1.3/
Pages: 7-9
Copyright: Warren Myers
License: http://creativecommons.org/licenses/by/3.0/

Module: "Submitting Bug Reports"
By: Warren Myers
URL: http://cnx.org/content/m40790/1.4/
Pages: 11-15
Copyright: Warren Myers
License: http://creativecommons.org/licenses/by/3.0/

Module: "Creating a Supportable and Maintainable System"
By: Warren Myers
URL: http://cnx.org/content/m40967/1.4/
Pages: 17-20
Copyright: Warren Myers
License: http://creativecommons.org/licenses/by/3.0/

Debugging and Supporting Software Systems
A guide to troubleshooting and supporting software systems from end-user, administrator, and developer points-of-view

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.