Design for Maintainability

Maintainability by Design

Amir Raveh & Ofra Homsky
42 Bitzaron St.
Tel-Aviv 67894
Israel
Email: tngt@netvision.net.il, amirr@netvision.net.il

"Hardware: those parts of the system you can kick.

Software: those parts of the system you can merely curse" (anon).

Introduction

We have all been in this scenario - a computer, a device, a system or software that does not perform as expected.

The impacts of such malfunction range from minor discomfort and frustration all the way through loss of life work, or even loss of life and limb [1].

The reasons for software problems vary widely - they range from programming errors and hardware failures, through deviations in behavior of the environment from the one assumed during its design.

This is further complicated by human nature - sometimes the problem is in the user's perception of how this software is supposed to work. Most software systems do not provide an adequate conceptual model for the user to understand their inner functionality, so the customers using the software make their own assumptions about such models.
Blame is another factor that further complicates problem resolution – we are quite used to people attributing human mistakes to "computer errors". The opposite also occurs – when people encounter a problem in using software they may engage in self-blaming ("I really don't understand computers, I must have done something wrong"). [2][3].

And it is at this point that we all meet the phase of software development that is least discussed - the maintenance phase.

If we have a support contract, we use it hoping to obtain a fix or at least an explanation for malfunctions. In other cases call on technicians or use online forums such as newsgroups to obtain assistance.

On the other side of the line there is a support person, who meets the customer at one of the worst moments in life – the machine has stopped working, a system is now leaking, they just lost a few hours worth of work, a collection of love letters, or have a blue screen on a life supporting system saying there is a General Protection Failure at some obscure hexadecimal address.

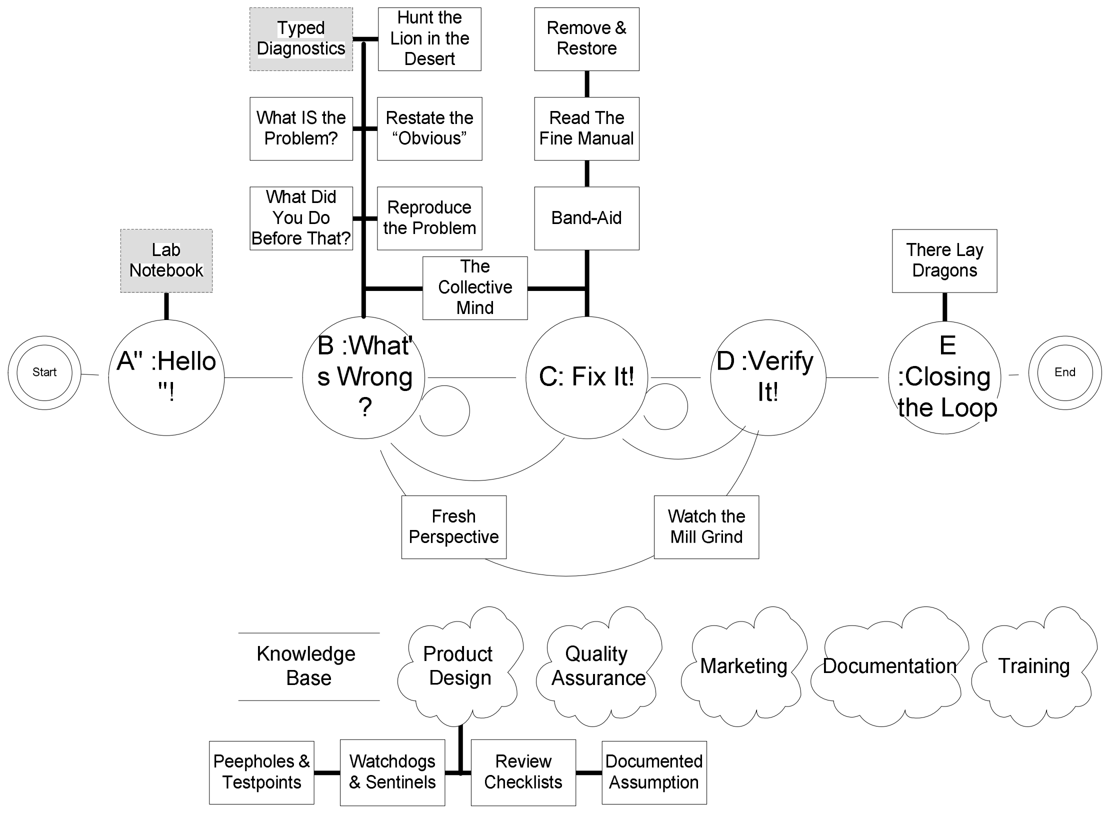
The customer is angry, aggravated and anxious, sometimes pressed for time and definitely wants it all solved, fixed, corrected and restored immediately.

The support person now begins a process of problem resolution. S/he needs to go through the steps in a systematic order, despite the pressure from the customer for expeditious solution.

The skills called for are many: a combination of crisis containment worker, detective, in-depth professional knowledge of the system or software in question and its environment.

We all read some horror stories from both sides of the fence. The existence of a vast quantity of Internet folklore [4] and urban legends [5] are indications of how widespread and troublesome the process of problem resolution can be.

This pattern language tries to shed some light on the aspects involved in software maintenance and troubleshooting. It is written for software designers, architects, support engineers, systems engineers, technical documentation writers, marketing people and managers.



The problem resolution process model is based on the one presented by Limoncelli [6]. Text in `Courier font` is quoted from Limoncelli's process description, text in regular font was added by the authors.

- Phase A: The Greeting (``Hello")

  •Step 1: The Greeting

`The customer is greeted by a person or a problem collection mechanism, and reports the problem encountered. This might be done by phone, email, Web page, walk-up helpdesk, dropping in the system administrator's office, using a custom application or by a report of an automated monitoring system (such as network performance monitor).` It is from this point that the reported problem should be assigned an identifier (such as a problem report number), and that all actions, hypotheses and interactions should be recorded in a **'Lab Notebook'** [11] – i.e. a call log, customer request file. This recording is aimed at providing a clear understanding of what was the initial complaint about, and recording all the steps that were carried out until the problem was closed.

- Phase B: Problem Identification (``What's wrong?")

  •Step 2: Problem Classification

`A support person or system (such as the dreaded IVR menu system) classify the problem and assign its resolution to a support person with the presumed skill set for working on its resolution.` This can be facilitated by preparing error messages and on-board diagnostics easy to use and understand by the customer, providing clear information about failures, some of which are described in **'Typed Diagnostics'** [7]

  •Step 3: Problem Statement

`The customer states the problem with as full details as possible and this information is recorded. This person is often the same person as the classifier.` The skill required by the recorder in this phase is the ability to listen and ask the right questions to draw out the needed information from the user. The recorder extracts the problem statement and records it. This is facilitated by using **'What *IS* the Problem'**, **'What Did You Do Before That?'** and **'Re-State the "Obvious"'** patterns.

  •Step 4: Problem Verification

`The support person tries to` **'Reproduce the Problem'**. `If the problem cannot be reproduced, often the problem being reported is not being properly communicated and one must return to Step 3 (Problem Statement).` If the problem is intermittent, then this process becomes more complicated but hopefully not impossible.

  •Step 5: Problem Isolation

We suggest adding this step to the Limoncelli process model, before proceeding to `Solution Proposals.`

Sometimes the problem statement and its reproduction are not enough to properly identify where the problem is. The support person tries to determine exactly what is

broken down – what is the minimal sub-system that is affected by the problem, which input triggers it, where is the “earliest” in the chain of events that the problem manifests itself.
‘Hunt the Lion in the Desert’, ‘Reproduce the Problem’, ‘The Collective Mind’, ‘Peepholes & Testpoints’ and ‘**Remove & Restore’** patterns are helpful in this step.

- Phase C: Planning and Execution (``Fix it'')
  - •Step 6: Solution Proposals

The possible solutions are enumerated. This role is performed by a ``Subject
Matter Expert''. Depending on the problem, this list may be large or small. For some problems the solution may be obvious and there is only a single proposed solution. Other times there are many possible solutions. Often verifying the problem in the previous step helps finding possible solutions. Solutions can be sought out by using **‘Remove & Restore’**, **‘The Collective Mind’, ‘RTFM – Read The Fine Manual’** and **‘Documented Assumptions’**

•Step 7: Solution Selection

Once the possible solutions are enumerated, one of them is selected to be attempted first (or next, if we are looping through these steps). The Subject Matter Expert also performs this role. Selecting the best solution tends to be either extremely easy or extremely difficult. However, solutions often cannot be done simultaneously so possible solutions must be prioritized, usually with the help of the user. This may be simplified by using ‘**Remove & Restore’**.

•Step 8: Execution

This is where the solution is attempted. The skill, accuracy, and speed at which this step is completed is dependent on the skill and experience of the person
executing the solution.

Since the execution of this phase and the verification phase might be lengthy, it is worth considering using a **‘Band-Aid’** to allow the customer to continue work or to reduce the impact of the problem until the problem is fully resolved.

- Phase D: Verification (``Verify it'')
  - •Step 9: Craft verification

This is the step where the person that executed Step 7 (Execution) verifies that the actions taken to fix the problem were successful. If the process used to ‘**Reproduce the Problem**’ in Step 4 (Problem Verification) is not recorded properly, or not repeated exactly, the verification will not properly happen. There is potential that the problem still exists, but verification fails to demonstrate this, or the problem may have gone away but the support person does not know this.

If the problem still exists, return to Step 5 (Solution Proposals) or possibly an earlier step. Using ‘**Fresh Perspective’, ‘RTFM – Read The Fine Manual’** and ‘**Watch the Mill Grind’** can help break out of this loop if it seems that all attempts at solutions reach a dead end.

•Step 10: User Verification/Closing.

`Now it is time for the customer to verify the problem has indeed been resolved.`

We would like to suggest an additional phase to Limoncelli's model:

- Phase E: Closing the loop

Step 11: Analysis of the reported problem and its resolution process.
Going over the recordings of the reported problem throughout its entire resolution process helps us warn of mistaken or dangerous actions, where **'There Lay Dragons!'**. This can be done by the support person, by a colleague or by someone assigned for the post-mortem analysis.
The problem might be categorized for statistical purposes, so trends in problem reports can be analyzed.
Hindsight may also help find out how the problem can be identified and isolated faster the next time it occurs.

Step 12: Return feedback
Feedback from software maintenance must continue into product design, development, testing, marketing, training, documentation, **Review Checklist**s, FAQs, support and troubleshooting guides and knowledge bases – **'The Collective Mind'**. The feedback is aimed at preventing the problem from recurring, reducing its severity or impacts, resolving it faster and in looking at preventing similar problems from reaching the customers [8].

Step 13: Design for maintainability
It is always easier to maintain software if the product is designed with making it easy to provide support for it in advance. Patterns such as **'Peepholes & Testpoints'**, **'Documented Assumptions'**, **'All Resources are Finite'** and **'Watchdogs & Sentinels'** help make software more maintainable.

Name

What IS *the Problem?*

Context

A customer calls a support engineer trying to describe a problem. There is a wide gap between a customer describing a problem and a support engineer trying to resolve it. The gap starts at perspective and context and works its way down to the language used.

Problem

The customer presents a complaint but the support engineer may not be able to pinpoint the difficulty.
This is aggravated because people do not usually describe a problem, but rather display their analysis of it. Or even what they perceive as a solution.
A problem statement that is incorrect, incomplete or improperly communicated might mislead the support engineer towards solving a wrong or unnecessary problem.

Forces

Both the customer and the support engineer have their own world models, cultural background, environment and experience, which lead to a gap in problem perception.
The customer may lack knowledge to precisely describe the problem.
Customers tend to describe their analysis of a problem thus leading the support engineer astray.
Support engineer triggers many paths of solutions by keywords and stops listening to the customer's words. Surplus of knowledge might lead the support engineer into cognitive tunnel vision [3] – taking a wrong turn in a chain of assumptions about the problem, and staying there...

...And the truth is out there, somewhere...

Solution

The support engineer will restate his/her understanding of the problem using the simplest language possible. The customer is to comment and correct any discrepancy. This process is iterated until no gaps are found.
The support engineer should focus on facts, such as actions and results.

Resulting Context

Having pinpointed the actual malfunction, the support engineer can now turn to the process of problem resolution using patterns such as **'Lion in the Desert'**, **'Remove & Restore'**, '**RTFM**' and '**Peepholes & Testpoints**' patterns.
Still, there are times the support engineer has to rely on the customer for executing the resolutions process. Re-State the "Obvious" pattern refers to this part of the work.

Known Uses

- A customer calling Help Desk saying “I lost the printer on my computer”. Further investigation reveals that the problem is inability to print from one of the software installed on the computer, requiring installing a patch to that program and not reinstalling the printer driver as it appeared at first.
- Customer: "My computer crashed!"
  Tech Support: "It crashed?"
  Customer: "Yeah, it won't let me play my game."
  Tech Support: "All right, hit Control-Alt-Delete to reboot."
  Customer: "No, it didn't crash - it crashed."
  Tech Support: "Huh?"
  Customer: "I crashed my game. That's what I said before. I crashed my spaceship and now it doesn't work."
  Tech Support: "Click on 'File,' then 'New Game.'"
  Customer: [pause] "Wow! How'd you learn how to do that?"

  http://www.geocities.com/Wellesley/5337/

Name

Re-State the "Obvious"

Context

After the technical support engineer pinpointed the problem, s/he now has another problem: s/he is trying to help an off site customer requesting assistance in resolving a problem. The support engineer is not at the location of the customer so they both rely on verbal communication to describe a technical problem.

Problem

When talking of a known subject people tend to assume knowledge or ignore automatic steps and these may introduce gaps in communication with the customer or even be the very mistake the customer made.
When communicating with the customer, the support engineer may make assumptions about what the customer is describing, and the customer may have assumptions based on what the support engineer talks about. Techno-speak might aggravate the problem, as people who do not understand it, might feel at discomfort to say they do not understand what a "Scuzzy Terminator" is nor what it looks like, when told to check if it is in its place.
And there is always the risk that the support engineer may go to solution before making sure basics are as they should be.

Forces

- The support engineer is not at the location of the customer so they both rely on verbal communication to describe a technical problem.
- Both the customer and the support engineer have their own world models, cultural background, environment and experience, which lead them to a gap in understanding.
- Support knows by heart location of tools and procedures.
- Customer may have different level of knowledge and experience.
- Support tends to expedite well familiar actions and skip important steps.
- The customer may need time to search for tools.
- Support may go to solution before making sure basics are as they should be.
- While speaking of known objects there is a tendency to assume the performance of automatic behavior that may be unknown to the customer.

Solution

Re-State what is "Obvious" to you so you can compare it to what the customer perceives: The support engineer will restate his/her understanding of the situation starting from basics such as wires or files used through spelling of commands to the description of an output. Restating is done using the simplest terms possible – preferably using no jargon, acronyms or technical terms, as much as possible. For example the SCSI terminator might also be described as "that shiny piece of plastic

with a green light on it, which should be firmly connected to the socket labeled ‘SCSI’ on panel number three”.
The customer is to comment and correct any discrepancy.
This process is iterated until no gaps are found.
Re-stating the obvious may be required in every communication with customers.

Resulting Context

By Re-Stating the "obvious" we try to establish a bridge across two (or more) perspectives, so we can get in a more effective way to a clear statement of situation. Having a clear understanding of what needs to be done and what each of the parties sees and does will help guide the customer through the required steps.

Known Uses

- Customer: "My printer isn't printing!"
  Tech: "Is your printer turned on?"
  Customer: "Ummm... oh. [click]"
  http://www.ecis.com/~weasel/support/techsup.html,
- Support engineer in the computers industry will describe the shape of a certain window; its colors and layout, until the customer confirms s/he is looking at the same window containing the same function keys.
- A support engineer verifying, “you have typed the letters xyz before the command” that isn’t working for the customer.

Name

Reproduce the Problem

Context

The support engineer has obtained a statement of the problem from the customer. From now on this problem statement will be the base for the work on resolving this problem.

Problem

How can the support person avoid working on an incorrect, unnecessary or incomplete problem statement?

Forces

- A malfunction may be environment or time dependent.
- Not all problems are known in advance, some may require creating a solution on the fly.
- There are times that only a specific set of steps or events will bring out a problem.
- There could be an issue the designers or creators of the product overlooked.
- For some people it is easier to understand when they see rather then just hear the symptoms.

Solution

Reproduce the problem: repeat the steps reported by the customer, so you receive the same error message or erroneous result. It may be good to create a similar environment to the one the customer has in order to achieve the same results. Alternately you may ask the customer to recreate the problem, as you watch it.

Resulting Context

Reproducing the environment the customer has and following the steps taken by the customer may reveal expected conditions not foreseen by the designers or reveal the point of error. This will allow to either request a solution from the designers (a fix for a bug) or showing the customer where the wrong step was and correcting or teaching the customer the preferred actions.

Known Uses

- In high-tech industry a product developing group will have a laboratory with the product installed on different systems in order to replicate bugs or problems reported.
- Remote maintenance software such as PC Anywhere™ and VNC™ allow a support person to observe the problem experienced by the customer without flying all the extra miles to the problem site.

Name

What Did You Do Before That?

"It worked just fine until yesterday. Today all of a sudden it's malfunctioning."

Context

After understanding what troubles the customer, establishing grounds for work together if needed, and hopefully having seen the malfunction appear; the support engineer wishes to collect clues to the possible reasons that caused the problem.

Problem

The problem statement doesn't provide all the information a support engineer may need. There is value to the knowledge of the chain of events that lead to the appearance of the problem. This can aid in re-producing the problem, and gaining an insight into the causes for the problem can give valuable clues for the solution.

Forces

- Information about the chain of events leading to the appearance of the problem can shed light on the reasons to the malfunction.
- The customer as the one closer to the system and the events may have valuable information.
- The customer doesn't want to appear as the one who caused the problem, especially if s/he did something to it, either related or unrelated to the malfunction.
- The process of questioning may appear judgmental or patronizing if not conducted carefully.

Solution

Ask the customer what were the last events that took place before the malfunction first appeared. Compose your questions carefully not to reflect accusation, so the customer will not get defensive and omit crucial information. The aim it to discover changes done recently, either by the customer, other parties or processes. System Logs, Recent Changes files, Package Installation Logs, system performance archives such as 'sar' in Unix, core dumps, Registry values, can all hold clues that may help asking guiding questions such as: "Was this before or after patch X.Y.Z was loaded?"

Resulting Context

Sometimes the information gained about the events leading to the appearance of the malfunction can give the support engineer important clues towards understanding of the causes to the problem and possible solutions to it.

Known Uses

- A customer complained that his computer doesn't work. A check revealed that the operating system kept crashing. Careful questioning revealed that the customer attempted to install new, incompatible software on the computer just before it stopped working properly.

- Ofra recalls a customer complaining about a notebook PC that cannot be powered up. Following careful questioning it was determined that the customer deleted "unnecessary files" on his boot drive in order to free disk space...
- User: My computer won't work.
  After much discussion on the phone. No reason obtained.
  Tech: Did you do anything to it?
  User: Well, it fell off my desk this morning? Could that be the reason?

  http://www.geocities.com/Wellesley/5337/

Name

Hunt the Lion in a Desert

*A desert is a very big space and it is difficult to find the lion, so how do you hunt the lion in a desert? You draw a line splitting the desert space in two. The lion is either on one side of the line or the other. You cut the half space on the side the lion is again in half. Again the lion will be only on one side of this line. Thus relatively fast you get to a manageable space where it will be easy for you to find and hunt the lion. [**Note:** No lions are harmed during application of this pattern.]*

Context

After having asserted the malfunction, and hopefully attaining an understanding of the situation the customer is facing, it's time to start defining the problem in order to plan a solution. The major preliminary issue at this time is where to look.

Problem

The support engineer is presented a general problem, whose domain is not immediately identifiable. Several inputs and any module can cause the symptoms presented in the suspected system.

Forces

- Initially the scope of problem solution can be very big.
- The presentation of the problem may not be clear.
- Finding where the problem occurs helps concentrate efforts in the correct area.

Solution

Ask questions that will help you target the area where the problem resides, by drawing that imaginary line, and asking questions or inspecting the system to find on which side of the *line* the problem is.
Ideally, you half the problem domain in each iteration, to optimize on the number of iterations needed.
This can be facilitated if the system is designed with '**Peepholes & Testpoints'** in it. Testpoints allow injecting known input into the system at each test point, and observing the processed output in the next peephole, so the problem can be isolated between the first point where the output appears to be corrupt and the last point the input was known to be correct. **'Documented Assumptions'** and **'RTFM'** can help knowing what inputs and outputs should be the proper ones for the system.
The problem is more difficult to isolate when its manifestation is time or environment dependent.

Resulting Context

By confining problem to the smallest possible region of the system, the support engineer minimizes the scope of the search for a solution.

Known Uses

- Support engineer will first try to assert through a series of questions whether the problem belong to hardware or to software.
- A programmer will check through a hierarchy of tests to isolate the faulty code line.
- Compiler writers require that a minimal code segment that reproduces the bug will accompany a bug report, using the fewest steps possible. The bug submitter is therefore required to isolate the minimal subsystems required to reproduce the bug, rather then submit entire modules of specific code.

Name

Remove & Restore

"... The IT group recommends restarting your computer and retrying the failed operation before contacting the help desk. [From an Intranet Web page of a help desk team]"

Context

A technical support engineer is trying to fix a problem but it re-appears at the end of every attempt.

Problem

The support engineer may have an idea where the problem resides but has trouble isolating the core of a problem. Support engineer may also be facing an unstable environment that makes it difficult to fix the stated problem.

Forces

- The engineer cannot always know all components of a system.
- Exact problem isolation and analysis is a lengthy process.
- The customer might not be able to provide details that can lead to problem recreation.
- There are times an unstable environment prevents the support engineer from seeing or working on the problem.

Solution

Remove the suspected part and restore it. (In software terms: uninstall a program and if the problem is not solved reinstall it). Repeat this until you either pinpoint the problematic part or attain a stable situation from which to go on.

Resulting Context

By removing parts the support engineer may reach a stable, controlled environment. This by itself may resolve the problem, or at least may separate which part is causing the problem and isolate it.
This may also provide an opportunity to allow the customer to continue working while the full solution to the problem may be preformed at a later time.

Known Uses

- Support engineer may uninstall a program and if the problem is not solved reinstall it.
- Removing a hardware module and reseating it in its position is used in many hardware troubleshooting schemes.
- A variant to this pattern is interchanging components or interfaces for crosschecking to find out where the fault lies. Such as a friend encountering a problem in connecting a digital camera to a FireWire interface on a PC. The problem was isolated by trying to connect the same camera to another PC with a FireWire

interface that is known to work, and testing the original PC with a digital video camera that was know to work using FireWire.

Name

Band-Aid

"My computer does not work, and I have a plane to catch in three hours!"

Context

There are times a technical support engineer may judge it better to allow the customer to continue work rather then fixing the problem. Either the implementation of the solution requires down time the customer can ill-afford or the solution is elusive and will require more time and tests to be found.

Problem

The support engineer needs time, either to find the exact problem or to implement a complex solution that will take time. The customer however cannot spare the time.

Forces

- The customer wants the problem resolved ASAP.
- The customer has pressures and needs that do not intertwine with those of the support engineer.
- Full problem analysis and resolution may take a long time.
- There are times, such as monthly closing of accounting books and problem domains, such as call processing in a telecommunication system, that ability to continue work is more important then fixing the problem.
- Support engineer need to see the wider scope rather then concentrate on the narrow problem presented.

Solution

Implement Band-Aid solutions, short term or partial solutions that will reduce the severity of the problem or prevent the problem from recurring by bypassing it. Remove or discard data that triggers expensive or severe failures, or help the customer to solve a resulting pressing secondary problem. This way you allow the customer to continue work on those parts of the system that are at higher priority.

Resulting Context

By this you buy time either for yourself to perform more tests in order to locate the exact problem and matching solution, or for your customer to pass his/her critical emergency and then will be available for implementation of your solution.
During the time gained, the support engineer can re-apply **'Hunt the Lion in the Desert'** pattern, or try for a **'Fresh Perspective'** or **'RTFM'** patterns.

Known Uses

- When a file system fills up repeatedly, the support engineer can write a cron job or a script to periodically delete un-required files.

- If a certain data section triggers a problem, the support engineer can change this section of data to prevent the problem from recurring.
- A Car garage that has no spare part, may implement a fix only to enable the customer to get to the nearest big garage where they can replace the malfunctioning part.

Name

Fresh Perspective

"Do you have a moment to look into this problem?!"

Context

A technical support engineer feels s/he got "stuck" – not only without a solution but also with no ideas for further avenues of investigation of the problem.

Problem

The support engineer has exhausted his or her ideas and experience for identifying the problem or of finding a solution to a problem.
Support engineer may even get too frustrated with a problem or the customer to be able to productively search for a solution.

Forces

- There is a limit to personal knowledge and flexibility of thinking, by human nature [3].
- A support engineer can get fixed on a certain perception of a situation and be unable to change direction [3].
- A support engineer may have focused on the wrong area of problem definition or solution.
- A support engineer may have misunderstood or been misled by part of the interaction with the customer.
- A different support engineer may focus on different aspects.
- A different support engineer may have different amount of technical knowledge.
- A different support engineer may ask different questions thus defining a different area to search for solution.
- Frustration can cause a person to lose focus in problem solution.
- Assigning another support engineer to work on the problem takes time and taxes the customer's patience.

Solution

Refer the problem to a colleague, a parallel professional, who will start solving from the beginning thus **gaining a fresh perspective and unbiased analysis**.

Resulting Context

By having another support engineer solving a problem from the beginning, one gains a fresh perspective and unbiased analysis of the problem and maybe a new direction towards a solution. On the other hand, the customer may feel aggravated by being asked the same questions, again, by the new support engineer. Explanation of the reasons for what appears to be starting from scratch to the customer may help reduce this negative impact.

Known Uses

- Having spent the better part of a morning trying to solve a hardware problem, support engineer turned the problem to a colleague (stating the original problem). The colleague re-asked the customer questions regarding the malfunction, thus discovering an action taken by the customer that started the problem. From there the road to solution was clear.
- A new support engineer in a team will turn a problem to a more experienced colleague on the team.
- The '**Cardboard Consultant'** [11] pattern.

Name

Watch the Mill Grind

"1545 Relay #70 Panel F (moth) in relay. First actual case of bug being found"
Naval Surface Warfare Center log entry, September 9, 1947 [9]

Context

The support engineer has tried many tests, asked questions, brought in a colleague and even sat long hours reading the manuals. Still something eludes him/her and there is no solution in sight

Problem

The support engineer has exhausted all personal knowledge and experience, outside human resources, and manuals, has tried to reproduce the problem or had the customer show when the problem appear. Still the support engineer stands at a dead end without an insight into the cause of the problem…

Forces

- People get accustomed to the system they work with and either ignore deviations or remember only major deviations from the routine [2][3].
- Customer may neglect to notice part or parts of the workflow that may indicate a problem [2][3].
- The customer has formed a conceptual model [2] of how the system is supposed to be working internally. The assumptions he makes in this model might not correlate to how the system really works.
- The support engineer usually knows the system from documentation, and may have a lot less time in the field with the system.
- There are times that outside forces or an unforeseen sequence of events influence the system causing malfunctions. These influences might be time dependent or triggered by environmental conditions.
- It may take an outsider view, one that isn't involved in the process, to see a deviation or misbehavior.

Solution

Watch the Mill Grind for a flaw: watch the activity of the malfunctioning system and the activity of the people working on it. Follow the actions and results and look for any deviation from the expected behavior of parts, actions or results. Look for additions or detachments, incidents or activities that the customer has added or created in the specific environment.

Resulting Context

Having sat and watched the activity hopefully gave the support engineer an opportunity to catch a flaw, a deviation overlooked by the customer and even other engineers.

It may even be a deviation judged to be acceptable that will turn out to be the cause of the problem.

Known Uses

- The term “debug” originated with the Harvard Mark II project at the US Naval Surface Warfare Center on 1947 – when apparently random errors showed up in calculations, manual inspection of the hardware for a failed electronic valve or relay revealed that a bug (an actual moth) caused a malfunction. Following that event, each time an error was detected, people asked if the computer was recently de-bugged [9].
- Amir recalls being on a team called to a customer who bought an automated packaging system that was malfunctioning. He and his colleagues spent a couple of days trying to find the reason for the malfunction to no avail, so much that part of the team decided to go back to the manufacturing company to change the design of the packaging system. Amir says he decided to stay behind, found a sitting location that gave him view of the entire process of packaging (involving several automated machines), and recorded step by step the procedures. This allowed him, after some time, to notice a tiny deviation on the expected process that indeed revealed upon inspection a second sensor that was added on site, and triggered by vibration, it disturbed the proper flow cycle by starting a new cycle before the previous ended.
- James Harriot, a veterinarian [10] recalls a case of calves that were displaying symptoms that could only be explained by poisoning, but no harmful substance was discovered. After all avenues of testing, questioning and searching for a cause were exhausted, and even calling a colleague for a **Fresh Perspective** did not help, the colleagues resorted to watching the process of feeding from the start (early hours and all). This revealed a piece of scabbing from the horns that were smeared with a toxic chemical (Antimon) that fell off the horns into the bucket of milk the calf was drinking from.
- Amir recalls an industrial automation project in an orange packing plant that reported a problem in an oblique manner. The customer mentioned while renewing a maintenance contract that the system is “great, but takes time to warm up during the winter”. Observation of the system in field eventually led to discovery of an out of spec photocell that was triggered by a ray of sun through a skylight from 06:00 to 06:30. Replacing the photocell to a different type allowed the plant to start working earlier.

Name

The Collective Mind

"Better go and check this one on the newsgroups"

Context

The support engineer is seeking a solution to a problem, after obtaining the problem statement from the customer. The support engineer has gone through phases B and C, to no avail – the problem is still unresolved.

Problem

No support person can encounter all the potential problems lurking in a system, software, hardware and environment.
Also, how can a support person know what assumptions were used by the people who designed and developed the product?

Forces

- Some malfunctions are rare or happen under unique set of events.
- It is impossible for a single support person to have encountered all the possible problems a system can have.
- As there are many professionals working on similar systems, it is likely at least one of them has met the particular problem and solved it.

Solution

Turn to the collective mind – use troubleshooting diagrams, FAQs, troubleshooting guides, Usenet groups, forums, knowledge bases and solution reservoirs on the Web. Most likely someone ran into this problem before you, and uploaded a solution, to share with colleagues across the world.
Beware of cases where '**There Lay Dragons!**' - it is safer to get independent verification of the proposed solution, by checking for proposed solutions in more then one site, even if the problem description matches the problem exactly.
The better your problem isolation is, the better your chances are of finding a matching solution.

Resulting Context

Hopefully, you found one or more proposals for solutions to the problem. Now you can turn to evaluating the possible solutions before selecting one of them.
In other cases, you might not find an exact solution, but still have more leads and directions to explore, following the search of the collective mind.

Known Uses

- Usenet FAQs ftp://rtfm.mit.edu/
- Microsoft Support web site http://support.microsoft.com/
- http://is-it-true.org/nt/nt2000/hottips.shtml

- The original collective mind - http://www.wikipedia.org/wiki/Borg

Name

RTFM – Read The Fine Manual

"...And there it was, on page 8 of the User's Manual"

Context

The support engineer still cannot resolve the problem. Even **'Brainstorming'** or '**Shouting'** [11] to fellow engineers doesn't help, turning to a colleague for a 'Fresh Perspective' also proved inefficient. There's no avoiding it any more, it's time to hit the books...

Problem

The support engineer has exhausted all personal knowledge and outside human resources but still stands at a dead end without solution to the problem.

Forces

- No person encounters all possible potential problems of their profession.
- Not all systems need all the features a product can provide, and each system tests the product differently then other systems.
- The technical support engineer may feel going to the manual is a personal offense, indicating personal lack of ability of knowledge.

Solution

Read The Fine Manual. When all other resources do not help, take the big manual supplied by the producer of the software and try to find leads and ideas for tests that may lead you to identifying the problem of the solution.
Reading the manual support engineer may learn about requirements, assumptions or constraints unknown before, revealing leads into the problem.

Resulting Context

By Reading The Manual support engineer may have found new leads into the problem, optimistically resolving it or at least leading to more tests that will lead to new possible solutions.

Known Uses

- Online manuals – `man` in Unix™, `Help (F1)` in Windows™.
- Command line help convention in Unix™ commands ("`obscure_two_to_five_letter_command -h`").

Name

There Lay Dragons!

"...Next time, I will make it a habit to use **pwd** *before typing* **rm –rf***..."*

Context

The support engineer is trying to resolve a problem. Browsing through the wealth of information stored in the collection of resolved problems (see **'Lab Notebooks'** [11]) presents a wealth of information about problems. Searching through them using tools ranging from 'grep' to Artificial Intelligence yields a few resolved problems that seem similar. But…

Problem

How can you be sure the steps taken yield the most expedient way to resolve a problem?

Forces

- No single person encounters all possible potential problems of their profession.
- Using previous experience can cut short the time to problem resolution.
- Using previous experience might also mean repeating the mistakes made along the way by the person who handled that problem.
- The support person is trying to bring the problem to closure in the fastest way, and might not have all the time in the world to read through tons of text.

Solution

Once a problem is resolved, go through its **'Lab Notebook'** [11], and add comments on the steps taken and their validity. Make sure that all the unnecessary, unwise, useless and dangerous actions taken are marked as Dragon County.
If any special measures are needed to reduce risk (i.e. fresh backups, safety goggles), these measures should be listed as well.
This way you save your colleagues (and yourself) the embarrassment of making the same needless mistake twice, just by not reading through the entire recording.

Resulting Context

By clearly marking such mistakes upon closure of the problem, you make it easier for people not to make the same mistakes again.

Known Uses

- A support engineer has used by mistake a command that completely erased all the schemas in a database. Another support engineer, a few months later, handled a problem, which showed the same symptoms. Having read through the first few paragraphs of the call log, she executed the same command, only to read two paragraphs below "Regrettably, I should NOT have done so…". Following the second incident, both call logs were modified to include warnings immediately following the action taken, in bold text.

Name

Documented Assumptions

“Minimal system requirements – Pentium II”

Context

Design and development of software is an intellectual process that includes making many assumptions. Assumptions are constantly made about topics such as the operating environment, customer training and knowledge, input external to the developed system, values of parameters allowed in APIs. But…

Problem

The people who designed and developed the product had to use a series of assumptions and common agreements. These assumptions may create constrictions that influence the way the product works. In some cases the problem is caused by the real system environment departing from these assumptions.

Forces

- Knowing the assumptions made may expose which of the assumptions is inconsistent with the conditions that trigger the problem in the reported system.
- Documenting assumptions post-factum is a lengthy process, and usually ineffective.
- Documenting all assumptions is time consuming and requires personal discipline.

Solution

Document all assumptions made during the entire development process, as they are made. Make this documentation available to support people.
Special focus is required for assumptions about input parameters, availability of resources (see ‘**ALL Resources Are Finite**’) and error behavior such as assertions, error/exit codes and exceptions.

Resulting Context

By browsing through the assumptions, a support person can try to compare them with the operating conditions in the reported system. Once a deviation from an assumed condition is found, it should be checked against the possibility that this deviation might be the cause of the problem. Overuse of documentation may lead to "trapdoor" documentation – where no one bothers updating documentation, because there is so much of it, and updating both source code & documentation is considered too much of a burden. In-lining documentation as comments into source code, and using tools such JavaDoc & C-Doc facilitate keeping a single location for updates.

Known Uses

- Checking for deviations from the dreaded “system requirements” – such as supported operating system version, minimal memory requirements, disk space, operating temperature, voltage, current and other environmental factors are usually placed at the top of troubleshooting guides for support engineers. This helps prevent looking for more complex problems, when the problem might be the lack of electricity in the office, wrong voltage or frequency, an incompatible or untested operating system version or an input that no one expected.

- Assertions and exceptions help catching cases where assumptions are violated, in some coding methodologies.
- Minimum Requirements displayed on the boxes of PC based games. The following is from the box of Lucas Arts™ "The curse of Monkey Island":

  Computer: 100% Windows 95 DirectX-compatible computer required.
  Graphics Card: PCI graphics card required.
  CPU: Pentium 90 or faster required.
  Memory: 16MB RAM required.
  CD-ROM: Quad-speed or faster CD-ROM drive required.
  Sound Card: 100% Windows 95-compatible 16-bit sound card required.
  DirectX: Microsoft™ DirectX 5 is included on this CD and must e installed prior to playing the game.
  Note: Your system may require the "latest" Windows 95 drivers for your particular hardware.
  Installation: Requires at least 1.2MB free hard drive space. An additional 20MB recommended for multiple save games.

Name

Review Checklist

"Everybody thought somebody would do it,
But eventually, nobody did what anybody could have done" [Anon.]

Context

A problem has been successfully resolved. Its root causes were investigated and analyzed.

Problem

Design groups encounter many problems during product development and deployment. Many times the means to prevent these problems or reduce their impact are forgotten by the time we get again to the review phase. Moreover, the organization aspires to standardize the questions checked during the review.
At the review itself, team members tend to forget the criteria, ambiguities and past lessons – because time has passed.
Preventive practices and culture gained by experience tend to be lost when teams change their staffing.

Forces

- The impact of errors diminishes as time goes by. People tend to forget not only the error, but also the means to prevent them from recurring or reduce their impacts.
- A chore without an owner might be ignored, forgotten or poorly performed.
- The Piranha Effect – during reviews, people tend to focus their attention on a small area of the work item where a flaw was found. This prevents the participants from exploring the entire work item for more flaws and for flaws of other categories
- Too many criteria for review intimidate and go way over the abilities of most people to perform them.
- Developers align their work towards meeting the standards required of them.

Solution

Throughout the project life cycle, set a person to maintain review checklists. This person will get suggestions for additions to the review checklist for future reviews. This might be done by means such as email, suggestions box, corridor talk or Wiki.
Whenever collective wisdom is enriched by investigating a new problem or by finding a new bug or flaw – the means to prevent this problem from recurring should be added to the review checklist.
Each item in the review checklist should include a recommendation, the reasoning behind it and preferably an example or reference to the problem that triggered this item in the checklist. This helps in maintaining the viability of the list over time.
The longer the checklist is - the fewer are the chances it will be used.
Tools may be used to reduce the amount of manual work done in preparation for the review, and the frustration associated with it.

Resulting Context

By preserving collective wisdom and experience, people who set out to perform a task can learn from other people's mistakes (and from their own mistakes, given sufficient time or denial).
Collecting and documenting this support standards and cultural climate that strive to prevent errors rather then merely testing to detect them.
Keeping an eye on the checklist while preparing work items and during the review helps preventing people from focusing most of their attention on a small number of issues or limiting the scope of the review.

Known Uses

- Automated tools for detecting potential problems in code, such as *lint, Purify*™ or compilation with a high level of warning provide means to reduce the amount of manual work needed to detect problems.
- Amir Raveh has added the use of review checklists to the software development process in teams and projects he led or participated in. Other projects and groups in Motorola have adopted this practice.
- Coding guidelines, such as IBM [12], Elemtel [13], Sun Java coding style [14], C++ programming guidelines [15], and Usenix papers [16] present elaborate lists of rules aimed at reducing errors.

Name

ALL Resources are Finite

Context

You are designing a new system or adding a new feature to an existing product.

Problem

Each software entity – at any hierarchy level - a class, a software module, a process, a system – makes use of resources.
The resources can be files, handles, sockets, message queues, locks, threads, communication bandwidth, memory footstamp, current consumption in embedded devices, CPU utilization, I/O utilization, storage (long & short term), database cursors.

Some of these resources seem to developers to be infinite – the most common (and notorious) are CPU, disk space and dynamic memory storage space.
Given enough slack, a software entity can become so bloated and resource greedy that it will simply spiral itself, its neighbors and even the entire system down.

In other cases, the usage profile of resources becomes a major problem when the system needs to be scaled up or down – only then you discover that one of these resources becomes a bottleneck when the system needs to support more concurrent users, drive higher data rates, or use a smaller battery.

How can you raise the awareness to this potential problem and reduce the probability of it arising, without going through unnecessary optimization of every single piece of code written?

Forces

- You want to keep the development schedule as short as possible.
- You want to avoid the code from becoming too complex.
- You want to optimize only the sections that are critical, if and when needed.
- Preparing and checking more resources takes more time and effort at the programming stage.
- Adding too many checks on resource availability compounds the program.
- Adding more resources drives up system cost, and sometimes maintainability costs.

Solution

View all resources as finite.
Start with a project-wide resource checklist, based on project goals, problem domain and previous experience.
Get everyone involved to identify and count the resources they use (or plan to use) in their context, and maintain this count throughout the development process.

Make sure each resource is shown with a quantifier (how much/many are needed) and a multiplier (what is it multiplied by – active user sessions, number of concurrent processes, etc.) to check future growth is possible.
Decisions about which resources are monitored and which would not be monitored, as well as code behavior if a resource is not available at run-time, should be consciously made, documented and published. This process is repeated at each iteration of the development life cycle, and whenever a major change is considered.
The project-wide checklist is kept updated to reflect the changes, and is publicly available to all project members.

Resulting Context

By developing and maintaining this awareness, we facilitate a clear, common understanding of the resource problem, as the first step towards its solution.
Overdose effects are running a very long, complex list of resources to be monitored or complex code that attempts to check for every possible contingency.
Some developers might get carried over into optimizing the use of each and every resource, investing uncalled for effort on the legendary 80% that will never need optimizing.

The following patterns are related:
- **Memory budget** (Weir & Noble) [17] gives an example how the use of a resource can be planned.
- **The Resource Miser** [18] shows how to go on enforcing resource allocation, and how changes in resource requirements can be moderated.
- **Peepholes & Testpoints** shows a way to monitor resource utilization and verify system performance.
- **Eager/Lazy Acquisition, Leasing** and **Pooling** are design strategies that facilitate resolving resource problems.

Known Uses

Memory Budget and **Make the User Worry** by Weir and Noble [17] are patterns which implement specific uses of this pattern, for memory limited systems.

Peepholes & Test points

Context

Development of a performance constrained system, from the design phase onwards.

Problem

You have a system where the use of an off-the-shelf solution (such as a profiler, an OS accounting system) to measure resource utilization is ineffective or unacceptably deteriorate its performance.
This is further aggravated after unit testing of software is completed, because in many cases the system is running through expanding circles of integration testing. But, some of the sources feeding the current subsystem, or some of the sinks it interacts with are not ready, yet…
How can you peek under the hood, find out resource utilization, verify performance levels, and keep this as an ongoing option to go on doing this throughout product development, deployment and even in-field?

Forces

- Introducing a measurement or troubleshooting module (such as debug code or printing traces) into a system modifies its performance.
-Standard debugging techniques, such as single-steps and breakpoints are not applicable for some problem domains, such as real-time systems, and for some development phases, such as commercial deployment.
- Measuring & streamlining performance is an ongoing process, not a one shot effort.
- Some resources are inaccessible to external probing by their nature.
- Integration is an ongoing process, where some components arrive later then others, and each new arrival may upset an already existing balance.

Solution

Build execution and performance tracing into the system from design onwards –as mechanical designers place peepholes into potential failure points, and electronics designers put testpoints, which allow monitoring the progress of a known input signal throughout the system.
Such mechanisms can be found in operating system kernels – as counters for events and system call gates, in telecommunication systems as performance statistics counters, and in profiling tools.

Resulting Context

Measurement and tracing is always there, and therefore using it should bear little or no impact on the observed system, since these mechanisms are part of it.
The ability to inject known input into a testpoint, and check how it gets processed further downstream facilitates checking resource utilization, verifying correctness, and

in some cases simulating overload and exception behavior. This can be used in a system in the field to check for correct behavior, or in attempting to recreate a problem scenario.
An overdose may lead to over tasking the observed system, by using too many resources for fine grain monitoring. Most typical examples are trace files overflowing disk space, or debug messages overloading the observed system.
The probes that were designed and introduced into the system might miss the point where they are needed, being too far and few in between, or just because Murphy's Law is still applicable.
This test setup may cause system degradation or outage, as the injected input may be erroneous or cause system misbehavior, as it might be exempt from system consistency checks. The additional code and data is not immune from programming errors…

Watchdogs & Sentinels

Context

Development of a performance constrained system, from the design phase onwards.

Problem

It is impossible for software to foresee any possible fault it may run into. No matter how well the design, how well coding and quality assurance filter out bugs, there is always that “one that got away” or that combination of events that “should not have happened”.
And as Murphy’s Law would have it, the fault will be where we least expect it, and its effects will be the worst possible one.
How can we detect those unforeseen failures in performance critical systems?

Forces

- Continuously checking for errors and error conditions consumes development resources, as well as runtime resources.
- Failure creeps in where we least expect it, yet we need to detect the unexpected, so we can take action to remedy it and/or determine its cause.

Solution

Use watchdogs and sentinels to detect possible failures, and respond accordingly.
A watchdog is a timer that is started when an activity is initiated, and will let the system know if the activity has not completed by the time constraint. It will be assumed that there is an undetected failure, and proper action can be taken.
A sentinel is a mechanism to verify program data is intact. It relies on known rules as to how data can be verified to be valid, data is checked for validity (periodically or prior to access), and when invalid data is encountered, the system can take action to correct the failure.

Resulting Context

The system can include mechanisms to respond to failures, once a sentinel or a watchdog detects them. Examples of such mechanisms are traffic lights that go to blinking yellow mode if a watchdog timer expires, taking out of service a redundant module if it fails a majority test in astronautic systems.
If settings are oversensitive, false alarms will be generated, affecting system availability and reliability. Using a **Leaky Bucket Counter** to debounce transient alarms in cases where the results of a false alarm are expensive to the system can mitigate this.
An avalanche of error messages might occur if overused or poorly designed.

Acknowledgements

This paper started out with a single pattern (**‘Re-State the "Obvious"’**), written by Amir Raveh in a pattern writing workshop delivered by Jim Coplien and Christa Schwanninger in Tel-Aviv, 1998. The pattern traveled to EuroPLoP 1999, where Christa Schwanninger coached Amir into seeing how this single pattern leads to more patterns, and these form a language.
The outline of the language kept growing as a draft in mind (and on the Palm), until Ofra helped in pushing it from a vision into a full-blown set of interrelated patterns.
We would like to thank our customers and colleagues who have contributed from their experience and efforts towards enriching us with their views and cultures.
We would like to thank our shepherds – Jorge Ortega, Arno Haase and Neil B. Harrison for their efforts and suggestions towards making our papers far better than they have started.

References

1 Doing Hard Time, Bruce Powel Douglass, Addison Wesley, 1999, pp 98-99 for software safety hazards such as Therac-25, Patriot missiles, Aegis tracking system and other documented events.
2 The design of everyday things, Donald A. Norman, Addison Wesley, 1990, on the topic of self-blaming when people encounter problems in computer-based systems.
3 Things that make us smart, Donald A. Norman, Addison Wesley, 1993, pp 131-138, on human cognition, error and tunnel vision.
4 Customer support horror stories
Computer Stupidities, http://www.rinkworks.com/stupid/
IT Doom Dome, http://www.geocities.com/Wellesley/5337/
Tech's Support, http://www.ecis.com/~weasel/support/techsup.html
5 www.snopes.com -
Vanilla vapor lock (http://www.snopes.com/autos/techno/icecream.asp),
Word Imperfect (http://www.snopes.com/humor/business/wordperf.htm).
6 Deconstructing User Requests and the Nine Step Model, Thomas A. Limoncelli, Usenix Association, Proceedings of LISA '99: 13th Systems Administration Conference.
7 Patterns for Logging Diagnostic Messages, Neil B. Harrison, PloP 1996.
8 Key Practices of the Capability Maturity Model SM ,Version 1.1, CMU/SEI-93-TR-25, Software Engineering Institute, February 1993, pp L5-1 – Defect Prevention Key Process Area.
9 Annals of the history of computing, Vol. 3 (July 1981), pp. 285-286
http://wombat.doc.ic.ac.uk/foldoc.foldoc.cgi?bug
10 All Things Wise And Wonderful, James Harriot, 1976.
11 Process Patterns for Personal Practice, Charles Weir & James Noble, Proceedings of EuroPLoP 1999, Universitaetsverlag Konstanz.
12 IBM ICU Coding guidelines,
http://oss.software.ibm.com/icu/userguide/conventions.html
13 Elemtel C++ coding rules, http://www.chris-lott.org/resources/cstyle/Ellemtel-rules-mm.html
14 Code Conventions for the Java™ Programming Language,
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
15 C++ Programming Guidelines, Plum, Thomas and Saks, Dan, Plum Hall, 1991
16 Can't Happen or /* NOTREACHED */ or Real Programs Dump Core, Ian Darwin and Geoff Collyer, Dallas USENIX Conference, January 21 1985
17 Small Memory Software: Patterns for Systems with Limited Memory, Weir & Noble, Addison-Wesley, 2000

18 Performance Patterns, Amir Raveh, Proceedings of EuroPLoP 2002