Noname manuscript No.
(will be inserted by the editor)

Are Delayed Issues Harder to Resolve?

Tim Menzies, William Nichols, Forrest Shull, Lucas Layman

Abstract Many practitioners and academics believe in a delayed issue effect (DIE); i.e. as issues linger longer in a system, they become very much harder to resolve. This belief is often used to justify major investments in new development processes that promise to retire more issues, sooner.

This paper tests for the delayed issue effect in 171 software projects conducted around the world in the period from 2006–2014. To the best of our knowledge, this is the largest study yet published on this effect. We found no evidence for the delayed issue effect; i.e. the time to resolve issues in a later phase was not consistently more than when issues were resolved soon after their introduction.

This paper documents the above studies and explores reasons for this mismatch between this commonly held belief and empirical data. In summary, DIE is not some constant across all projects. Rather, DIE might actually be an historical relic that occurs intermittently only in certain kinds of infrequently occuring projects. This is a significant result since it predicts that new development processes that promise to faster retire more issues will have only a limited return on investment.

Categories/Subject Descriptors: D.2.8 [Software Engineering]: Process metrics.

Keywords: software economics, phase delay, cost to fix.

1 Introduction

In 2013-2014, eleven million programmers [30] and half a trillion dollars [3] were spent on information technology. Such a large and growing effort should be managed and optimized via well-researched conclusions.

It is standard practice in other fields, such as medicine, to continually revisit old conclusions [47]. Accordingly, this paper revisits the commonly held belief of a *delayed issue*

Tim Menzies
CS, North Carolina State University, USA, E-mail: tim.menzies@gmail.com

William Nichols, Forrest Shull
Software Engineering Institute , Carnegie Mellon University, USA. E-mail: {wrn,fjshull}@sei.cmu.edu

Lucas Layman
Fraunhofer CESE, College Park, USA, E-mail: llayman@fc-md.umd.edu

effect (hereafter, DIE). Later in this paper, we offer a precise definition for this effect. For the moment, we describe it as follows: it is far easier to resolve issues earlier rather than later in the lifecycle. Figure 1 shows an example of the delayed issue effect (relating the relative cost of fixing requirements issues at different phases of a project).

Basili & Boehm comment that since the 1980s, this effect

> "...has been a major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes" [12].

Other prominent authors have commented on its perceived usefulness as a rule of thumb for software engineers. McConnell mentions it as a "common observation" in the field and summarizes the intuitive argument for why it should be so:

> "A small mistake in upstream work can affect large amounts of downstream work. A change to a single sentence in a requirements specification can imply changes in hundreds of lines of code spread across numerous classes or modules, dozens of test cases, and numerous pages of end-user documentation" [37].

Glass also endorses this effect, asserting that "requirements errors are the most expensive to fix when found during production but the cheapest to fix early in development" is "really just common sense" [25]. Other researchers are just as adamant in asserting that that the delayed issue effect is a proven fact. For example, what we call the delayed issued effect was listed first by Boehm and Basili in their "Top 10 list" of "objective and quantitative data, relationships, and predictive models that help software developers avoid predictable pitfalls and improve their ability to predict and control efficient software projects" [12].

This paper calls into question all the above claims about the delayed issue effect. We suggest that the delayed issue effect might have been an dominant effect decades ago, but

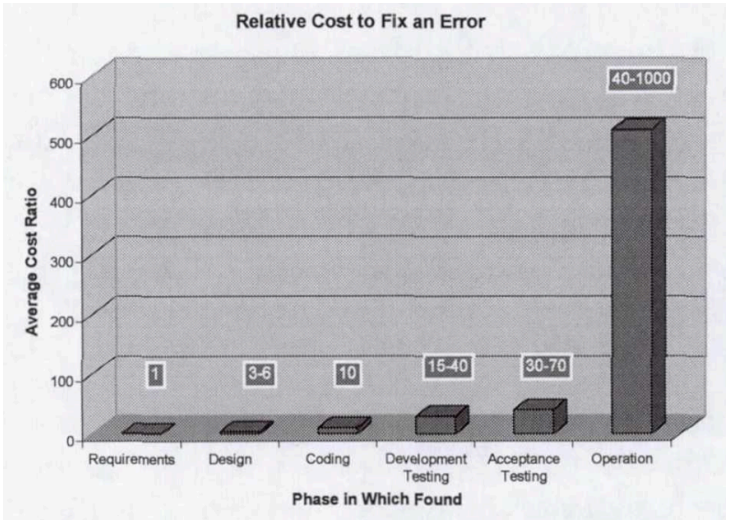

Fig. 1: An example DIE effect. From Boehm'81 [10].

not for 21^{st} software development. The delayed issue effect was first reported in 1981 in a era of punch card programming and non-interactive environments [10]. In the 21^{st} century, we program in interactive environments with higher-level languages and better source code control tools. Such tools allow for the faster refactoring of existing code– in which case, managing the changes required to fix (say) an incorrect requirements assumption is far less onerous than before.

Also, we note that our development practices have changed in ways that could mitigate the delayed issued effect. Previously, software was large monolithic systems that were "write once and maintain forever." Today, there are more architectures that support extensive and faster changes to software. For example, Microsoft is adjusting its development practices towards a continuous release paradigm. Theisen et al. reports their experiences where Microsoft architects continually adjust their modules in response to security issues [55]. Also, upgrades to their Windows operating system is moving from service patches (which occur rarely) to continuous deployment (so there will be no Windows 11- just a stream of continuous updates to what is currently called Windows 10 [14]).

For another example, consider the MEAN stack preferred for web development by the continuous deployment community. Older architectures for web development often used some variant of LAMP (Linux + Appache + Mysql + PHP) that was an intricate combination of tools written in different languages. MEAN stacks, on the other hand, use Javascript throughout– which makes large scale reorganizations faster to complete [56].

Inspired by the success of agile approaches, other development organizations are similarly reorganizing their work– witness the US Department of Defense's 2010 mandate that all new software acquisitions must adopt agile methods [34]. This change in DoD policy arises from a separation of baseline architecture (e.g. the design of an aircraft carrier) and the development of applications within that architecture. For the baseline architecture, bad decisions made early in the life cycle may be too expensive to change. But at least at the DoD, the majority of software development occurs within the framework of some larger architecture (e.g. an aircraft carrier). These smaller projects can certainly lever the agile advantage, while using fast refactoring tools that mitigate against the delayed issue effect.

The above argument is an anecdotal evidence that the delayed issue effect might no longer exist. But anecdotes are not really rigorous evidence. Accordingly, this article explores the currency of the delayed issue effect. After some initial definitions, we discuss the value of checking old ideas. Next, a survey of industrial practitioners and researchers to document a widespread belief that delayed issues have a negative impact on projects. After that, we analyze 171 software projects developed in the period 2006–2014 to find *no evidence* of the delayed issue effect:

- To ensure reproducibility, all the data used in this study is available in the PROMISE repository at openscience.us/repo.
- To the best of our knowledge, this the largest study devoted the delayed issue effect yet conducted.

Finally, we discuss the validity and implications of our results.

1.1 Preliminaries

Before beginning, it is appropriate to make the following full disclosure statement.

All the 171 software projects studied here were developed using the Team Software Process (TSP^{SM}), which is a software development methodology developed and promulgated

by the employer of the second and third author of this paper (for more details on TSP, see §6.1). Hence, it is tempting to use these results to advocate for TSP since TSP might solve the DIE problem for software projects without needing all the extra machinery seen in (say) Silicon Valley continuous deployment organizations[1].

That said, it is not clear to us that TSP is such a radical change to software development that it can stamp out a supposedly rampant problem like the delayed issue effect. We view TSP as a better way to *monitor* the activities of existing projects. TSP does not significantly change a project– it just offers a better way to log the activity within that project.

Hence, given our data, the conclusion of this paper is *either* the delayed issue effect can be removed via: simple changes to a project (like TSP) *or* that the DIE problem does not exist in the first place. Either way, our general point is the same: the delayed issue effect does not plague software development (which is a contrary claim to statements from leading figures in the field of software engineering – see the introduction).

2 Definitions & Hypotheses

This paper uses the following definitions:

- The *delayed issue effect*: it is *very much* more *difficult* to resolve issues in a software project, the *longer* they remain.
- *Longer* time is defined as per Boehm'81 [10]; i.e. the gap between the phases where issues are introduced and resolved.
- We say that a measure collected in phase $1, ., i, ..j$ is *very much* more when that measure at phase j is larger than the sum of that measure in earlier phases $1 \leq i < j$.
- Issues are more *difficult* when their resolution takes more time or costs more (e.g. needs expensive debugging tools or the skills of expensive developers).

Note that we use the term "delayed issue effect" as generalization of the more specific rule "requirements errors are hardest to fix". This generalization is valid since the rationale for the rule about requirements is usually done as per McConnell [37]; i.e. small upstream mistakes very early in the system can cascade into huge problems later in the lifecycle. That said, we prefer our more general term "delayed issue effect" since not only might requirements errors cascade, so too might analysis errors, design errors, etc.

This paper defends the following claims. Note that we call them "claims" not "hypotheses" since the later require defense via some statistical significance test. On the other hand, our claims will be supported via a variety of arguments, presented later in this paper.

Claim1: "DIE" is a commonly held belief. Using a literature review, we can confirm that there are numerous historical papers (dating back decades) that endorse DIE. Also, using a survey conducted from this paper, we find that DIE appears as the single most strongly-held beliefs amongst commercial software engineers.

Claim2: "DIE" is a poorly documented. As discussed in our literature review, many of the papers reporting the DIE effect are either (1) quite old (papers dating from last century); (2) quoting prior papers without presenting new data; (3) or citing data sources that can no longer be confirmed.

Claim3: Delayed Issues not Harder to Resolve. In our sample of 171 software projects developed, we will show *no evidence* that, during development, delayed issues were very much more harder to resolve the longer they were left in the software.

[1] E.g. large scale process changes such as open source development and/pr switching to new tools like NoSQL, Docker, AWS, Javascript and MEAN, Jenkins, Github, etc.

3 But Why Reassess Old Truisms?

Before going any further, we digress to discuss the merits of revisiting old conclusions in software engineering.

Beliefs in general principles of software engineering are common to both research and practice. Professional societies assume such generalities exist when they offer lists of supposedly general "best practices" such as the IEEE 1012 standard for software verification [29]. Endres & Rombach offer dozens of lessons of software engineering [20]. Many other commonly cited researchers do the same; e.g. Glass [25]; Jones [31]; Boehm [13]. Budgen & Kitchenham seek to reorganize SE research using general conclusions drawn from a larger number of studies [17, 35].

That said, there are many empirical findings that raise doubts that general laws of SE even exist:

1. Turhan [41] lists 28 studies with contradictory conclusions on the relation of OO measures to defects. Those results directly contradict some of the laws listed by Endres & Rombach [20].
2. Ray et al. [48] tested if strongly typed languages predict for better code quality. In 728 projects, they found only a modest benefit in strong typing (and caution that that effect may actually be due to other conflating factors).
3. Fenton & Neil [22, 23] critique the truism that "pre-release fault rates for software are a predictor for post-release failures" (as claimed by [18], amongst others). For the systems described in [24], they show that software modules that were highly fault-prone prior to release revealed very few faults after release.
4. Meyer claims that object-oriented (OO) encapsulation will reduce error rates in software [43]. Yet empirical results suggest that debugging an OO program is many times harder and longer than debugging a standard procedural program [26].
5. A truism of visual programming is that "visual representations are inherently superior to mere textual representations". A review by Menzies suggests that the available evidence for this claim is hardly conclusive [39].
6. Numerous recent *local learning* results compare single models learned from all available data to multiple models learned from clusters within the data [6, 7, 41, 42, 44, 46, 57, 58]. A repeated result in those studies is that the local models generated the better effort and defect predictions (better median results, lower variance in the predictions).

To be fair, SE is not the only field where practitioners hang on to persistent beliefs, even if the evidence for those beliefs is not strong. The medical profession applies many practices based on studies that have been disproved. For example, a recent article in the Mayo Clinic Proceedings [47] found 146 medical practices based on studies in year i, but which were reversed by subsequent trials within years $i + 10$. Even when the evidence for or against a treatment or intervention is clear, medical providers and patients may not accept it [1]. Aschwanden warns that "cognitive biases" such as confirmation bias (the tendency to look for evidence that supports what you already know and to ignore the rest) influence how we process information [2].

The cognitive issues that complicate medicine are also found in software engineering. According to Passos et al. [45], commercial developers are all too willing to believe in general truisms. They warn that developers usually assume that the truisms they learn from a few past projects are general to all their future projects. They comment "past experiences were taken into account without much consideration for their context [45]." The results of Jørgensen & Gruschke [33] concur with Passos et al. They report that supposed soft-

ware engineering "experts" rarely use lessons from past projects to improve their future reasoning [33]. They note that when the experts fail to revise their beliefs, this leads to poor conclusions and software projects (see examples in [33]).

In summary, just like medicine, our field suffers when software engineers do not revise old beliefs. Therefore, it is important to regularly reexamine old beliefs such as the delayed issue effect.

4 "DIE" is a commonly held belief

To assess the prevalence of DIE, we conducted a survey of software engineers. If our surveyed practitioners make management decisions based on their understanding of SE theory, then the DIE may well influence their decisions.

Our survey collected data on software engineers' views of commonly held software engineering "laws". One of the laws questioned is a specific form of our delayed issue effect: "requirements errors are the most expensive to fix when found during production but the cheapest to fix early in development" (from Glass [25] p.71 who references Boehm & Basili [12]). We abbreviate this law (illustrated in Figure 1) as RqtsErr.

(Technical aside: we use the RqtsErr formulation since this issue typically needs no supportive explanatory text. If we had asked respondents about our more general term "delayed issue effect", we would have had to burden our respondents with extra explanations).

The survey was conducted in two phases using Amazon's Mechanical Turk. The first phase was conducted only with professional software engineers solicited through the Mechanical Turk; participants were required to complete a pretest to verify their status as a professional or open source software developer and to confirm their knowledge of basic software engineering terminology and technology. The second survey was conducted with Program Committee members of the ESEC/FSE 2015 and ICSE 2014 conferences solicited via email.

The respondents answered questions on two scales:

- **Agreement:** "Based on your experience, do you agree that the statement above is correct?" A Likert scale captured the agreement score from Strongly Disagree to Strongly Agree. A text box was provided to explain the answer.

- **Applicability:** "To the extent that you believe it, how widely do you think it applies among software development contexts?" The possible answers were: -1 meaning "I don't know"; and 0 meaning "this law does not apply at all"; 1 meaning "applies in a very narrow range of projects"; 2 meaning "rarely applies"; 3 meaning "occasionally applies"; 4 meaning "very frequently applies"; and 5 meaning "always applies". Respondents were required to explain the applicability score in a text box.

In order to baseline the participant's answers, participants were presented with the RqtsErr law and others. For the purposes of this paper, the nature of the other laws other than RqtsErr are not relevant– we only added them in as a way to calibrate responses to the RqtsErr question. All laws were drawn from [25] and [20]. The PC member survey contained an additional question on "In general, the longer errors are in the system (requirements errors, design errors, coding errors, etc.), the more expensive they are to fix" (the Delayed Issue Effect). Responses were recorded using the agreement question Likert scale.

		agreement		applicability	
Practitioner survey	N	med	mode	med	mode
Rqts errors are most expensive...	16	5	5	4	5
Inspections can remove 90% of defects	18	4	5	4	5
80-20 rule (defects to modules)	12	4	5	4	5
Most time is spent removing errors	16	4	4	4	5
Process maturity improves output	17	4	4	4	4
Missing reqts are hardest to fix	17	4	4	4	4
Reuse increases prod. and qual.	16	4	4	4	4
OO-programming reduces errors	13	4	4	4	3
Adding manpower to a late project	15	4	4	4	4
Smaller changes have higher error density	14	3	3	3.5	5
A developer is unsuited to test own code	17	3	1	4	5
Researcher survey					
Process maturity improves output	4	4	4	4	5
Rqts errors are most expensive...	30	4	4	4	4
DelayedIssueEffect	30	4	4	–	–
Reuse increases prod. and qual.	6	4	4	4	4
80-20 rule (defects to modules)	6	4	4	4	3
Missing reqts are hardest to fix	7	4	4	4	3
OO-programming reduces errors	6	4	4	3	4
Inspections can remove 90% of defects	7	4	4	3	3
Adding manpower to a late project	4	3	4	4	3
Most time is spent removing errors	6	3	3	4	4
Smaller changes have higher error density	4	3	–	4	4
A developer is unsuited to test own code	7	2	1	3	3

Fig. 2: Agreement and applicability of SE axioms.

Summary statistics for the agreement and applicability scores for the RqtsErr and DelayedIssueEffect laws are presented in Figure 2. Responses whose Applicability response was ”I don’t know” are omitted from analysis.

Both practitioners and researchers strongly believed in RqtsErr: In both sets of responses, RqtsErr received scores higher than most other laws. Further, in the case of practitioners, this law was rated as the single most believed effect.

Caveat: in the free reponse texts, we note that the researchers who disagreed with the law generally asserted that requirements change can be expensive, but that the effect depends on the process used (e.g., agile vs. waterfall) and the adaptability of the system architecture.

Overall, the RqtsErr law was the most agreed upon and most applicable law of 11 surveyed amongst practitioners, and the second most agreed upon law amongst researchers. This results strongly support that the notion that the delayed issue effect is a commonly held belief in software engineering.

5 “DIE” is Poorly Documented

One reason that industrial practitioners and academics believe so strongly in the delayed issue effect is that it is often referenced in the SE literature. For example, we know that Figure 1 has been presented to the gradaute SE class at North Carolina State University, without quarrel or critical comment, every year for the last decade.

Yet when we look at the literature, the evidence for delayed issue effect is both very sparse and very old. The first data on the difficulties of resolving delayed issues as a function of lifecycle phase date back to large systems in the late 70s from IBM [21], TRW [8], GTE [16], and Bell Labs [54] (Figure 3). These studies are most often cited by secondary sources regarding the delayed issue effect. We note that it is unclear from the text in [16] and [8] if cost is defined in terms of effort, or in actual cost (i.e., labor, materiel, travel, etc).

These studies suggest that the difficulty (in terms of effort) to find and fix an error monotonically increases with lifecycle phase. In 1990, Boehm [9] provides data suggesting that the cost-to-fix curve for small projects (from two student projects of 2000 deliverable source instructions) is flatter than for large projects (the dashed line of Figure 3).

In the 40 years since these initial studies, few studies have explored the difficulty to resolve issues as a function of lifecycle phase. Shull et al. [52] conducted a literature survey and held a series of e-workshops with industry experts on fighting defects. Workshop participants from Toshiba and IBM reported cost-to-fix ratios between early lifecycle and post-delivery defects of 1:137 and 1:117 for large projects respectively [52] – but we note here that the raw data points were not provided (which makes confirming those numbers a difficult task).

This was a common theme in the literature reviewed for this paper– i.e. that it was no longer possible to access the data used to make prior conclusions. As an example of this, Figure 4 shows one 2004 survey that reports an greatly increased delayed issue effect for requirements issues in eight case studies. Note that we cannot verify any of those results since the links in the references of that survey are all broken ("page not found" errors).

As to the research into agile methods, one goal of that approach is to reduce the difficulty associated with making changes later in the lifecycle [5]. Relatively little empirical data exists on this point. Elssamadisy and Schalliol [19] offers an anecdotal report on the growing, high cost of rework in a 50 person, three-year, 500KLOC Extreme Programming project as the project grew in size and complexity– but again we cannot access their exact figures.

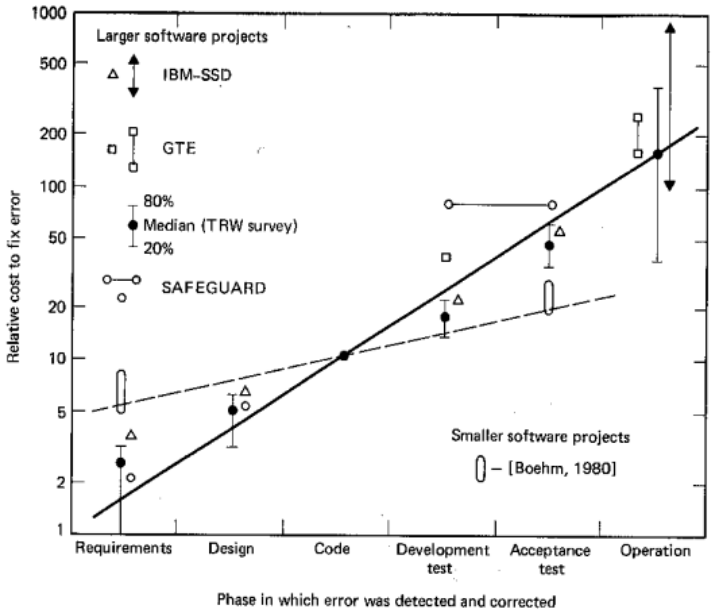

Fig. 3: Historical cost-to-fix curve. From [10].

case study	Phase Requirements Issue Found Requirements	Design	Code	Test
1	1	3	5	37
2	1		10	40
3	1		10	40
4	1	5		50
5	1	3	7	51
6	1	5	33	75
7	1	20	45	250

Fig. 4: Cost to resolve requirements issues, relative to resolving during requirements. From [53].

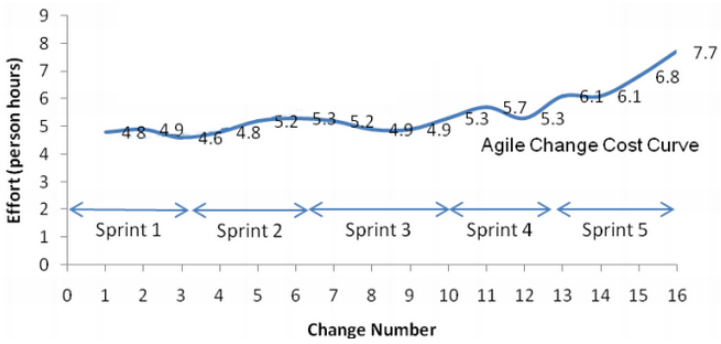

Fig. 5: Cost of change from an agile case study. From [15].

All in all, the literature reviewed in this section does not inspire confidence in a DIE effect due to a lack of applicability and lack of replication. Adding to our doubts are several studies that report less-than large increase in the difficulties associated with making the changes associated with delayed issues. Clutterbuck et al. [15] studied a 5-month effort by a small-to-medium enterprise team developing a 71KLOEC web interface to a database application to implement 18 change requests– see Figure 5 (note that these were for new and changed user requirements, not defects). Clutterbuck et al. found the cost of change to be relatively flat until the later phases, with much of the effort spent in analysis of the change requests [15]. Note that in this study, the effort increased by only 60% (see the start and end of the curve in Figure 5).

Another example of less-than large increase in the difficulty associated with delayed issues comes from Royce [49]. He studied a million-line, safety-critical missile defense system (see Figure 6). Design changes (including architecture changes) required approximately twice the effort of implementation and test changes, and the cost-to-fix in implementation and test phases increased slowly. Boehm [11] attributes this success to the development process, which focused on removing architecture risk early in the development lifecycle.

Other examples that report the opposite of DIE are:

- Boehm [9] reported a flatter growth rate for small, non-critical projects.
- Data from NASA's Johnson Space Flight Center, reported by Shull [52], found that the cost to find certain non-critical classes of defects was fairly constant across lifecycle phases (1.2 hours on average early in the project, versus 1.5 hours late in the project).

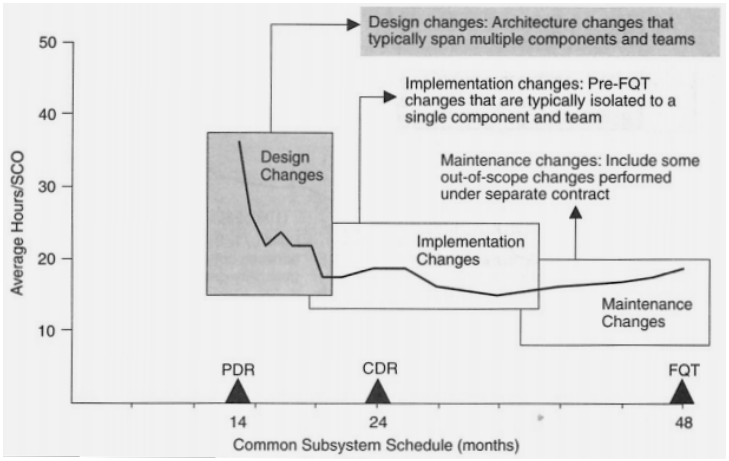

Fig. 6: Exception to the rule - The Royce study: cost-to-fix curve. From [49].

5.1 Early Onset of the DIE Effect

One feature of the above results will become very important in the subsequent discussion. All the literature described above that reports the onset of DIE *prior* to delivery.

- Figure 1 reports a 40-fold increase in effort requirements to acceptance testing
- Figure 3 reports a 100-fold increase (for the larger projects) before the code is delivered
- The projects surveyed in Figure 4 report changes of a {37,40,40,50,51,70,75,250}-fold increase in pre-deployment period when the software was being developed.

Any manager noticing this early onset of DIE (prior to delivery, during the initial development) would be well-justified in believing that the difficulty in resolving issues will get much worse. Such managers would therefore expect DIE to have a marked effect, post-deployment.

We make this point since, in the new project data presented below, there is no early onset of DIE. That is:

- There is no evidence of a growing problem prior to delivery that delayed issues are becoming harder to manage
- Hence, there there is less evidence that DIE is a trend that will significantly and negatively effect the product, post-delivery.

6 Delayed Issues not Harder to Resolve

The above analysis motivates a more detailed look at the delayed issued effect. Accordingly, we examined 171 software projects conducted between 2006 and 2014. These projects took place at organizations in many countries and were conducted using the Team Software Process (TSP^{SM}).

Since 2000, the SEI has been teaching and coaching TSP teams. One of the authors (Nichols) has mentored software development teams and coaches around the world as they

deploy TSP within their organizations since 2006. The most recent completions were in 2014. The projects were mostly small to medium, with a median duration of 46 days and a maximum duration of 90 days in major increments. Several projects extended for multiple incremental development cycles. Median team size was 7 people, with a maximum of 40. See Figure 7 for the total effort seen in those projects.

As to problem domain, many of the projects were e-commerce web portals or banking systems in the US, South Africa, and Mexico. There were some medical device projects in the US, France, Japan, and Germany as well as a commercial computer-aided design systems, and embedded systems.

An anonymized version of that data is available in the PROMISE repository at openscience.us/repo. For confidentiality restrictions, we cannot offer further details on these projects.

6.1 About TSP^{SM}

TSP is a software project management approach developed at the Software Engineering Institute (SEI) at Carnegie Mellon University [27]. TSP is an extension of the Personal Software Process (PSP^{SM}) developed at the SEI by Watts Humphrey [27]. The data from these TSP projects were collected and stored in the Software Engineering Measured Process Repository (SEMPR) at the SEI. The Software Engineering Institute (SEI) at Carnegie Mellon University explores methods for software process improvement.

Common features of TSP projects include *planning*, *personal reviews*, *peer inspections*, and *coaching*. A TSP *coach* helps the team to plan and analyze performance. The coach is the only role authorized to submit project data to the SEI. Before reviewing data with the teams, therefore before submission, these coaches check the data for obvious errors.

During *Planning*, developers estimate the size of work products and convert this to a total effort using historical rates. Time in specific tasks come from the process phases and historical percent time in phase distributions. Defects are estimated using historical phase injection rates and phase removal yields. Coaches help the developers to compare estimates against actual results. In this way, developers acquire a more realistic understanding of their work behavior, performance, and schedule status.

Personal review is a technique taken from the PSP and its use in TSP is unique. Developers follow a systematic process to remove defects by examining their own work products using a checklist built from their personal defect profile. This personal review occurs after some product or part of a product is considered to be constructed and before peer reviews or test.

Peer inspection is a technique in traditional software engineering and is often called peer review. Basili and Boehm commented in 2001 [12] that peer reviews can catch over half the defects introduced into a system. Peer inspection can be conducted on any artifact generated anywhere in the software lifecycle and can quickly be adapted to new kinds of artifacts. TSP peer reviews follow the Fagan style in which the reviewer uses a checklist composed of common team defects prior to a review team meeting.

Overall, the effort associated with adding TSP to a project is not onerous. McHale reports [38]:

- The time spent tracking time, defects, and tasks requires less then 3% of a developer's time. Weekly team meetings require at most an hour, which is only 2.5% of a 40 hour work week.

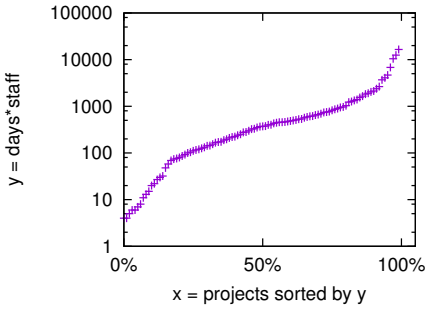

Fig. 7: Distribution of *effort* (which is *team size* times *days of work*). Median value = 271 days.

– Team launches and replans average about 1 day per month or 5% planning overhead.

It is true that one staff member is needed as a "coach" to mentor the teams and certify and monitor that data collection. However, one of us (Nichols) has worked with dozens of TSP teams. He reports that one trained coach can support 4 or 6 teams (depending upon team experience).

6.2 Data Integrity

A common property of real-world data sets is the presence of noisy entries (superfluous or spurious data). The level of noise can be quite high. As reported in [50], around 10% to 30% of the records in the NASA MDP defect data sets are affected by noise.

One reason to use the SEI data for the analysis of this paper is its remarkably low level of noise. Nichols et al. [51] report that the noise levels in the SEI TSP data are smaller than those seen in other data sets. They found in the SEI TSP data that:

– 4% of the data was incorrect (e.g. nulls, illegal formats);
– 2% of the data has inconsistencies such as timestamps where the stop time was before the start time;
– 3% of the data contained values that were not credible such as tasks listed in one day that took more than six hours for a single developer.

One explanation for this low level of noise is the TSP process. One the guiding principles of TSP was that people performing the work are responsible for planning and tracking the work. That is, all the data collected here was entered by local developers. This data was then checked by local coaches before being sent to the SEI databases. Coaches are certified by demonstrating competent use of the TSP process with the artifacts and data. The use of certified local coaches within each project increases the integrity of our data.

6.3 Data Details

Using tools provided by the SEI, developers kept very detailed logs of their daily activity. Our data includes work start time, work end time, delta work time, and interruption time. Software engineers are often interrupted by meetings, requests for technical help, reporting,

and so forth. These events are recorded, in minutes, as interruption time. In this paper, when we report "time to resolve an issue," we show the difference between the start and end times of a work session, with any interruption time subtracted (the difference in times, minus the interruptions).

As of November 2014, the SEI TSP database contained data from 212 TSP projects. The projects completed between July 2006 and November 2014; they included 47 organizations and 843 people. The database fact tables contain 268,726 time logs, 154,238 task logs, 47,376 defect logs, and 26,534 size logs. After selecting defects from the data log and joining the data to the time log table 171 of these projects remained (the excluded projects had no or too few defects to use in this analysis).

In these logs, a *defect* is any change to a product, after its construction, that is necessary to make the product correct. A typographical error found in review is a defect. If that same defect is discovered while writing the code but before review, it is not considered to be a defect. SEI TSP defect types are:

- Environment: design, compile, test, other support problems
- Interface: procedure calls and reference, I/O, user format
- Data: structure, content
- Documentation: comments, messages
- Syntax: spelling, punctuation typos, instruction formats
- Function: logic, pointers, loops, recursion, computation
- Checking: error messages, inadequate checks
- Build: change management, library, version control
- Assignment: package declaration, duplicate names, scope
- System: configuration, timing, memory

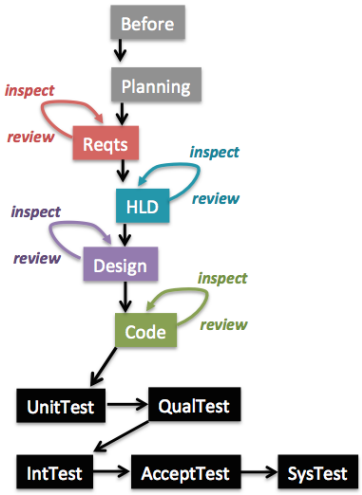

Fig. 8: Phases of our data. Abbreviations: *Before*= before development; *Reqts* = requirements; *HLD* = high-level design; *IntTest* = Integration testing (with code from others); *SysTest* = system test (e.g. load stress tests); *AcceptTest* = acceptance testing (with users); *review* = private activity; *inspect* = group activity.

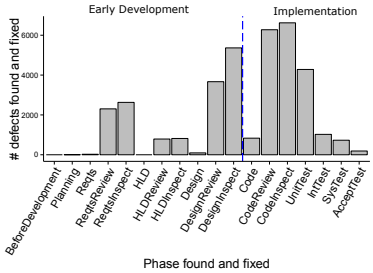

Fig. 9: Distribution of defects found and fixed by phase.

The defect logs in that data include the time and date a defect was discovered, the phase in which that defect was injected, the phase in which it was removed, the time (in minutes) required to find and fix the defect, and the categorical type.

The phases logged by our data are shown in Figure 8. Although the representation suggests a waterfall model, the SEI experience is that all real implementations of any size follow a spiral approach with many team performing the work in iterative and/or incremental development cycles.

One special feature of Figure 8 is the *before* phase, in which the TSP team assures that management has clearly identified cost, schedule, and scope goals appropriate to the upcoming development activities, often including a conceptual model [28]. For example an architecture team must have sufficient requirements to reason about, prototype, and specify an architecture [4] while a coding only team within a larger project would have more precisely defined requirements and high level design.

Note that, in that Figure 8, several phases in which the product is created have sub-phases of *review* and *inspect* to remove defects. In TSP reviews, individuals perform personal reviews of their work products prior to the peer review (which TSP calls the inspection). Also, Figure 8 divides testing as follows. Developers perform unit test prior to code complete. After code complete a standard phase is integration, which combines program units into a workable system ready for system test. Integration, system test, and acceptance test are often performed by another group.

Using the above, our units of analysis are:

- *defects* - individual defects are recorded as line items in the defect logs uploaded to the SEMPR at the SEI. One or more defects are reported against a single *plan item* in the time tracking logs, e.g., a review session, an inspection meeting, a test execution.
- *time* - Time is tracked per person per plan item in the time-tracking logs, e.g. a 30 minute design review session involving 3 people will have three time log entries summing to 90 minutes. Time includes the time to analyze, repair, and validate a defect fix.
- *time per defect* - The total # of defects found in a plan item during a removal phase divided by the total time spent on that plan item in that phase.

Phase injected	Phase removed	Percentile 50th	Percentile 95th	Ratio w.r.t. to phase where issue was injected) 50th	Ratio w.r.t. to phase where issue was injected) 95th
Before	DesignInspect	30	97	1	1
	CodeReview	28	94	0.93	0.96
	CodeInspect	45	154	1.5	1.58
	UnitTest	77	187	2.56	1.92
	IntTest	65	424	2.16	4.37
	SysTest	56	164	1.86	1.69
Planning	ReqtsReview	35	99	1	1
	DesignInspect	24	81	0.68	0.81
	UnitTest	33	238	0.94	2.4
Reqts	ReqtsReview	92	240	1	1
	ReqtsInspect	93	243	1.01	1.01
	DesignReview	33	119	0.35	0.49
	DesignInspect	49	137	0.53	0.57
	CodeInspect	38	156	0.41	0.65
	UnitTest	49	180	0.53	0.75
	IntTest	67	300	0.72	1.25
	SysTest	122	459	1.32	1.91
Design	DesignReview	103	331	1	1
	DesignInspect	101	272	0.98	0.82
	CodeReview	66	215	0.64	0.64
	CodeInspect	79	259	0.76	0.78
	IntTest	88	337	0.85	1.01
	UnitTest	119	383	1.15	1.15
	SysTest	80	336	0.77	1.01
	AcceptTest	45	120	0.43	0.36
Code	CodeInspect	113	337	1	1
	CodeReview	115	300	1.01	0.89
	IntTest	125	473	1.1	1.4
	UnitTest	133	434	1.17	1.28
	SysTest	95	383	0.84	1.13
	AcceptTest	65	360	0.57	1.06

Fig. 10: SEI TSP data: 50th and 95th percentiles of issue resolution times for issues injected in phase X and resolved in phase Y. The right hand side bars illustrate how these resolutions increases (expressed as ratios of resolution time in the phase where they were first injected). The **BLACK** and **RED** bars show the increases for the 50th (median) and 95th percentile values, respectively.

6.4 Observations from 171 Projects

The distribution of defects found and fixed per phase in our data is shown in Figure 9. high percentage of defects (44%) were found and fixed in the early phases, i.e., requirements, high level design, and design reviews and inspections. (This distribution is similar to that observed for other projects that emphasized investment in software engineering quality assurance practices. For example, Jones and Bonsignour report 52% of pretest defects removed before entering implementation, for large projects that focus on upfront defect removal techniques [32]. NASA robotics projects had a slightly higher percentage (58%) of defects removed before implementation began, although these had invested in IV&V on top of other forms of defect removal [40].)

Figure 10 shows the 50th and 95th percentile of the time spent resolving issues (note that, in TSP, when developers see issues, they enter *review* or *inspect* or *test* until that issue is retired). These values include all the time required to (a) collect data and realize there is an error; (b) prepare a fix; and (c) apply some validation procedure to check the fix (e.g. discuss it with a colleague or execute some tests).

In that figure, the data is split out according to issues that were fixed in phase Y after being introduced in phase X. The data is sub-divided into tables according to X; i.e. according to *before, planning, requirements, design* or *code*. To ensure the representativeness of each line, we display examples where there exists at least $N \geq 30$ examples[2] of issues *injected* in phase X then removed in phase Y.

The right hand side bars of that figure illustrate how these resolutions increases (expressed as ratios of resolution time in the phase where they were first injected). The **BLACK** and **RED** bars show the increases for the 50th (median) and 95th percentile values, respectively. Those bars offer some support for the claim that early lifecycle errors are most expensive to fix. In all our results, the biggest scale ups are found with resolving issues introduced in the *before* phase. For example, resolving *before* issues during unit testing was 2.56, 4.37 times (at the 50th and 95th percentile, respectively) slower than resolving issues earlier in the lifecycle.

Nevertheless, the scale up seen in Figure 10 is much less than that suggested by the DIE literature. Returning to the *before/integration test* result: 2.56 and 4.37 is far smaller than the scale ups seen in Figure 1. In fact, nowhere in our results do we see the kind of very large increases reported in the papers documenting DIE.

7 Threats to Validity

7.1 External Validity

When discussing these results with colleagues, they raised the following questions as potential threats to the external validity of this work.

Question1: All the projects used in this study were built using TSP. Does that mean non-TSP projects might expect DIE effects?

Answer2: As mentioned in §1.1, TSP is a lightweight data collection methodology that does not require significant changes to the other processes and tools. If the DIE effect was some widespread effect that significantly cripples many projects, it is unlikely that it could be solved by something as simple as TSP. What we find more likely is that DIE is an historical relic from last century when development practices were less flexible and tolerant of change.

Question2: Has this paper proved that DIE does not exist? Or just shown that it does not exist in certain projects?

Answer2: The claim of this paper is that the delayed issue effect is *much less of a concern* than stated by various authors (see the introduction). Our results show DIE is not highly prevalent and can be mitigated by relatively simple methods (e.g. TSP). In support of that claim, we offered data from 171 TSP projects and, in §5, reports from five projects by Clutterbuck [15] and Royce [49], Boehm [9, 11], and Shull [52]. That is, while there exist some reports of a DIE effect (see the handful of pro-DIE papers in [**?**]), we found many more projects that did not have problems with delayed issues. This is a significant result since, as stated by Boehm & Basili in the introduction, DIE has been used to justify many changes to early lifecycle software engineering.

Question3: The data collection for this work does not distinguished between "low risk" and "high risk" issues; i.e. between easy-to-fix issues and other most-expensive fixes. It is possible that such high risk issues might occur infrequently, yet are still detrimental since their resolution time is extremely large?

[2] We selected 30 for this threshold via the central limit theorem [36].

Answer3: If such outliers were common, they would appear in the upper percentiles of results. As shown in the **RED** bars of Figure 10, no such effect appears in the 95th percentiles of Figure 10. That is, while such "high risk" issues may certainly exist, they were not a major effect in our data.

Question4: The data used in this analysis does not extend into post-delivery deployment. Did this study failed to find the DIE effect since it does not arise until after deployment?

Answer4: As mentioned in §5.1, every other paper reporting DIE also reported early onset of DIE within the current development. Specifically: those pro-DIE papers reported very large increases in the time required to resolve issues *even before delivery*. That is, extrapolating those trends it would be possible to predict for a large DIE effect, even before delivering the software. This is an important point since Figure 10 shows an *absence* of any large DIE effect during development (in this data, the greatest increase in difficulty in resolving requirements issues was the 2.16 to 4.37 scale-up seen in the *before* to *integration testing* which is far smaller than the 37 to 250-fold increases reported in Figure 1, Figure 3, and Figure 4). That is, extrapolating the above data, we would not predict DIE effects, post-deployment.

Question5: While the 171 project studies here are a large sample, are they representative of all software projects? For example, Figure 7 shows that the median development time of these projects less than a year. What about longer projects?

Answer5: These 171 projects contain examples of a wide variety of systems (ranging from e-commerce web portals to banking systems) run in a variety of ways (agile or waterfall or some combination of the two). Hence, rather than being unduly biased in some manner, we view our 171 projects as a large sample of current industrial processes for software development.

As to projects that take more than a year to build, there certainly exists another class of very large software project where early requirements errors would be catastrophic for later development. For example, when NASA launches deep space missions, each of those satellites are special-purpose, uniquely-built devices built by a large contractor population. Such systems are very hard to reconfigure halfway through the development. We concede that for those very large complex NASA-like developments, the delayed issue effect could still hold. However, perhaps we need to split our software engineering principles into two parts- one for very large NASA-like projects (which are not very numerous) and one for smaller-sized projects like the ones studied here (which are far more numerous). Most software is *not* like the software used on NASA deep-space satellites. As agile methods grew more popular last decade; and as personal computers grew more powerful; and as the libraries associated with particular languages grew more extensive; then smaller teams starting deliver systems that (previously) had required very large teams.

7.2 Construct validity

Our definition of *difficult to resolve* combines two concepts: time to change and cost to change. In the above we have used them interchangably, comparing our time to resolve data (from the 171 TSP projects) against the cost-to-fix results of Boehm et al. e.g. Figure 1. Is it valid to assume the equivalence of time and cost?

Certainly, there are cases where time is not the same as cost. Consider, for example, if debugging required some very expensive tool or the services or a very senior (and hence, very expensive) developer. Under those circumstances, time does not equate to cost.

Also, there is the issue of the implication of a defect. Evidence discussed in [52] suggests that low severity defects may exhibit a lower cost to change. Nonetheless, even "small" errors have been known to cause enormous damage (e.g., the Mars Climate Orbiter).

Having documented the above issues, we assert that they are unlikely to be major issues in the study. One of us (Nichols) was closely associated with many of the projects in our sample. He is unaware of any frequent use of exorbitantly expensive tools or people on these projects.

7.3 Internal validity

Confounding variables are a threat to any study. In this research, we rely on the substantial sample size of 171 TSP projects and over 47,000 defect logs to avoid confounding effects that may arise from the nature of the projects, teams, or defects. Similarly, the 10-year data collection period should help to ameliorate any maturation or history effects. As described in §6.3, all TSP teams are required to contribute time and defect data to the SEI, and thus there should be no selection bias in this sample compared to the overall population of TSP projects. However, there is likely selection bias in the teams that elect to use TSP compared to the entire population of software development teams. Further, we assume that each team has similar defect recording practices, and the TSP coaching provides guidance on what constitutes a defect. Nonetheless, individual developers and teams may apply their own internal rules for filtering defects, which would lead to inconsistent reporting thresholds among the projects in our sample.

Another issue that threatens internal validity is that our results are dependent on developers correctly identifying which phase initially created an issue. Certainly, if that was done incorrectly in this study, then all our results must be questioned. However, this issue threatens *every* study on the delayed issue effect– so if our results are to be doubted on this score, then all prior work that reported the delayed issue effect should also be doubted. Moreover, the TSP method used in this study encourages greater care with error reporting:

- Developers are trained to make that judgement;
- All data entry is double-checked by the team couch
- TSP demands that developers analyze their data to make process improvements.

That is, TSP developers are always testing if their project insights are accurate. In such an environment, it is more likely that they will better identify the injection phase.

8 Conclusion

In this paper, we explored the papers and data related to the commonly believed *delayed issue effect* (that delaying the resolution of issues very much increases the difficulty of completing that resolution). Several prominent SE researchers state this effect is a fundamental law of software engineering [12, 12, 25, 37]. Based on a survey of both researchers and practitioners, we found that a specific form of this effect (requirements errors are hardest to fix) is commonly believed held in the community.

We checked for traces of this effect in 171 projects from the period 2006–2014. That data held no trace of the delayed issued effect. To the best of our knowledge, this paper is the largest study of this effect yet performed.

Al the data collected in this study comes from TSP-based developments. Hence, it is possible that the reason we did not find DIE is that TSP effectively mitigates for this problem. However, if DIE can be controlled by something as simple as TSP then it can no longer be regarded as a *"major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes"* (as said by Boehm & Basili in our introduction).

To be clear, we do not claim that this effect *never* holds in software projects; just that it cannot be assumed to *always* hold. Our explanation of the observed lack-of-effect is fivefold:

1. The effect might be an historical relic. Evidence: the effect was first described in the era of punch card computing and non-interactive environments.
2. The effect might have been intermittent (rather than some fundamental law of software). Evidence: we can found nearly as many papers reporting the effect [8, 10, 21, 53, 54] as otherwise [9, 49, 52].
3. The effect might be confined to very large systems- in which case it would be acceptable during development to let smaller to medium sized projects carry some unresolved issues from early phases into later phases.
4. The effect might be mitigated by modern software development approaches that encourage change and revision of older parts of the system.
5. The effect might be mitigated by modern software development tools that simplify the process of large-scale reorganization of software systems.

Our results beg the question: why did an idea, with so little support, become so widespread in the software engineering literature? No doubt the original evidence was compelling at the time, but much has changed in the realm of software development in the subsequent 40 years. Possibly the concept of the delayed issue effect (or its more specific description: requirements errors are the hardest to fix) has persisted because, to use Glass's terms on the subject, it seems to be "just common sense" [25]. Nevertheless, in a rapidly changing field such as software engineering, even commonly held rules of thumb must be periodically reverified. Progress in the domain of software analytics has made such periodic checks more cost-effective and feasible, and we argue that an examination of local behaviors (rather than simply accepting global heuristics) can be of significant benefit.

Acknowledgements

The authors wish to thank David Tuma and Yasutaka Shirai for their work on the SEI databases that made this analysis possible. In particular, we thank Tuma Solutions for providing the Team Process Data Warehouse software. Also, the authors gratefully acknowledge the careful comments of anonymous reviewers from the FSE and ICSE conferences. This work was partially funded by an National Science Foundation grants NSF-CISE 1302169 and CISE 1506586. Personal Software Process^{SM}, Team Software Process^{SM}, and TSP^{SM} are service marks of Carnegie Mellon University.

References

1. Christie Aschwanden. Convincing the Public to Accept New Medical Guidelines. `http://goo.gl/RT6SK7`, 2010. FiveThiryEight.com. Accessed: 2015-02-10.

2. Christie Aschwanden. Your Brain Is Primed To Reach False Conclusionss. http://goo.gl/OO3B7s, 2015. FiveThirtyEight.com. Accessed: 2015-02-10.
3. Abel Avram. Idc study: How many software developers are out there? infoq.com/news/2014/01/IDC-software-developers, 2014.
4. Felix H. Bachmann, Luis Carballo, James McHale, and Robert L. Nord. Integrate end to end early and often. *Software, IEEE*, 30(4):9–14, July 2013.
5. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
6. Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *MSR'12*, 2012.
7. Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Towards improving statistical modeling of software engineering data: think locally, act globally! *Empirical Software Engineering*, pages 1–42, 2014.
8. Barry Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, dec 1976.
9. Barry Boehm. Developing small-scale application software products: Some experimental results. In *Proceedings of the IFIP Congress*, pages 321–326, 1980.
10. Barry Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.
11. Barry Boehm. Architecting: How much and when? In Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*, pages 141–186. O'Reilly Media, 2010.
12. Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Software*, pages 135–137, January 2001.
13. Barry Boehm, Ellis Horowitz, Ray Madachy, Donald Reifer, Bradford K. Clark, Bert Steece, A. Winsor Brown, Sunita Chulani, and Chris Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.
14. Peter Bright. What windows as a service and a 'free upgrade' mean at home and at work. http://goo.gl/LOM1NJ/, 2015.
15. Peter Clutterbuck, Terry Rowlands, and Owen Seamons. A case study of sme web application development effectiveness via agile methods. *The Electronic Journal Information Systems Evaluation*, 12(1):13–26, 2009.
16. Edmund B. Daly. Management of software development. *Software Engineering, IEEE Transactions on*, SE-3(3):229–242, May 1977.
17. Barbara Kitchenham David Budgen, Pearl Brereton. Is evidence based software engineering mature enough for practice & policy? In *33rd Annual IEEE Software Engineering Workshop 2009 (SEW-33), Skovde, Sweden*, 2009.
18. Hubert E. Dunsmore. Evidence supports some truisms, belies others. (some empirical results concerning software development). *IEEE Software*, pages 96–99, May 1988.
19. Amr Elssamadisy and Gregory Schalliol. Recognizing and responding to "bad smells" in extreme programming. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 617–622, New York, NY, USA, 2002. ACM.
20. Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison Wesley, 2003.
21. Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
22. Norman E. Fenton and Martin Neil. Software metrics: A roadmap. In Anthony Finkelstein, editor, *Software metrics: A roadmap*. ACM Press, New York, 2000. Available from http://citeseer.nj.nec.com/fenton00software.html.
23. Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, pages 797–814, August 2000.
24. Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.
25. Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, Boston, MA, 2002.
26. Les Hatton. Does oo sync with how we think? *IEEE Software*, pages 46–54, May/June 1998.
27. Watts S. Humphrey. *Introduction to the Team Software Process*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.
28. Watts S. Humphrey. *TSP(SM)-Leading a Development Team (SEI Series in Software Engineering)*. Addison-Wesley Professional, 2005.
29. IEEE-1012. IEEE standard 1012-2004 for software verification and validation, 1998.
30. Gartner Inc. Gartner Says Worldwide Software Market Grew 4.8 Percent in 2013. gartner.com/newsroom/id/2696317, 2014.
31. Capers Jones. *Estimating Software Costs, 2nd Edition*. McGraw-Hill, 2007.
32. Capers Jones and Olivier Bonsignour. *The Economics of Software Quality*. Addison Wesley, 2012.

33. Magne Jørgensen and Tanja M. Gruschke. The impact of lessons-learned sessions on effort estimation and uncertainty assessments. *Software Engineering, IEEE Transactions on*, 35(3):368 –383, May-June 2009.
34. Don Kim. Making agile mandatory at the department of defense, 2013.
35. Barbara A. Kitchenham, Tore Dyba, and Magne Jørgensen. Evidence-based software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 273–281, Washington, DC, USA, 2004. IEEE Computer Society.
36. K D Maxwell. Applied Statistics for Software Managers. Englewood Cliffs, NJ, 2002. Prentice-Hall.
37. Steve McConnell. An ounce of prevention. *IEEE Softw.*, 18(3):5–7, May 2001.
38. James McHale. Tsp: Process costs and benefits. *Crosstalk*, September 2002.
39. Tim Menzies. Evaluation issues for visual programming languages. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*. World-Scientific, 2002. Available from `http://menzies.us/pdf/00vp.pdf`.
40. Tim Menzies, Markland Benson, Ken Costello, Christina Moats, Michelle Northey, and Julian Richarson. Learning better IV&V practices. *Innovations in Systems and Software Engineering*, March 2008. Available from `http://menzies.us/pdf/07ivv.pdf`.
41. Tim Menzies, Andrew Butcher, David R. Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans. Software Eng.*, 39(6):822–834, 2013. Available from `http://menzies.us/pdf/12localb.pdf`.
42. Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs global models for effort estimation and defect prediction. In *IEEE ASE'11*, 2011. Available from `http://menzies.us/pdf/11ase.pdf`.
43. Betrand Meyer. *Object-oriented Software Construction*. Prentice-Hall,Hemel Hemstead, 1988.
44. Leandro L. Minku and Xin Yao. Ensembles and locality: Insight on improving software effort estimation. *Information & Software Technology*, 55(8):1512–1528, 2013.
45. Carol Passos, Ana Paula Braun, Daniela S. Cruzes, and Manoel Mendonca. Analyzing the impact of beliefs in software project practices. In *ESEM'11*, 2011.
46. Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of ASE'11*, 2011.
47. Vinay Prasad, Andrae Vandross, Caitlin Toomey, Michael Cheung, Jason Rho, Steven Quinn, Satish Jacob Chacko, Durga Borkar, Victor Gall, Senthil Selvaraj, Nancy Ho, and Adam Cifu. A decade of reversal: An analysis of 146 contradicted medical practices. *Mayo Clinic Proceedings*, 88(8):790 – 798, 2013.
48. Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the ACM SIGSOFT 22nd International Symposium on the Foundations of Software Engineering*, FSE '14. ACM, 2014.
49. Walker Royce. *Software Project Management: A Unified Framework*. Addison-Wesley, Reading, MA, 1998.
50. Martin J. Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Software Eng.*, 39(9):1208–1215, 2013.
51. Yasutaka Shirai, William Nichols, and Mark Kasunic. Initial evaluation of data quality in a tsp software engineering project data repository. In *Proceedings of the 2014 International Conference on Software and System Process*, ICSSP 2014, pages 25–29, New York, NY, USA, 2014. ACM.
52. Forrest Shull, Victor Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258, 2002.
53. Jonette Stecklein, Jim Dabney, Brandon Dick, Bil Haskins, Randy Lovell, and Gregory Moroney. Error cost escalation through the project life cycle. In *14th Annual International Symposium; 19-24 Jun. 2004; Toulouse; France*, 2004.
54. W. E. Stephenson. An analysis of the resources used in the safeguard system software development. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 312–321, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
55. Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Companion Proceedings of the 37th International Conference on Software Engineering*. IEEE Institute of Electrical and Electronics Engineers, May 2015. Please note that this paper is not yet published, but accepted for inclusion.
56. Peter Wayner. Mean vs. lamp for the future of programming, 2015.
57. Ye Yang, Zhimin He, Ke Mao, Qi Li, Vu Nguyen, Barry W. Boehm, and Ricardo Valerdi. Analyzing and handling local bias for calibrating parametric cost estimation models. *Information & Software Technology*, 55(8):1496–1511, 2013.
58. Ye Yang, Lang Xie, Zhimin He, Qi Li, Vu Nguyen, Barry W. Boehm, and Ricardo Valerdi. Local bias and its impacts on the performance of parametric estimation models. In *PROMISE*, 2011.