People Make Mistakes – A Survey of Common Causes of Software Defects

Alex Elentukh(✉)

Metropolitan College of Boston University, 1010 Commonwealth Ave, Boston, MA 02215, USA
elentukh@bu.edu

Abstract. Software projects spend a significant effort on fixing defects. In fact, some 'successful' companies are unable to develop new features, since they devote all resources on supporting the multitude of customers and addressing the flow of incoming issues. This justifies the adoption of a well-defined testing and bug fixing process that is tailored for the organizational specifics. The effectiveness of such process drives the effectiveness of the whole organization. Our paper surveys the root causes (RCA/ARCA - Root Cause Analysis and ODC - Orthogonal Defect Classification) of common software defects and makes recommendations based on actual examples. In a context of a public university, the key recommendation is to assure that various angles of a defect prevention and bug fixing processes are explored, as part of a standard computer science curriculum.

Keywords: K.3.2 computer & information science education · software engineering · taxonomy of defects · root cause analysis

1 Introduction

Complexities of finding and fixing software issues are commonly overlooked [1], which results in additional residual defects drifting around a code base [2]. Consider these steps,

Step 1. A Reviewer or Tester finds a defect, then
Step 2. Communicates it to an Author, who
Step 3. Fixes the defect.

If we examine these three steps in detail, an immediate discovery will be that the process is vastly oversimplified. The assumption that an Author successfully fixes only those defects that a Tester reports is distant from reality. Both Author and Reviewer (for a virtue of being human) do err - every step of the way. In fact, a common process usually includes several additional sidekicks, as follows,

– Reviewer imagines an issue, which is not a real defect;
– Reviewer reports a wrong issue, not the one which was discovered;
– Author fixes a different defect, which has never been reported;

T. Zlateva and G. Tuparov (Eds.): CSECS 2023, LNICST 514, pp. 117–133, 2023.
https://doi.org/10.1007/978-3-031-44668-9_9

– And finally (to the surprise of the Data Analyst, (a side-line observer) an undoubtedly improved software emerges from the review and test process.

Pictorial on Fig. 1 reflects the expanded process.

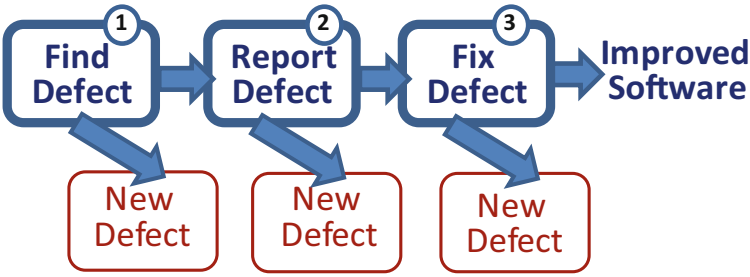

Fig. 1. Complexities of finding and fixing software defects

Question arises as to why the resulting software is still cleaner than the original software. Apparently, in most cases, the number of defects that are being fixed - is greater than the number of new defects that are being introduced.

2 An Example of a Book Review

To illustrate these notions, we consider six actual cases, that are well-known in the industry.

CASE # 1. Pictorial on Fig. 2 shows the front cover of the book published in September 2019. There is an initial copy (or the so-called 'Pre-Publication Draft' marked with a yellow stamp) [3] and Final Version [4] (that is being offered through Amazon, translated into twenty languages and sold over a million copies worldwide). We shall consider an example from initial copy, then the corresponding comment which we provided toward this example, and then the final result. Over three hundred reviewers (including author of this paper) examined the draft and submitted their comments.

Page 128 from Prepublication Draft is shown on Fig. 3.

Page 129 from Prepublication Draft is shown on Fig. 4.

One does not need to be a particularly thorough reviewer to notice that Fig. 9–1 and the top part of Fig. 9–2 – are redundant. However, that was not the comment provided. Here is the exact comment which was communicated, "*Fig. 9–2 at the line 2751 implies that a product could be released, if 'fix rate' is approaching the 'discovery rate'. This is really incorrect. If the 'discovery rate' does not come down, a product should not be shipped, regardless of a 'fix rate'.*"

We should bring about the importance of this comment. Since it attempts to reflect and to correct a common misinterpretation of a real-life situation, "as long as we are able to address *quickly* whatever is coming our way, the product remains in a releasable state". Nothing can be further from the truth. Even though the 'fixing curve' is shifting to the left, the 'discovery rate' will always remain an independent measure, which stands on its own feet; it should never be overlooked.

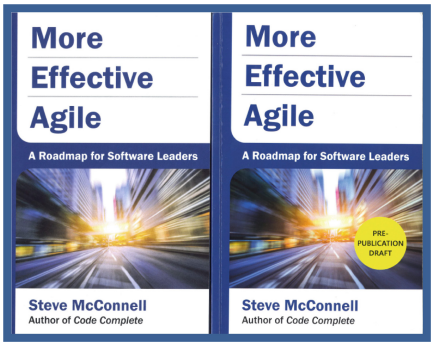

Fig. 2. The front cover of the book along with its prepublication draft

Consider the Fig. 5 of the Final Version, reflecting the same pages as shown at Fig. 3 and Fig. 4 of the Preproduction Draft. Apparently, the redundant picture has been eliminated; but the comment offered toward this section – was *not* taken.

Why did Reviewer miss the issue with redundant picture? Because Reviewer was preoccupied with a different notion, which is covered in great length in the course they are currently teaching. Why didn't Author take the comment offered by Reviewer? Because Author was preoccupied with the notion of Latent Defects and how to express this notion, assuming the comment does not relate to it. The good news is (as predicted at Fig. 1) that the final version is immensely better than the initial draft.

The biggest surprise of our analysis is that Peer Reviews were missed all together. Reviewer missed it. Author missed it. Writing a comprehensive book that enumerates the key engineering practices and omitting Peer Reviews – seems to be improbable. Although the fact is – Peer Reviews are not covered in the book. One can only speculate about the circumstances of this omission. Particularly considering that Author is a well-recognized expert in Peer Reviews, who went into a great length to publish the book's draft and to air this draft among several hundred reviewers.

What can be a more fitting case study of root causes - but rationing about issues with a book that delves into semantics of software defects. It is worth bringing about the related statistics implied in [5, 6],

- During a peer review, up to twenty percent of all defects are discovered by the Author, due to the stimulation provided by peers.
- There are multiple residue defects that remain undetected, long after a product is shipped.

Such statistics supports the notion that in real scenarios, defects arrive from most unexpected directions. In other words, "even a small ant can bring a large message". Which is opposite to the assumption that defects are exclusively found and reported by professional testers/reviewers and these are the only defects that are being fixed.

CHAPTER NINE

In contrast with the defect insertion, defect detection and removal is not a function of general effort. It is a function of a specific kind of effort: quality assurance (QA) activities.

As Figure 9-1 illustrates, on many projects defect detection and removal significantly lags defect insertion. This is problematic because the area between the two lines represents latent defects—defects that have been inserted into the software but haven't been detected and removed yet. Each of those defects represents extra bug-fixing work that is rarely "on plan." Each of those defects represents work that will unpredictably extend the budget, extend the schedule, and, in general, disrupt the project.

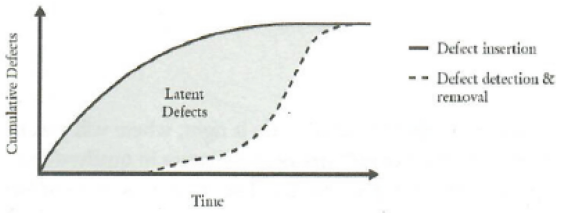

Figure 9-1

The gap between the cumulative defect-insertion line and the defect-detection-and-removal line represents latent defects.

Well-run projects attempt to minimize the gap between defect insertion and defect detection, as shown in Figure 9-2.

128

Fig. 3. Example from Preproduction Draft

3 Flukes and Common

Suppose you are adopting a Defect Prevention process in your organization. As a first step, you need to distinguish common defects and then to assure they are addressed ahead of all other defects.

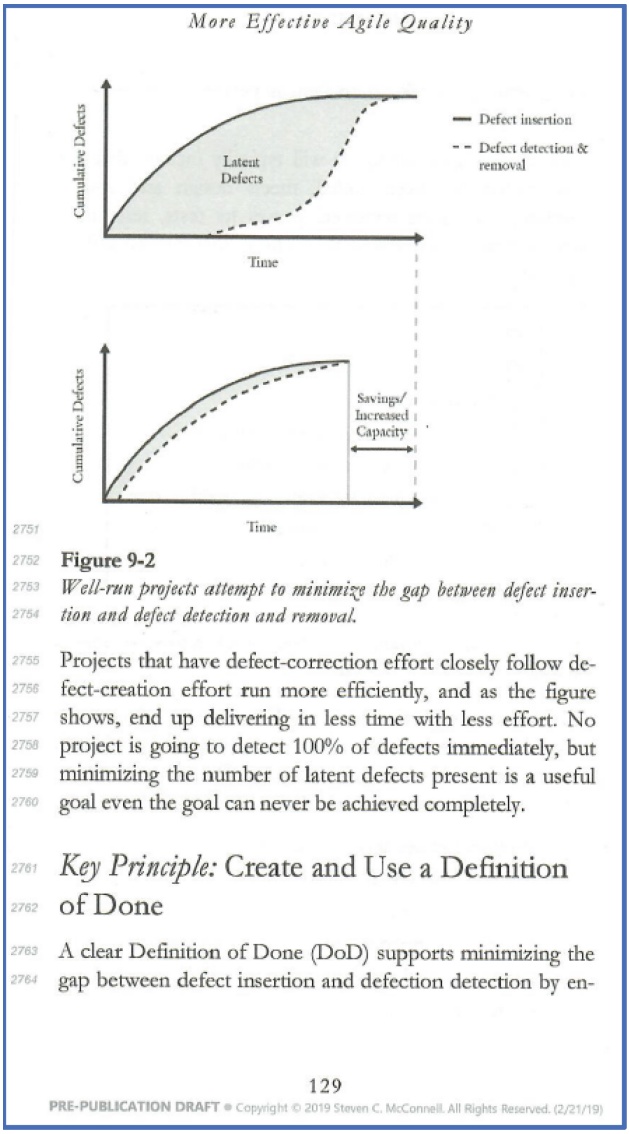

Fig. 4. Example from Preproduction Draft

Consider the following checklist that separates 'common' from other defects, let's call such other defects 'flukes'.

- A defect that manifests itself on ten customers' sites is a 'common' defect
- A defect that causes repeated failure on a single customer site, every ten hours is a 'common' defect
- If ten different types of intermittent failures have the same root-cause, then such a root-cause corresponds to a 'common' defect.

CHAPTER ELEVEN

As the top part of Figure 11-1 illustrates, defect detection and removal significantly lags defect insertion on many projects. This is problematic because the area between the two lines represents latent defects—defects that have been inserted into the software but haven't been detected and removed. Each of those defects represents extra bug-fixing work that is rarely "on plan." Each of those defects represents work that will unpredictably extend the budget, extend the schedule, and, in general, disrupt the project.

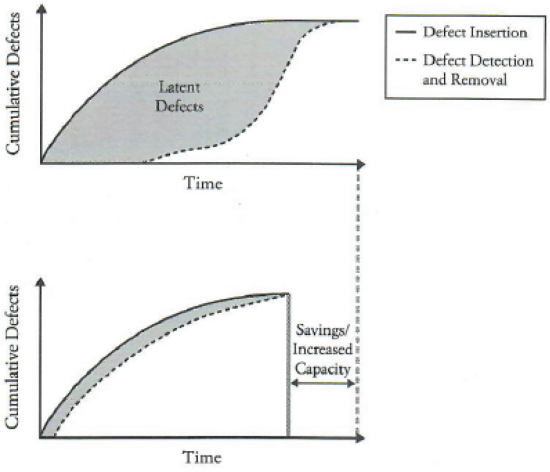

Figure 11-1
The gap between the cumulative defect-insertion line and the defect detection-and-removal line represents latent defects.

Well-run projects minimize the gap between defect insertion and defect detection, as shown in the bottom part of the figure.

156

Fig. 5. Example of Final Version with Comments Addressed

– If ten software engineers, while working independently from each other, inserted the same defect in different parts of code base, such defect is markedly a 'common'.

The purpose of this checklist is to show various angles of common defects. Each of the four points of the checklist can potentially have a corresponding root-cause, as follows,

– User Interface is inconsistent
– Reliability issue, memory leaks
– Design flaw causing a race condition
– Issue with coding standard or operational procedure

Mature organizations have a distinctly different view of software defects, if compared with such a view of immature organizations. For an immature organization, most defects are 'flukes', since their root-causes are hidden. There is an over-abundant flow of defects, and it is impractical to fix all of them, let alone to analyze them. This is contrary to a mature organization, where defects are carefully analyzed and root-causes are eliminated. In other words, mature organizations do not fix defects, they fix root-causes.

The Software Engineering Institute Capability Maturity Model (SEI CMM) [7] includes the process area called Defect Prevention, strategically positioned at the highest Level 5. This process area covers steps for a meticulous investigation of root causes one-defect-at-a-time. It is inappropriate to attempt adopting such process at a lower maturity level, since it would be as 'attempting to drink from a fire hose'.

The ISO 9001:20015 [8], the most generally recognized improvement standard in the world, has one of its six mandatory procedures dedicated to 'preventive action'. The importance of this procedure is emphasized in [9] as an organization without such a procedure documented and used is bound to fail the ISO audit.

4 Taxonomy of Defects as Opportunities

Further we establish several directions toward a systematic analysis of root causes of software defects. It seems counterintuitive to interpret the occurrences of defects as something positive, although this is the only way to focus organizational efforts and to make the best of the situation.

a) Examination of a *repository* of famous defects 'that brought the house down' and 'caused a panic on a street'– should be an imperative part of any training program. Such repository can be a simple extension of a company-wide bug database. Newly-hired engineers should be introduced to both sides of the coin, first, how to do it, and second, an equally important, how *not* to do it. To continue with this concept further, we could not forget the vivid note from a developer, "please do not highlight my defects in the repository; they make me look as an idiot". In fact, overcoming a defensive mentality, creating a safe work place by decriminalizing defects - are the first steps in adopting the repository and administering a company-wide training based on such repository.
b) A comprehensive *regression* suite that is executed all the time; it keeps producing various failures. It also produces a constant flow of opportunities for a team to learn and to improve. To establish such 24X7 regression is easier said than done, since it requires a never-ending support process to address causes of defects.
c) Defects defy logic. They can appear from anywhere. If defects would be logical, it would be very easy to prevent them. A so-called 'fat-finger problem' can be quite persistent and difficult to eliminate, taking into an account its random nature. Chaos Monkey used so successfully by network providers, e.g. Netflix, is the clear proof

that *random testing* can be more effective than following some predefined sequence of quasi-logical test steps.

5 Orthogonal Defect Classification

Linking defects with their causes can be an arduous task. Correlation between faults and failures is not trivial, [10], as it depends on multiple factors. Orthogonal Defect Classification (ODC) [11] offers a methodology to redirect the flow of defects into several predefined buckets. Establishing consistent and independent categories is the prudent step in delving into root causes.

It should be noted that dwelling on RCA (Root Cause Analysis) can misguidedly feed its own purpose. Creating a clumsy and bureaucratic RCA process will ultimately deviate from the original goal of avoiding real customer issues. This brings about the ARCA (Lightweight Root Cause Analysis) [10].

It is crucially important to adopt an in-process technique to systematically avoid defects. Still, establishing a *balance* between correction and avoidance – elevates an organization into a steady state and enables it to advance uniformly. One should not give unjust preference to correction or to avoidance. Organizational maturity and business specifics - shall drive this balance. Ram Chillarege, et al. writes in [11] about the gap between statistical defect modeling and qualitative casual analysis. Apparently, such concepts as 'defects semantics' and 'process signature' – can only apply to a context of a mature organization.

Review of voluminous literature on RCA - brings about some interesting research. Andy Ho et al. in [12] describes a linguistic study of some 200,000 problem reports. This inevitably leads to a common language, a lingua franka, an organizational template of how defects should be recorded. James Reason in [13] bridges the disciplinary gulf between psychological theory and reliability engineering. Apparently, mapping such concepts as 'slip' and 'over-confidence' into the parallel universe of coding standards – can produce some invaluable actions.

6 Matching Defect Types with Test Types

Any measurement program, as well as any defect prevention program – are based on an established classification of software defects. Such classification cuts across multiple phases of software process. Ram Chillarege, et al. in [11] speaks volumes about the critical role of ODC.

The common typology of tests includes the following,

- *Usability.* How quickly can a naïve user learn the system?
- *Functional.* Are there any stated requirements that are uncovered by tests?
- *Regression.* Are there any new defects introduced by a software update?
- *Reliability.*
- *Overload.* Is the system exits gracefully from a prohibited state?
- *Performance.* Does response time correspond to the spec?

Each test type has a specific goal. For example, Reliability Test exposes the time-dependent aspect of a system behavior. Hitting a UI button once, could suffice for a Functional Test. Hitting the same button, a thousand times and verifying that memory pool does not leak, constitutes a Reliability Test.

Figure 6 illustrates the classification of defects, that is matched with their corresponding classification of tests.

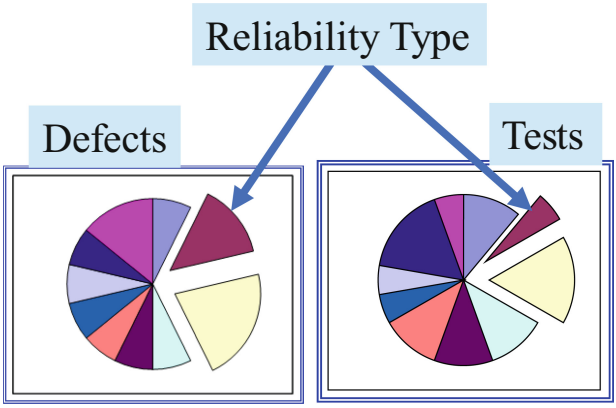

Fig. 6. Matching defect types with test types

Suppose you experience multiple reliability defects. In simplified terms, your system does not stay up and fails frequently. The first direction in investigating the issue is to examine the corresponding Reliability Tests. Looking at a Functional Tests would be a misdirection. Apparently such an investigation is only possible if failures in a defect database, as well as the tests in Quality Center – both have a consistent typology.

Test Types correspond to various *angles* from which a product is examined. Multiple angles (test types) scrutinize the *same* product. The degree of completeness of test coverage is assured by the comprehensive set of test types. Gaps in test types remind a QA Engineer about some tests that could have been missed. Additionally, a great deal of confusion is avoided by prioritizing and focusing the test effort on few selected test types, making sure they do not overlap.

Should note from our experience that Test Management, as an organizational practice, is more mature than Defect Management. An auditor walking into a door is bound to ask about a bug database first. If engineers are not aware of their major defects, this is indicative of some profound organizational issue. On another hand, establishing a consistent Test Management with all its standard templates and training is a significant undertaking. Therefore, when adopting a comprehensive Root Cause Analysis Program, it is logical to start with Defect Management.

7 Integrated Approach Succeed

Here we shall build on the conclusion from the previous section VI. In order to optimize the finding and fixing of software defects, it is prudent to match defect modes with test types. At Fig. 7, Information System Division of NASA, [14] reports the distribution of defects that have been found through a variety of techniques.

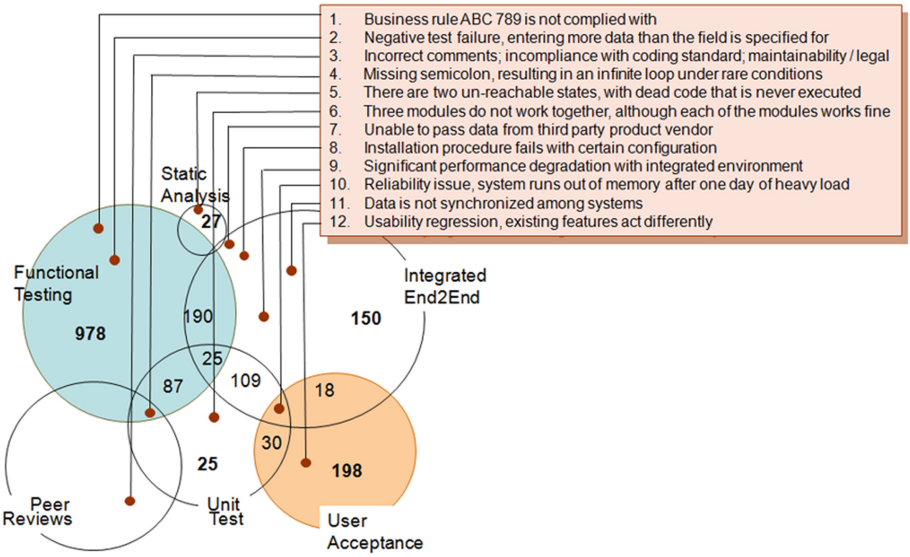

Fig. 7. Integrated Approach to Finding Defects

Here are several general and specific observations.

- There are multiple defects that could only be revealed through a peer review.
- It is overoptimistic to expect finding a reliability issue during a unit test.
- It is far too late to find defects at User Acceptance. One should conduct a detailed root-cause analysis of such defects, so in future to be able identifying similar issues through a different means
- A functional defect found in User Acceptance – should have been found earlier during the functional test.
- Installation defect found in User Acceptance – should have been found by a technical writer during review of the user doc
- Maintainability defect found in Production – should have been found during a code review

In summary, looking at the distribution of defects at Fig. 7, we can arrive at the following conclusions. On one hand, defect modes overlap, as the same defect could be found using several techniques. On another hand, certain techniques are unable to detect certain defect types. For example, a functional test is unable to detect a maintainability defect. Hence, only the *integrated* approach can assure an adequate coverage and succeed in revealing a reasonable number of defects.

8 Defect Cost

When enacting a defect prevention program, many companies prioritize costly failures. [15]. This triggers the whole new set of techniques to monetize the prevention and to convert it into a money-making machine under the umbrella of Cost Avoidance.

Everyone feels in his/her heart that it is cheaper to fix defects early. Question is, "How much cheaper?" In fact, when starting to deal with significant amounts, one should prepare for a clear response.

We shall consider the following example to illustrate the above stated concept. Suppose we deal with an environment of daily software inspections that reveal three major defects per inspection.

Table 1. Calculation of Yearly Expense on Inspections

3	major defects per inspection
1	inspections per day
$200	burden rate
30	effort person/hours per inspection
260	working days in a year
780	major defects in a year
$1,560,000	yearly expense on inspections

Table 1 reflects the scenario of a mid-sized company with a hundred developers. Apparently, the cost of inspections should not be underestimated. It is also a strong recommendation to maintain these numbers for *any* organizational size.

Table 2. Yearly Benefit from Inspections

Cost of a Defect	Yearly Benefit
$200	$ (1,418,400)
$800	$ (993,600)
$5,000	$ 1,980,000
$8,000	$ 4,104,000
$20,000	$ 12,600,000
$50,000	$ 33,840,000

Table 2 shows the calculation of Yearly Benefit as a function of the Cost of a Defect. Table 2 starts with an uncharacteristically-small amount of $200. No company in the world spends so little on fixing a defect in the field. Large companies with millions of live customers - estimate the cost of fixing defects in billions. In fact, looking at the

chart at Fig. 8, one can see the benefit from inspections is growing exponentially, after the cost of a defect rolls over a thousand dollars.

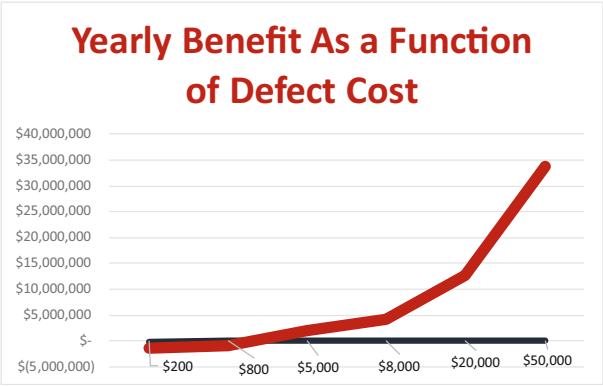

Fig. 8. Yearly benefit from inspections as a function of Defect Cost

Philip Crosby in his classic work [16] titled, "Quality Is Free" establishes that, "systematic drive for quality pays for itself."

Herb Krasner, in [17] writes, "The cost of finding and fixing defects is the largest single expense element in the software lifecycle" estimated at the staggering $1.56 trillion for 2020 in US.

An adoption of a consistent inspection program across a large organization always meets with a natural human resistance. The most common response that a Process Champion hears is that inspections consume too much time. This is contrary to reality (supported by our calculations) – inspections do save time.

9 Holistic Approach

To adopt a consistent practice across an organization, one needs to demonstrate the importance of *every* part of a software process. One way to accomplish this task is to keep performing an on-going Root Cause Analysis and to retain in organizational memory a series of catastrophic failures triggered by skipping each part of the process. In this scenario, RCA is used as a powerful process-adoption tool. To this end, consider the following massively egregious failures. [18, 19].

CASE # 2. Here are quotes from the newspaper covering US Congress investigation into collapse of Health Care Site.

- "*Private contractors in charge of building the federal online health insurance marketplace testified Thursday that the administration went ahead with Oct. 1 launch of HealthCare.gov despite insufficient testing*"
- *"This system just wasn't tested enough," Julie Bataille, director of CMS's office of communications, acknowledged to reporters.*

From reading these quotes, we can attempt to select what most likely went wrong with the site. As a daily event, millions of people jammed into site and eventually brought it down within the course of several morning hours.

a) Incomplete test coverage of original requirements
b) Skill level of test personnel is inadequate
c) Traceability of requirements and tests was not explicitly documented
d) *System limitations due to insufficient design*

The apparent reason for this failure is the point (d), which is contrary to what was originally aired.

CASE # 3. The goal of this analysis is to focus on small software changes that frequently result in uncontrolled ramifications. Consider the following paper quote describing the crash of the New York Stock Exchange.

> "…. On Tuesday evening, the NYSE began the rollout of a software release in preparation for the July 11,2015 test of the upcoming SIP timestamp requirement. As a standard NYSE practice, the initial release was deployed on one trading unit. As customers began connecting after 7am on Wednesday morning, there were communication issues between customer gateway and the trading unit with the new release. It was determined that the NYSE and NYSE MKT customer gateways were not loaded with the proper configuration compatible with the new release….".

Selecting from the list of possible root-causes shall undoubtedly yield the point (d) as a true reason of failure.

a) Requirements
b) Design
c) Test
d) *SCM*

CASE # 4. On Christmas of 2017 South Carolina Lottery sold 70,000 tickets between 5:51 and 7:52 pm. Each ticket gave a purchaser an unfair chance of winning $500. Some folks seized on an opportunity, continued buying tickets and cashed them immediately. Most folks followed the strategy of saving winning tickets for a rainy day. The error-filled tickets totaled for $34 MLN. Tim Madden from state lottery confirmed that they paid $1.7 MLN due to the "programming error of their vendor Intralot" and did not honor any other erroneous tickets. Other officials called the situation a "machine malfunctioning" and "glitch".

Selecting from the following list the most likely cause of a problem - shall yield the point (b).

a) Requirements
b) *Coding*
c) Test
d) SCM

CASE # 5. A well-described case of a brokerage company going bankrupt. On August 1, 2012 Knight Capital lost $440 million, when its trading software went kablooey. As

a software engineer inadvertently left a piece of code commented out. Resulting in each transaction missing a fairly small amount. Although the huge volume of a flow of daily trades caused a colossal loss to accumulate very quickly. Within thirty minutes, it was all over, the market maker was out of business and has never recovered.

Apparently, (i) is the most probable root-cause of a problem.

e) Requirements
f) Design
g) Test
h) SCM
i) *Peer Reviews*

CASE # 6. Here is an example of several minor defects that manifested concurrently, causing a catastrophe, because of their compound effect. March 2019, two Boeing 747 Max crashed within a short time span, because of the following,

– faulty hardware sensor at the end of its life
– autopilot (AI system) misbehaved and pushed down the nose of the aircraft
– an urgent software fix was delayed due to some bureaucratic process
– disagree light (feature that is supposed to highlight the variance among sensors) was not installed; this safety feature was marked as optional and sold for $80K
– pilot training did not cover, how to override the misbehavior of an autopilot
– documented procedure was incorrect

Looking at this long list of issues - we arrive at no other conclusion that point (f) is the root cause.

a) Requirements
b) Design
c) Test
d) Documentation
e) Training
f) *Project Management*

Expanding on these far-reaching thoughts, [20, 21], we can confirm that it is insufficient to know a *part* of the process. One must be aware of *all* parts, to comprehend the gist of the *whole*. For example, it is impossible to grow into an expert in Design, while being unaware of Testing. Process components are interdependent; they influence each other. One might assume, "it is enough just to do my job, my little something". But they will be unable to comprehend even a small part, since in-depth angles are bound to be missed.

Figure 9 makes a visual representation of each part of a software process as a sector of a unified circle. As we delved into real-life cases (CASE 2 – 6), it became clear that skipping even a single sector can become a root cause of a profound tragedy.

10 Conclusions

A curriculum of a mature course must follow a clear direction, starting with an introductory material and moving toward more advanced concepts. [22, 23]. A course should 'develop a topic' by showing an explicit path to a learner. To this end, if we divide a

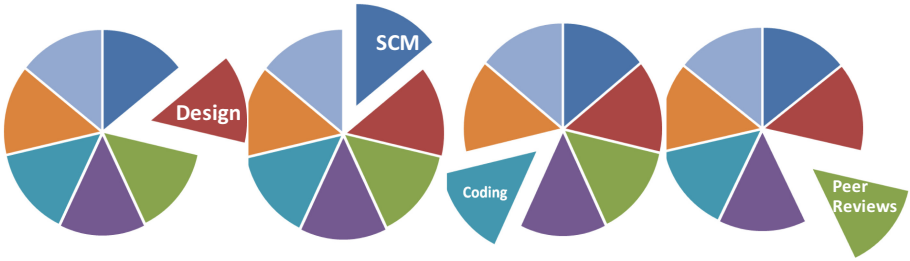

Fig. 9. Mapping catastrophic failures into process components

course into two parts, the first part covers a technique itself, its steps and roles; while the second part delves deeper into the *reasoning* of this approach and its connections to other approaches. In other words, we should divide the scope of a course into WHAT and WHY.

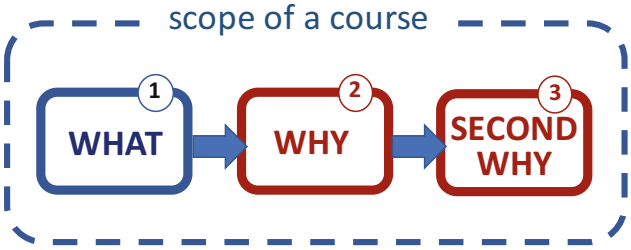

Fig. 10. Scope of the course is divided into WHAT and WHY

For example, the initial WHAT-part covers the software testing technique of *equivalence partitioning* or the peer reviews method of *step-by-step scenarios*. While the second part raises various WHY-questions. Here are several examples of a WHY-question.

- Why – defects tend to cluster
- Why – peer reviews are hugely more effective than personal editing
- Why – Cone of Uncertainty of a test effort is lopsided, as its top part is most commonly missing
- Why - Allpairs combinations reveal the same number of defects as Cartesian reveal
- Why – percent of a project's effort spent on unit test is diminishing, as the size of a project widens
- Why - Sarbanes-Oxley Act (SOX) mandates to separate the test design from test execution
- Why - An organization should keep growing its regression suites
- Why - A unit test should avoid communicating across a network
- Why – A unit test hardy ever goes through a redundancy checks, as a system test does
- Why - Technical debt cannot be addressed with a functional test

During our decade-long practice of teaching courses on quality software engineering – reviewing with students these WHY-questions - proves to be most effective to assure

that concepts did sink-in. We orchestrate a certain time pressure, as we ask students to write responses to several dozen WHY-questions in their own words and without a help of ChatGPT. Alternatively, during a verbal questioning, it becomes immediately clear - how fluent our students with these concepts and how comfortable they are discussing them. Such an articulation, reducing ideas to paper - does bring about various additional virtutes. Students who thought-through those topics that we covered in the class, students who spent time crafting responses based on their personal experience – are most likely know the subject well.

The topic of our paper is the survey of root causes of various software defects. In a context of a public course (in our case it is the software engineering course) it is important to structure the syllabus in a similar manner, so not to omit the second part, the WHY-part. As in bug fixing, we first observe the manifestation, the *failure*, although we still need to comprehend the *fault* so to correct it and to eliminate the failure. Much like during a university class, we first need to communicate to student *what* needs to be done and then to explain *why* it needs to be done.

Note that it is quite possible to mix-and-match the WHATs and the WHYs. One can very well intertwine the description of steps of a technique with the explanation of reasons it is done that way. Although in our experience, such teaching approach does not prove to be effective. The clear delineation of various parts of a curriculum (showing where WHAT ends and where WHY begins) clarifies and strengthens the overall message.

Figure 10. Develops the topic further by depicting the third step, the so-called SECOND WHY. It exposes the recursive nature of a discovery process, as it points to the deliberate structure of the course itself. Here are several examples fitting into the category of the SECOND WHY.

- Why – Design part of the course is juxtaposed with descriptions of extra-long Black Swan projects that have been halted without their completion
- Why – Peer Reviews part of the course is strategically positioned next to the description of Agile, as the most common software development paradigm
- Why – the description of clustering defects – is positioned within the *equivalence partitioning* chapter of the class

The introduction of the SECOND WHY brings about the concept of never-ending improvement. It motivates one to keep advancing and to keep delving deeper without limitations.

References

1. Jones, C.: Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies. McGraw-Hill Education (2010). A study by a renowned software engineering expert, confirming that bug fixing can consume 30% to 50% of total development effort
2. Coverity Scan: Open Source Report. A survey conducted by the provider of the static analysis tools, found that fixing bugs can consume up to 50% of development time (2012)
3. McConnell, S.: More Effective Agile. Constrrux Press. ISBN: 978–1–7335182–1–5. Prepublication Draft 21-February-2019
4. McConnell, S.: More Effective Agile. Constrrux Press. ISBN: 978–1–7335182–1–5. 01-September-2019

5. IEEE Standard for Software Reviews and Audits, IEEE Std 1028™-2008
6. Wiegers, K.: Peer Reviews in Software. ISBN-13: 978–0201734850
7. SEI CMM version 1.3. https://resources.sei.cmu.edu/library/index.cfm
8. ISO 9001:20015 QMS - Guidelines for Performance Improvements
9. Schmauch, C.H.: ISO 9000 for Software Developers, ASQC Quality Press. ISBN: 0–87389–348–4
10. Lehtinen, T.O.A., Mäntylä, M.V., Vanhanen, J.: Development and evaluation of a lightweight root cause analysis method (ARCA) - field studies at four software companies. Inf. Softw. Technol. **53**(10), 1045–1061 (2011)
11. Chillarege, R., et al.: Orthogonal defect classification - a concept for in-process measurements. IEEE Trans. Softw. Eng. **18**(11) (1992)
12. Ko, A.J., Myers, B.A., Chau, D.H.: A linguistic analysis of how people describe software problems. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (2016)
13. Reason, J.: Human Error. Cambridge University Press, Cambridge (1990)
14. NASA (Information System Division), report from finding software faults through a variety of defect prevention techniques
15. Keshta, I.M.: Software cost estimation approaches: a survey. J. Softw. Eng. Appl. **10**, 824–842 (2017)
16. Crosby, P.: Quality Is Free. McGraw-Hill (1979). For over four decades this book has been an inspiration for numerous improvement efforts
17. Krasner, H.: Cost of Poor Software Quality, Consortium for Information and Software Quality, Report (2021). One might not realize the depth and breadth with which software defects affect all angles of our lives, as shown in this report
18. Paschal, C.A., et al.: Reflections on Software Failure Analysis. CS-SE Conference, Washington, DC, USA (2017)
19. This site documents the variety of actual case studies, www.henricodolfing.com/. Many cases are also covered in the book "The Project Success Model", by Henrico Dolfing available from the same site
20. Carstensen, P., et al.: "Let's Talk About Bugs", Scandinavian Journal of Information Systems. Paper presents the detailed analysis of testing and bug fixing within the context of the Danish conglomerate
21. Elentukh, A., Kanabar, V.: Improving Teaching and Learning Effectiveness of Computer Science Courses – Case Study, CSECS 2019 (2019). This paper summarizes our experience teaching several courses at Metropolitan College of Boston University Computer Science department over five years. A number of innovative teaching techniques are presented in this paper. We specifically address the role of a project archive, when designing a course
22. Hillman, D., Schudy, R., Temkin, A.: Winning Online Instruction. Routledge (2022). The fabric of the book is weaved with the fundamental transformation from synchronous teaching (lectures) into asynchronous teaching (quizzes, assignment, discussions). Such transformation is not perfunctory, as it calls for an extensive discovery
23. Hou, J.: Project and module based teaching and learning. Int. J. Comput. Inf. Eng. **8**(3) 791–796 (2014)