What We Have Learned About Fighting Defects

Forrest Shull[†], Vic Basili[†‡], Barry Boehm[*], A. Winsor Brown[*], Patricia Costa[†], Mikael Lindvall[†], Dan Port[*], Ioana Rus[†], Roseanne Tesoriero[†], and Marvin Zelkowitz[†‡]

[†]Fraunhofer Center for Experimental Software Engineering, Maryland

[*]University of Southern California Center for Software Engineering

[‡]University of Maryland Empirical Software Engineering Group

Contributors:
Ed Allen (MSU), Frank Anger (NSF), Sunita Chulani (IBM), Noopur Davis (Davis Systems), Michael Dyer (Lockheed Martin), Christof Ebert (Alcatel), Bill Elliott (Harris Corp.), Eileen Fagan (Michael Fagan Associates), Martin Feather (JPL), Liz Green (Harris Corp.), Ira Forman (IBM), Scott Henninger (UNL), Philip Johnson (U. Hawaii), Oliver Laitenberger (IESE), Ray Madachy (USC), Yoshihiro Matsumoto (Toshiba), Tom McGibbon (ITT Industries), James Miller (U. Alberta), James Moore (MITRE), Don O'Neill (Don O'Neill Consulting), Stan Rifkin (Masters Systems), Dieter Rombach (IESE), Dan Roy (STTP, Inc.), Hossein Saiedian (U. Kansas), Giancarlo Succi (University of Alberta), Gary Thomas (Raytheon), Otto Vinter (independent software engineering mentor)

Abstract

The Center for Empirically Based Software Engineering helps improve software development by providing guidelines for selecting development techniques, recommending areas for further research, and supporting software engineering education. A central activity toward achieving this goal has been the running of "eWorkshops" that capture expert knowledge with a minimum of overhead effort to formulate heuristics on a particular topic. The resulting heuristics are a useful summary of the current state of knowledge in an area based on expert opinion.

This paper discusses the results to date of a series of eWorkshops on software defect reduction. The original discussion items are presented along with an encapsulated summary of the expert discussion. The reformulated heuristics can be useful both to researchers (for pointing out gaps in the current state of the knowledge requiring further investigation) and to practitioners (for benchmarking or setting expectations about development practices). The heuristics will be further refined during a physical expert workshop at the 2002 Metrics Symposium.

1. Building an experience base for software engineering

Software development is a people- and knowledge-intensive activity; it is a rapidly changing field, and although it is slowly maturing, many activities are still ad hoc and depend upon personal experiences. In order to cope with such restrictions as firm deadlines and shrinking budgets, software-developing organizations need assistance in setting up and running increasingly critical projects.

In order to reach their goals, software development teams need to understand and choose the right models and techniques to support their projects. They must answer key questions: what is the best life-cycle process model to choose for a particular project (from waterfall to extreme programming)? What is an appropriate balance of effort between inspections and testing in a specific context? What are the savings from buying a readily available software component instead of developing it?

These questions are not easy to answer. In some cases the knowledge exists to answer such questions; in other cases it does not, so instead of relying on knowledge and experience, we must trust our instincts. In order to support this decision-making activity, we need to develop empirically based software models in a systematic way, covering all aspects from high-level lifecycle models to low-level techniques, in which the effects of process decisions are well understood. However, context plays an important role as most projects and organizations differ. Consequently, the knowledge must be formulated relative to the development context and project goals.

The Center for Empirically-Based Software Engineering (CeBASE)[1] was organized to support this goal. CeBASE accumulates empirical models in order to provide validated guidelines for selecting techniques and models, recommend areas for research, and support software engineering education. CeBASE's objective is to transform software engineering from a fad-based practice to an engineering-based discipline in which development processes are selected based on what is known about their effects on products, through synthesis, derivation, organization, and dissemination of empirical knowledge on software development and evolution phenomenology.

CeBASE is a National Science Foundation-sponsored research center led by personnel with extensive industry and government experience, including co-authors

[1] http://www.CeBASE.org

Professors Barry Boehm (University of Southern California) and Victor Basili (University of Maryland and Fraunhofer Center for Experimental Software Engineering – Maryland). CeBASE collects, documents, and disseminates knowledge on software engineering gained from experiments, case studies, observations, and real world projects. While some of this empirical knowledge might be well known by the community, it has not yet been documented. Although this knowledge is believed to be generally applicable, the effects of its application have never been systematically investigated making it difficult to discern when it is useful. Some of this knowledge is distributed among many individuals, which means that we need to gather the pieces together and facilitate the collection and management of collective knowledge. The initial focus of CeBASE is on two high-leverage areas of software engineering, defect reduction and COTS based development.

2. Collecting expert knowledge on defect reduction

This paper describes the process and results to date of building up our understanding of what is currently understood about defect reduction in software development.

The goal of this work is to create a set of heuristics that represent what experts in the field consider to be the current state of understanding about the topic. To seed the discussion, a set of statements were proposed by Barry Boehm and Vic Basili in a "top-10" list that attempted to capture 10 useful and commonly accepted statements about the phenomena of software defects: the cost and effort associated with defects, the impacts of defects on software quality, and effective methods for reducing defects. CeBASE then sponsored a series of events to test, collect data on, and ultimately refine those statements. This series of events consisted of several eWorkshops followed by a physical capstone meeting at the Metrics Symposium, 2002, in Ottawa, Canada. Most participants in these events are experts in their respective domain. Our lead discussants (workshop leaders) formed part of the CeBASE team that interacted with an international group of invited participant experts.

Meetings among experts discussing their findings and recording their discussions are a classical method for creating and disseminating knowledge. By analyzing such discussions new knowledge can be created and the results can be shared. This is generally achieved by holding workshops. Workshops, however, possess limitations: 1) experts are spread all over the world and would have to travel, and 2) workshops are usually oral presentations and discussions, which are generally not captured for further analysis. To overcome these problems we designed the concept of the *eWorkshop*, using the facilities of the Internet.

The eWorkshop is an on-line meeting, which replaces the usual face-to-face workshop. While it uses a Web-based chat-application, it is structured to accommodate the needs of a workshop without becoming an unconstrained on-line chat discussion. The goal is to synthesize new knowledge from a group of experts as an efficient and inexpensive method in order to populate the CeBASE experience base. The idea behind the eWorkshop was to use simple collaboration tools, thus minimizing potential technical problems and decreasing the time it would take to learn the tools. Simultaneously, we set up a process, a support team and control room to ensure that there would be as few disturbances as possible once the eWorkshop was running.
To minimize disturbances during the meeting and to capture important information, we relied on a support team operating from a single control room. This support team consisted of the following roles: *moderator*, *director*, *scribe*, *tech support*, and *analyst*. The moderator was responsible for monitoring and focusing the discussion (e.g., proposing items on which to vote) and maintaining the agenda. Of the support team, only the moderator was an active participant in the sense that he contributed actual responses during the meeting. The director was responsible for assessing and setting the pace of the discussion. He decided when it was time to redirect the discussion onto another topic. As the discussion moved from one topic to another, the scribe highlighted the current agenda item and captured and organized the results displayed on the whiteboard area of the screen. When the participants reached a consensus on a particular item through a vote, the scribe summarized and updated the whiteboard to reflect the outcome. The contents of the whiteboard became the first draft of the meeting minutes. The analyst coded the responses according to the pre-defined taxonomy. The analyst entered one or more codes to categorize responses as they were entered. The tech support was responsible for handling any problems that might occur with the tools. For example, some participants accidentally closed their sessions and had difficulty logging into the meeting for a second time. The tech support assisted these participants in troubleshooting their problems
More details about the eWorkshop tool and processes can be found in [2].

3. Results to date

During the series of three eWorkshops on defect reduction, participants contributed their own data and experiences on the topic, which resulted in the following types of results:

- Refinement: The consensus of the experts was that the original statement was generally true, but new considerations were introduced that represented a deepening of understanding. For example, the statement might be accepted but bounds were put on the circumstances under which it applied.
- Addition: The experts added new and related

hypotheses that had some support and broadened the understanding of the same general topic.

- Restatement: The experts felt that the statement was not accurate, and reformulated a statement that was more generally accepted.
- Meta-statement: The experts were not satisfied with the original statement but discussed why the current state of knowledge did not allow it to be reformulated.

In the following sections, the original statements are presented along with a summary of the eWorkshop discussion concerning it. At the end of each section, the results of the discussion are summarized by the presentation of a new set of hypotheses or statements, organized using the following notation:

- *x*.1: Used to label a statement that is a refinement of the original statement *x*.
- *x*a: Used to label a statement that was added in response to statement *x*.
- *x*': Used to label a statement that re-states more accurately the topic addressed by original statement *x*.
- *x*m: Use to label a meta-statement concerning the current knowledge regarding statement *x*.

Excerpts of the discussions are presented below, along with the resulting statements about software defects. The full discussion summaries can be found at the CeBASE web site.[2]

3.1 Effort to find and fix

"Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase."

Discussion: General data were presented that supported an effort increase of approximately 100:1. Don O'Neill described data from IBM Rochester [10] in the pre-meeting feedback that found an increase in effort of about 13:1 for defect slippage from code to test and a further 9:1 increase for slippage from test to field (so, a ratio of about 117:1 from code to field). From Yoshihiro Matsumoto's experience in a software factory of 2600 IT workers, average rework time after shipment is 22.85 hours versus less than 10 minutes if the work had been done prior to shipment (a factor of 137:1). Other corroboration came from Ed Allen's experiences with a telecommunications client as well as Noopur Davis' experience.

An important distinction that emerged was that the large effort multiplier holds for *severe* defects; many defects with lesser impact will not cost appreciably more to change after delivery than before.

- Barry Boehm pointed out that the 100:1 factor was about right for critical defects on large projects, illustrated for example by the data from the Data Analysis Center for Software [11].
- Sunita Chulani also agreed that this factor was consistent with her experience for severe defects.

For non-severe defects, the effort multiplier was not nearly as large. Otto Vinter indicated that his data (which do not include requirements defects) show an approximately 2:1 relationship between after-shipment and before-shipment debugging effort: 14 hours after release versus 7.4 hours in testing before release. Barry Boehm said that the 2:1 relationship also held for the million-line CCPDS-R project done by TRW for the Air Force (described by Walker Royce [13]), in which early risk resolution and well-validated modular architecting were used to reduce early defects. Victor Basili also had data from NASA's Johnson Space Center, which didn't measure post-delivery defects but still showed that the effort multiplier associated with different defect types are different: the effort just to *find* a defect increased from

- 1.2 hours early in the project to 1.5 hours late in the project, for non-severe defects
- 1.4 hours early in the project to 3.0 hours late in the project, for severe defects.

Other variables likely to have an impact were proposed, although no supporting data was available. Gary Thomas pointed out that post-shipment costs would be expected to be raised even further when a different organization than the one that developed the software is responsible for the maintenance of the system. Philip Johnson said that research has so far neglected development environments that do not fit into the "waterfall family" of development approaches. For example, in XP, requirements and implementation phases are so entwined that it no longer makes sense to talk about "early" vs. "late" phases of development.

Result Summary: EWorkshop participants generally agreed that finding and fixing software defects after delivery is much more expensive than fixing during early stages of development – for certain types of defects. A 100:1 increase in effort from early phases to post-delivery was a usable heuristic for severe defects, but for non-severe defects the effort increase was not nearly as large. However, this heuristic is appropriate only for certain development models with a clearly defined release point; research has not yet targeted new paradigms such as extreme programming (XP), which has no meaningful distinction between "early" and "late" development phases.

Item 1'. Finding and fixing a *severe* software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

Item 1.1. Finding and fixing *non-severe* software defects after delivery is about twice as expensive as finding these defects pre-delivery.

[2] www.cebase.org/www/researchActivities/defectReduction/index.htm

3.2 Amount of avoidable rework

"About 40-50% of the effort on current software projects is spent on avoidable rework."

Discussion: Data in support of large amounts of rework on projects were cited by several participants:

- Vic Basili said that the 40-50% claim is borne out by the Cleanroom studies at NASA Goddard's Space Flight Center [1]
- Barry Boehm pointed out that Capers Jones' books (e.g. [9]) have data on rework costs, that indicate that the rework fraction goes up with the size of the project, and can go as high as 60% for very large projects.
- Don O'Neill submitted pre-meeting feedback data from the national benchmarking effort showing that the range across projects is wide, on the order of 20% to 80%.

However, there was some agreement that higher-maturity projects spend considerably less effort on rework. Brad Clark has published analyses of the effects of process maturity [6], in which the benefits at higher levels of maturity are traced mainly to the reduced rework effort.

- Gary Thomas, using data from Raytheon, cited a range of about 10-20% avoidable rework on higher-maturity projects.
- Barry Boehm said that some TRW projects, such as CCPDS-R, were also able to reduce rework effort down to 10-20%.

Because of this disparity between high- and low-maturity projects, Don O'Neill suggested that we should distinguish disciplined software engineering, structured software engineering, and ad hoc programming and seek to associate with each a characteristic level of effort spent on avoidable rework.

In general, comparing rework costs across projects is dangerous because it can be defined in several different ways. (For one example, Vic Basili pointed out that the rework effort collected from the Software Engineering Laboratory (SEL) at NASA was measured as the effort required to make changes due to defect corrections.) Winsor Brown pointed out that, further complicating the comparison, is how one accounts for the defect: if a detailed design defect is introduced during testing, it might be counted as a "test" defect or a "design" defect (although in the latter case it would likely be much cheaper to fix than other design defects).

But there are other potential rework measures for which researchers might not even be able to collect metrics, for example the rework that is found on volatile development teams, where people are often added or removed and as a consequence spend time relearning or redoing the same things.

Preventing defects and reducing rework is not free, but Barry Boehm reported that at TRW it was found that early prevention effort (via reviews, inspections, and analysis tools) had a 5:1 or 10:1 payoff.

Result Summary: Most eWorkshop participants believed that significant amounts of effort are spent on avoidable rework. However, the data across many projects had a much wider range than the proposed 40-50%; on some projects cited, for example, it was as low as 10-20%. In general, it was felt necessary to distinguish different types of software engineering process so that we could examine the avoidable rework rates for different types of environments.

Item 2': A significant percentage of the effort on current software projects is typically spent on avoidable rework.

Item 2.1: The amount of effort spent on avoidable rework decreases as process maturity increases.

3.3 Defects causing rework

"About 80% of the avoidable rework comes from 20% of the defects."

Discussion: On this topic, little data was put forward. While most participants (Thomas, O'Neill, Rifkin, Allen, Basili) indicated they believed that most of the avoidable rework comes from a small number of defects, no data from personal experience was cited. Some confirmatory data from the SEL and from the work of Khoshgoftaar and Allen was described.

Some time was spent trying to make definitions more clear. First, "rework" was defined broadly to include the effects of such things as changing operating systems, databases, or customer base; possibly also the re-configuration of tools.

Stan Rifkin suggested that the definition be refined by clarifying that "avoidable rework" is related to changes that are corrective (to mitigate the effect of defects) and performance-related (to improve system performance). A consensus then emerged that unavoidable rework was rework that came from other sources than defects, e.g. from adaptive, preventive, or user-requested changes to the code or architecture. Otto Vinter suggested to expand this definition by proposing that unavoidable rework could be caused by some defects that are simply too hard to prevent.

Most of the discussion centered on suggesting what types of defects were most likely to cause disproportionately large amounts of rework. Barry Boehm said that in his experience one source of high-rework defects is "architecture-breakers:" defects whose fix requires you to significantly change the architecture, which then ripples into design and code changes.

Stan Rifkin described his belief that it costs more to fix errors that are found "inappropriately" late in the process.

For example, we ought to be finding and fixing function errors early in the cycle and timing errors later, so function errors that aren't found until the later stages will cause a higher amount of rework. Barry Boehm commented that IBM data (from Ram Chillarege's papers [5]) do indicate that some defects tend to be found earlier (e.g., function defects) and others tend to be found later (e.g., timing defects). Another implication of this is that timing defects are probably more expensive to fix than function defects, because they can't be found in earlier phases where fixing would be cheaper.

Result Summary: There was general agreement that relatively few defects tend to cause most of the *avoidable* rework on software projects. (However, it was clear that there is a significant amount of unavoidable rework as well that comes from such sources as adaptive maintenance. We need to spend more work on characterizing the types of rework and their causes.) There wasn't a lot of confirmatory evidence about the 80/20 rule, but there weren't any strong counterexamples either. In terms of the implications of this statement, there was a general consensus that characterizing high-rework defects would be worthwhile.

Item 3': Most of the avoidable rework comes from a small number of software defects, where avoidable rework is defined as work done to mitigate the effects of errors or to improve system performance.

Item 3a: Some rework is simply unavoidable, for example, work arising from adaptive, preventive, or user-requested changes.

Item 3.1: Defects causing high amounts of rework are likely to be those that are "architecture-breakers" or that are found "inappropriately" late in the development process.

3.4 Modules contributing defects

"About 80% of the defects come from 20% of the modules and about half the modules are defect free"

Discussion: Data collected during development showed that defects are widespread among modules before release. Don O'Neill had data from the National Software Quality Experiment[3] (NSQE) that showed that almost no modules pass through an inspection without some defects being found. Dan Roy had student and industrial data from TSP which indicated that only about 10% of modules should be considered defect-free at compile time, progressing to around 90% by system test.

When the decision was made to focus the discussion on defects resulting from failures occurring after software delivery, the data submitted did tend to indicate that a majority of the defects come from relatively few modules. Aside from Dan Roy's TSP data showing that only 10% of the modules had any defects,

- Ed Allen cited studies of mature telecommunications products showing that only 10% of the modules that changed from one release to another contributed to user failures;
- Stan Rifkin recalled data from Nortel switches showing that 80% of the defects came from 20% of the most-changed modules;
- Christof Ebert said that data from Alcatel confirmed that 20% of modules contain about 40% to 80% of defects, depending on product line [8];
- Otto Vinter had data that 70% of defects come from 19% of modules.

Gary Thomas and Dan Roy (based on work on Landsat-D) also believed the heuristic to be true, although they had no hard data.

However it was recognized that the 80/20 heuristic is not a hard and fast rule but varies based on environmental characteristics such as: development processes, quality goals, complexity and age of the system, and degree of reuse.

Result Summary: This statement should really be split into two parts. As to whether about 80% of the defects come from 20% of the modules, some supporting data was submitted and the general consensus is that this heuristic can be used as a general rule of thumb. However, it should not be assumed to be true for all systems, but varies based on environmental characteristics such as development processes and quality goals.

On the second half of the statement, that half the modules are defect free, less data was submitted. Data from software inspections during development indicates that almost no modules are defect free; however post-release failure counts from an embedded system showed that about 40% of the modules contributed no defects.

Item 4': As a general rule of thumb, 80% of a system's defects come from 20% of its modules. However, the relationship varies based on environment characteristics such as processes used and quality goals.

Item 4'': During development, almost no modules are defect-free as implemented.

Item 4''': Post-release, about 40% of modules may be defect-free.

3.5 Defects contributing downtime

"About 90% of the downtime comes from at most 10% of the defects."

Discussion: The discussion began with a disparity of

[3] http://members.aol.com/ONeillDon/nsqe-results.html

opinions about whether organizations are collecting any data of this kind. Ray Madachy said that the intent of the heuristic was clear but he had no experience with organizations where downtime was a focus of measurement. Philip Johnson said he had seen that some companies have defect databases but it is highly unlikely that the information is traced back to resulting downtime or resulting changes. Stan Rifkin said that he had in fact seen data collected by clients of his but they are kept private, because the clients' contracts depend on service level agreements that give them a competitive edge.

Of those who had a feel for such data:

- Stan Rifkin said that in lower process maturity organizations, the downtime comes from a much larger spread of the defects than just 10%;
- Christof Ebert said that it depends on how you do the accounting, but that in telecommunications systems about 10% to 30% of all defects are classified as causing downtime and blocking systems;
- Gary Thomas said that this was a hard statement to support based on evaluating the data collected but that anecdotally, in his environment, for operational systems only 2% of defects recorded caused the system to go down (i.e. were Category 1 defects where the system was "Dead in the Water").

As Christof Ebert pointed out, what complicates the accounting is that downtime is not only a function of the system quality but also the environment, circumstances of usage, and the cost and time required for repair. Philip Johnson thought it unlikely that developers ever collect data at the level of detail that could answer this question, since there's no evidence that the effort required to trace defects to results would have a payoff for them.

Result Summary: On this statement, there was little consensus. There was a wide disparity of opinions about whether organizations even collect this information, and what they do with it if it is collected. Much of the discussion focused on what measures would need to be collected if organizations wanted useful insight in this area.

Item 5m: Insufficient data have been collected to posit a relationship between defects and the downtime they cause.

3.6 Contribution of peer reviews

"Peer reviews catch 60% of the defects."

Discussion: Participants submitted a wide range of data on this issue.

- Oliver Laitenberger submitted a list of published data from several companies, in which defect detection effectiveness ranged from 19% to 93%, with most (6/10) of the sources falling in the 50%-70% range.
- Bill Elliott provided data from a large project at Harris GCSD showing 64% of total defects in the product were found by inspection activities, and mentioned that these results seemed typical for the division, which had an average defect detection effectiveness of 68%. He also provided data summarizing industry averages for the number of defects remaining in software at various phases of the lifecycle. The averages (over hundreds of programs) show a 90% reduction in the number of defects between the design phase (99.5 faults/KLOC) and the beginning of testing phases (9.4 faults/KLOC), with most of that reduction due to inspection activities.
- Stan Rifkin referenced data from a case study in industry showing a rate of 70-80% for organizations with inspection experience.
- Don O'Neill summarized data received by the National Software Quality Experiment from across many organizations to show that the average detection rate was about 50-65% for less mature organizations, rising to 70-80% for structured software engineering organizations (which account for the majority of practitioners), and again to 85-95% for organizations employing disciplined software engineering practices.
- Dan Roy cited SEI PSP data showing an average of 60% of errors caught during design and code reviews, rising to 80% with the addition of cross reviews.
- Otto Vinter cited a study of various techniques to prevent requirements-related defects, which found that various forms of review techniques could together find about 60% of the requirements-related defects.
- James Miller cited data from experiments where review effectiveness was about 50%.
- Winsor Brown cited Michael Fagan's claim (which he has made since 1985) of "95% of defects found before testing" due to Fagan-style inspections.

Thus the 60% heuristic seemed useful as a rule of thumb to describe the data that was submitted – although that data described inspections in several lifecycle phases (requirements, design, or code), with different process definitions, in various domains. The discussion began with an attempt to do a better comparison of the data by at least agreeing on a common definition of the defect detection rate as a measure of review effectiveness.

Vic Basili proposed that the detection rate should be measured as the percentage found during a given review of all defects discovered in the product before release (that is, effectiveness should be determined by comparing the number of defects found during reviews with the number found during testing activities). The majority of participants

felt that, using this definition, a valid heuristic is that reviews find 60-90% of defects. [The 90% value as a "fuzzy upper bound" was supported by publications by Capers Jones, Richard Lindner [10], Tim Olson [12], and also from the data submitted by Brown, Laitenberger, and O'Neill (for disciplined teams).] As Philip Johnson pointed out, this definition is only meaningful in an environment where there is a clear boundary between development and release.

Several participants felt that this definition of defect detection rate does not capture all of the important aspects in measuring review effectiveness. Otto Vinter and Dan Port proposed a second definition, related to but extending the first: A review's defect detection rate should be measured as a percentage of all defects found over the lifetime of the product, including those found post-release. Vic Basili objected on the grounds that the total number of post-release defects can never be definitively known. However, Bill Elliott said that techniques exist to estimate the number of post-release defects at ship time.

Result Summary: On this issue there was consensus. Several participants described confirmatory evidence in the pre-meeting feedback and during the discussion. Although numbers varied, most sources reported that reviews caught more than half of a product's defects regardless of the domain, level of maturity of the organization, or lifecycle phase during which they were applied.

Item 6': Reviews catch more than half of a product's defects regardless of the domain, level of maturity of the organization, or lifecycle phase during which they were applied.

3.7 Contribution of perspective-based reviews

"Perspective-based reviews catch 35% more defects than non-directed reviews."

Discussion: The only data submitted come from Oliver Laitenberger, who observed a 20-60% detection rate due to perspective-based reviews during controlled experiments. [To preserve impartiality, the eWorkshop organizers did not enter their own data into the discussion, but a good summary of experiences with perspective-based reviews can be found in [14].] These data were observed for PBR, a procedural approach to the individual preparation for reviews. Despite the lack of data, participants tended to agree in general that such reviews had promise for increased effectiveness, since they:

- provide more assurance that all pages of a review document are covered equally. That is, giving reviewers a perspective or scenario to follow helps keep them focused when they might normally lose interest or get tired toward the end. (Barry Boehm)
- help enforce a "speed limit" by requiring reviewers to process the information they read from their particular point of view, rather than skimming quickly over it. (Dan Roy)
- alter thinking patterns, by asking people to review the information not by itself but by relating it back to some particular point of view. (James Miller)

Since perspective-based reviews seem to provide an effective way for reviewers to do the individual preparation phase, subsequent discussion centered on the relative importance of team meetings. Oliver Laitenberger felt that the individual preparation phase is more important for finding defects than the team meeting, so that efforts spent to improve an individual's effectiveness at finding defects in the first place are more important than efforts spent at optimizing the team meeting afterwards. Eileen Fagan disputed this by saying that Michael Fagan has data that show more defects found as a result of the meetings than were discovered during preparation. Winsor Brown wondered if the apparent discrepancy could be explained by the fact that, in Fagan's paradigm, inspection participants during the preparation phase are only required to prepare themselves to fulfill their role in the inspection meeting, at most identifying "areas of concern" that may be addressed during the meeting. Hence, the objective of individual preparation is not to identify defects explicitly.

Participants did generally agree that the review meeting has positive effects even aside from defect detection, such as filtering out false positives, allowing participants to learn from each other, and convincing authors to submit future documents with fewer defects (Oliver Laitenberger, Eileen Fagan, Otto Vinter, Barry Boehm).

Result Summary: Few data were provided to quantify the effectiveness of perspective-based reviews. However, most participants agreed that having multiple perspectives represented during software reviews was an effective practice. Discussion centered on why this might be so and the implications as a result for review processes.

Item 7': Having multiple perspectives represented during software reviews is an effective practice.

3.8 Contribution of disciplined personal practices

"Disciplined personal practices can reduce defect introduction rates by up to 75%."

Discussion: The pre-meeting feedback contributed little data concerning the effectiveness of disciplined *personal* practices. Don O'Neill cited figures from the NSQE showing that disciplined software engineering practices in general lowered defect insertion rates (10-15 defects/thousand lines vs. 20-30 defects/thousand lines for structured software engineering and 40-60 defects/thousand lines for undisciplined, ad hoc processes) as well as defect detection rates. Barry Boehm mentioned that similar rates

were seen during the calibration of the COQUALMO model: average defect introduction rates were 10/KLOC in requirements, 20/KLOC in design, and 30/KLOC in code (numbers are not cumulative across phases).

Dan Roy felt that the phrase *defect introduction* must be clarified. For example, the Personal Software Process defines a "defect" as anything that requires modification to the product, including compile errors. Because the NSQE is intended to measure inspection effectiveness, it doesn't use (nor does it need to) a definition that is as inclusive. Oliver Laitenberger agreed, saying that the focus of an inspection should not be on syntactical defects, which can be caught more cheaply by a compiler. Thus Dan Roy's definition would include many more items than are commonly counted in inspection data.

Participants spent time refining the heuristic by discussing what kinds of practices could be included under the heading "disciplined personal practices." The majority felt that PSP was not the *only* such practice, just the best known. Cleanroom was nominated as another such practice (Vic Basili, Barry Boehm), and Winsor Brown pointed out that under a suitably broad definition, even disciplined practices of "desk checking" could be included. There was a bit of debate about whether extreme programming (XP) practices could be included, although no clear consensus emerged.

Although little quantitative evidence of benefits for disciplined personal practices was established, participants did believe such practices provide benefits, especially in the areas of:

- Reducing defect introduction rates
- Increasing defect detection rates – although inspection yield may be lower because better practices introduce fewer defects into the product in the first place (Barry Boehm)
- Reducing the cost of repairing a defect – because disciplined practices may mean that the artifact creator can repair the artifact with less effort (Martin Feather)

In the absence of consensus on the issue, participants felt that a decision could only be made using some kind of framework to relate the defect insertion/removal numbers we do have from various phases. Two candidates were proposed:

- Martin Feather identified the "Defect Detection and Prevention" framework, an effort led by Steve Cornford of JPL. Although intended more for hardware defects, it seems applicable to hardware, software and combinations [7].
- Barry Boehm mentioned a model called COQUALMO, currently in its first draft, which tries to do this [3].

Result Summary: Few participants submitted data to support the heuristic concerning benefits of disciplined personal practices. Some discussion time was spent to come to a consensus as to what exactly should be included in the definition of such practices, although agreement was not reached on all candidates. Participants felt that the effectiveness of disciplined practices was related to a number of issues – defect introduction, removal, and cost-to-fix rates – across multiple stages of the lifecycle, and that without a framework to relate such numbers no global estimate of effectiveness could be reached. The discussion ended with two such frameworks being proposed.

Item 8': The effectiveness of disciplined personal practices is related to a number of issues (such as defect introduction, removal, and cost-to-fix rates) across multiple stages of the lifecycle.

Item 8m: The real effect of personal practices on software defects cannot be quantified without a framework for relating defect introduction and removal rates across lifecycle phases.

3.9 Cost of high-dependability

"All other things being equal, it costs 50% more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if significant operations and maintenance costs are involved."

Discussion: Barry Boehm, who formalized the original heuristic, explained that the phrase "all other things being equal" in Item 9 came from the context of the COCOMO II calibration, which found that it was necessary to normalize the effects of often-correlated variables such as system complexity, development time and storage constraints. The total system cost escalation factor becomes a good deal higher when these effects are compounded.

The data contributed by participants was sparse, but did indicate that high dependability is much more expensive than 50%. Dan Roy cited a study at NASA HQ that found a factor of 3 increase between the cost of relatively low-dependability ground software ($70/LOC) and that of high-dependability flight software ($220/LOC). Christof Ebert also reported that Alcatel considers high-dependability software 10 times more expensive in the domain of distributed embedded legacy real-time environments.

Don O'Neill addressed the question of whether development processes suitable for high-dependability software are more expensive, by using data from the NSQE to track the relative cost of ad-hoc, structured, and disciplined software engineering. NSQE data show that structured software engineering is 100% more expensive than ad-hoc while disciplined costs 200% more.

Although participants found no appropriate measures during the discussion, Frank Anger felt that having ways to measure "levels of confidence" or "levels of dependability"

for a system, and relating those levels to productivity rates or costs levels, was still a useful research goal.

Result Summary: High dependability software costs more per source instruction than low dependability software products. This being the consensus, participants of the discussion tended to agree that the cost factor is much more than 50% higher for high-dependability. Participants suggested that the cost may be from 3 to 10 times more expensive.

Item 9': High-dependability software costs three to ten times more per source instruction than low-dependability software.

3.10 Software quality at delivery

"About 40-50% of user programs enter use with nontrivial defects."

Discussion: Participants began by discussing whether a majority of commercial systems contain defects at the time of release. Although no direct statistics were cited, several supporting facts were proposed that seemed to indicate a high likelihood of software being released with nontrivial defects:

- Results from the NSQE, cited by Don O'Neill, reveal that during development almost every inspection finds some significant defects. In over 3,000 inspection sessions, only a couple dozen have ever produced a zero yield.
- Dan Roy said that PSP data on hundreds of engineers shows that any program bigger than 200 LOC will have some bug in it. If only 10% of these are "nontrivial" then any program bigger than 2KLOC has one of these nontrivial defects.
- A rough estimate by Otto Vinter that every commercial product has a re-release every six months, not necessarily for updates in functionality but rather also to correct nontrivial defects that surface after some time in use, seems a consequence of high rates of systems entering use with nontrivial defects.

Barry Boehm narrowed the discussion by noting that the original intent of the statement was to measure the software written by software users, not professional developers. The participants reached a consensus around a definition proposed by Vic Basili, defining a user program as one written by a non-professional software developer for use by other than himself. To help illustrate that this is a real phenomenon, Scott Henninger gave the example of NASA, where scientists often write their own software to analyze the data being returned from satellites. Given this definition, the only data that could be found was the original data Barry Boehm cited for the heuristic: lab studies reported 35-90% of models had defects; 21-26% of operational spreadsheet models had defects [4].

Given the lack of data, participants discussed if this was a heuristic that was important to be further investigated. Winsor Brown felt the answer was "yes" and the underlying issue is that non-professional programmers can be helped if the software engineering field can reach out and teach them basic principles, for example for the process of creating high-quality spreadsheets. Scott Henninger felt the important issue was whether we had a true understanding of the difference between professional and non-professional programmers; how many professionals might not be actually applying what we would consider good software engineering practices?

Result Summary: There was a consensus that a majority of systems have non-trivial defects when they enter use, although participants felt that the figure is higher than 40-50% and not limited only to user-created programs. Experience seems to indicate that a large percentage of software systems of all types contain defects that affect execution. Such information can serve as a baseline for future assessments of development effectiveness.

Item 10': More than half of all types of software systems enter use with defects that affect execution.

4. Implications for researchers

Aside from recognizing where consensus agrees on aspects of software defect reduction, the eWorkshops were also helpful in identifying areas where the current state of the knowledge is insufficient to draw the level of conclusions that were desired. In this way the participants also helped point to important open research questions. For example, before the items on the original list can be entirely corroborated, some gaps in the existing knowledge must be addressed:

- Item 5m: Insufficient data have been collected to posit a relationship between defects and the downtime they cause.
- Item 8m: The real effect of personal practices on software defects cannot be quantified without a framework for relating defect introduction and removal rates across lifecycle phases.

Additionally, during the discussions important ideas were raised for extending the list, or for identifying situations in which the items on the list do not give enough information:

- How do we measure the impact of software defects in non-waterfall lifecycles, where requirements and implementation phases are so intertwined that it no longer makes sense to talk about "early" versus "late" phases of development?
- What are the root causes for different types of defects, and can we find preventive mechanisms that can improve resulting software quality?

- What are the root causes for rework effort on projects, and can we better distinguish when rework is avoidable and when it is necessary?

5. Implications for practitioners

For practitioners, the resulting list of statements represents a summary of what an influential sector of the software engineering community feels to be the current state of the knowledge in an important area, defect reduction. It shows that there are some underlying principles of software development that tend to hold across development environments and problem domains, and begins to identify some of the important factors that can cause results to vary from one project to another. For example, several statements described evidence that increased process maturity affects the results of software development in a positive way (e.g. Item 2.1, which said that effort spent on avoidable rework decreases as process maturity increases). Such statements can be useful for benchmarking (comparison of a particular project to what is known in general) and decision-making (summarizing what can be expected from software development in general).

6. Conclusions

CeBASE has an ambitious goal of collecting relevant empirically-based software engineering knowledge. Based on our experiences on the topic of defect reduction, the eWorkshop has been shown to be a mechanism for inexpensively and efficiently capturing this information. The eWorkshops have been useful for testing the items in the top-ten defect reduction list, and we have obtained additional references and data that seek to classify when specific defect reduction heuristics are applicable.

To provide a capstone for this series of discussions, we are holding a final, physical workshop co-located with the International Symposium on Software Metrics 2002 in Ottawa. Participating experts will argue for or against the revised statements, and present data to back up their position. The workshop will result in an updated, final list of heuristics useful to broad swathes of the software engineering community as a representation of what the field has learned based on years of observation of developers at work. The heuristics will be recorded along with a summary of the contributed data and observations so that the conclusions can be traced back to their supporting evidence.

7. Acknowledgements

This work is partially sponsored by NSF grant CCR0086078, establishing the Center for Empirically Based Software Engineering (CeBASE). We want to thank Scott Henninger, Rayford Vaughn, and Michael Frey as well as all participants for their contribution to the success of the eWorkshops.

8. References

[1] Basili, V.R., and Green, S. “Software Process Evolution at the SEL,” *IEEE Software*, July 1994, pp. 58-66.

[2] Basili, V. R., Tesoriero, R., Costa, P., Lindvall, M., Rus, I., Shull, F., and Zelkowitz, M. V. "Building an Experience Base for Software Engineering: A report on the first CeBASE eWorkshop", Bomarius, Frank and Komi-Sirviö, Seija, Springer, *In Proceedings of* Profes (Product Focused Software Process Improvement), pp. 110-125, 2001.

[3] Boehm, B., Horowitz, E., Madachy, R., Reifer, D., Clark, B., Steece, B., Brown, A.W., Chulani, S., Abts, C. *Software Cost Estimation with COCOMOII*, Prentice Hall, 2000.

[4] Chen, H.C., Ying, C., and Peh, C.B. “Strategies and Visualization Tools for Enhancing User Auditing of Spreadsheet Models,” *Information and Software Technology*, Dec. 2000, pp. 1037-1043.

[5] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M. “Orthogonal Defect Classification – A Concept for In-Process Measurements,” *IEEE TSE*, vol. 18, no. 11 (Nov. 1992), pp. 943-956.

[6] Clark, B. “The Effects of Process Maturity on Software Development Effort,” Ph.D. Dissertation, University of Southern California, 1997.

[7] Cornford, S.L., Feather, M.S., Hicks, K.A. “DDP – A Tool for Life-Cycle Risk Management,” In *Proc. of Aerospace Conference 2001*. Vol. 1, pp. 441-451.

[8] Ebert, C. “Metrics for Identifying Critical Components in Software Projects,” *Handbook of Software Engineering and Knowledge Engineering.* 2001.

[9] Jones, C. *Applied Software Measurement.* 1996.

[10] Lindner, R. J., and Tudahl, D. “Software Development at a Baldrige Winner,” Proceedings of ELECTRO’94, Boston, MA, May 12, 1994, pp. 167-180.

[11] McGibbon, T. Software Reliability Data Summary, DACS, 1996.

[12] Olson, T. “Performing Best In Class Software Inspections,” *Proc. of SEPG 2001.*

[13] Royce, W. *Software Project Management: A Unified Framework*, Addison-Wesley Object Technology Series, 1998.

[14] Shull, F. "Software Reading Techniques." In *The Encyclopedia of Software Engineering*, Second Edition. Copyright John Wiley & Sons, 2002.