Modeling Defect Trends for Iterative Development

John D. Powell
John.Powell@jpl.nasa.gov

John N. Spanguolo Jr.
John.N.Spagnuolo-Jr@jpl.nasa.gov

Caltech, Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena CA 91109-8099

Abstract

The Employment of Defects (EoD) approach to measuring and analyzing defects seeks to identify and capture trends and phenomena that are critical to managing software quality in the iterative software development lifecycle at JPL. The EoD approach uses defects as a source of knowledge about a software system and its iterative development in a manner that is advantageous to the development team. Since, the iterative software development lifecycle necessitates the building of a system while simultaneously learning about and gaining understanding of the software system, some level of defects can be expected due to incomplete knowledge during development of various software artifacts. The EoD approach not only seeks to track defects to their eventual removal from the system but to employ them as a means of measuring and facilitating the concurrent learn-while-building process.

1. Introduction

The waterfall life cycle is characterized by the full and correct completion of each phase (requirements, design, code and test) before beginning the next phase. Conversely, the iterative lifecycle seeks to complete the work in each of the phases in increments. By iterating through the requirements design code and test phases multiple times the software artifacts of each phase evolve in a far more concurrent fashion than the waterfall lifecycle. Software artifacts include requirements and design documents, code and its documentation, test plans and test results.

The iterative software development lifecycle has largely replaced the waterfall lifecycle throughout the software industry. Development of a series of parametric defect models that leverages predictive models, like COQUALMO, [1,2] and additional models and augmentations that specifically address software development phenomenon in the iterative development style are necessary to understand and control defects in this environment. Development of engineering models, including defect trend models, for this iterative environment requires a fundamental shift in the view of the role of defects and defect repairs within the development lifecycle. The treatment of defects within a waterfall lifecycle is solely as an unwanted byproduct of development. Conversely, an iterative lifecycle’s approach to learning about a software system while building it necessitates reliance, to some degree, on the discovery of defects to facilitate that learning. Thus, defects are a necessary development and management tool in the iterative lifecycle. This new reliance on an item (defects) that is ultimately detrimental to the software system, if allowed to go unchecked, demands rigorous management to ensure the delivery of high quality software on time and within budget. This paper describes a quantitative analysis approach (EoD), based on specific software metrics, to aid software projects in defect management. The EoD approach calls for the “employment” of defects to aid software development as a form of in-process knowledge acquisition about the system in addition to their eventual removal. Employing defects within an iterative style of software development seeks to formally:

- Make use of defects as a source of systematic learning throughout the lifecycle by formulating distinctions within the set of
defects and their characteristics and then tracking and analyzing data regarding these characteristics over time from past JPL software projects through current and future projects.

- Capture the characteristics that account for the developmental flexibility, which may benefit or endanger (depending on their usage) software projects developed in JPL’s iterative style. Then, form baselines to discriminate between the two possible effects probabilistically. These include but are not limited to proper deferment of defect repairs to later lifecycle stages in an attempt to maximize quality and minimize programmatic risks.

- Operate in conjunction with existing predictive defect models, such as COQUALMO, [3,4,5] as a means of prediction across and within an organization's multiple projects over time.

This defect approach augments the waterfall style of predictive models like COQUALMO. Defect measurement and analysis approaches that account for the dispersion in the development team’s learning from the requirements phase to all phases and iterations of software development. Currently, many software projects are forced to treat iterative software development as a series of “mini-waterfalls” for defect analysis purposes. This EoD approach while useful fails to fully capture many aspects of iterative software development such as the cumulative affect (good and bad) of previous iterations on the current iteration. Section 2 contains the rationale of the defect measurement approach (EoD). Section 3 will offer a brief discussion of the waterfall versus the iterative software development lifecycle in the context of defects and the EoD approach outlined in this paper. Section 4 will discuss the reasons why the “mini-waterfall” mapping is an incomplete defect rationale and fails to capture the sufficient/correct information in an iterative environment to enable reliable predictors and goes on to show how the employment approach to defects seeks to overcome these barriers. Section 5, is a detailed discussion of the EoD approach and techniques. These measures and techniques will be specifically related to iterative development’s intentional propensity to delay the full completion of software artifacts until more is known about the system. Finally, section 6 summarizes the view of defects as a valuable source of developmental feedback despite the expectation that significant levels of defects will be introduced as a result of developing software iteratively.

2. The Rationale for EoD in the Iterative Software Development Lifecycle

The rationale behind the EoD approach specifically for phenomena of iterative software development lifecycle is:

- To allow cross project analysis of defect trends.
- Ultimately anticipate, the advantages and disadvantages of iterative software development with respect to defects.

While iterative software development has largely become the standard development paradigm throughout much of the industry, defect measurements approaches often cling to assumptions rooted in the waterfall development paradigm. Thus, flexibilities inherent in iterative software development are often not formally addressed. When properly managed, incremental flexibilities offer powerful advantages during iterative software development. However when abused or poorly managed those same flexibilities can represent dangerous pitfalls. The EoD measurement approach described in this paper offers a means by which to manage the defect occurrence and elimination process to the software project’s benefit. The task of mapping waterfall-like rationales to an iterative environment is left for the analyst to informally perform on a case-by-case basis. The informal and the case-by-case nature of the mapping directly hinders an organization’s ability to perform cross project analysis of defect trends over time and thus the ability to develop reliable leading indicators.

Some components of defect measurement must begin before defects are even discovered in order to effectively employ defects during iterative software development to improve quality while removing. Namely, tracking when software artifacts are developed in both Regular Time and Logical Time. Regular time refers to a calendar date. Logical Time refers to the development phase
(requirements, design, code, test) and iteration (iteration 1, 2, 3...). These measurements can be easily obtained if a relatively well-defined configuration management process is in place.

When defects are discovered the characteristics of the defect and its repair must be tracked. At a high level these can be categorized as:

- Where defects were found?
- Where repairs were made?
- When artifacts were developed (See above)
- When defect were discovered?
- When artifacts were repaired? (regular and logical time)
- What precipitated the defect? (Environmental Change, New system knowledge, Previous repair, system evolution, etc.)

These measures will be discussed in further detail in section 5 along with non-exhaustive list of recommended analyses that are available as a result of taking these measures.

3. Waterfall verses Iterative Defect Management

The waterfall life cycle's paradigm of developing requirements completely and correctly before moving on to design and then developing a complete and correct design before coding and then testing allows defects to be defined and managed in a straightforward manner. That is, during any phase, any non-conformance to the previous phase is defined as a defect because artifacts for the previous phase are assumed to be complete and correct. Management of defects in the waterfall lifecycle is a matter of elimination as soon as possible because it is assumed that complete information about the system at any phase is already available from a previous phase.

Recall that software artifacts include requirements and design products and documentation as well as code and test results. Further iterative development disperses learning throughout the development process. (See Section 1) The assumption inherent in an iterative software development lifecycle differ from the waterfall lifecycle in a way that make the definition of a defect less clear but provides more flexibility in their management. Further there is valuable utility in formally

examining and analyzing defects in the iterative lifecycle before/during their removal. Since the iterative lifecycle necessitates that the building of a system begin using incomplete information the developers must learn about the system in process. Thus, the definition of defects is not clear. Consider an artifact that is developed based on the best available knowledge at the time. As new information about the system is learned (in process) that artifact will have to be updated (or corrected). Therefore, the line between a subsequent round of evolution in the system and its software artifacts and the repair of defects in artifacts is blurred. Management of defects in the fluid environment of iterative software development becomes crucial. Fortunately the iterative paradigm offers increased flexibility in many areas, including defect management to deal with this increased fluidity. For example:

- Repair of defects may be deferred to a later iteration, as opposed to being repaired as soon as possible
- Analysis of a defect can provide knowledge about the system to determine whether a newly developed portion of an artifact is in error or an older (or higher level) artifact needs to be corrected/updated.

The need to make correct defect management decision will not only affect the technical quality of the software but programmatic risks such as cost and schedule as well. By formalizing the tracking of an organizations experience with respect to iterative software development phenomena regarding defects though the EoD approach, the software manager is afforded an empirical basis for making and defending his/her decisions.

4. Adjusting “Mini-Waterfalls” to True Iterative Software Development

A strategy that many analysts employ in mapping waterfall style approaches to iterative software development in various domains is to treat the iterations of an iterative effort as miniature successive waterfall scenarios. This mapping often entails numerous informal case-by-case adjustments made based on the analyst’s intuitive understanding and extensive experience in their organization. While these approaches are
successful at times the intuitive adjustments rest with the analyst and not captured as part of the organization’s knowledge base. Further, cross project consistency becomes an issue for any attempts at institutional analysis due to the case-by-case tailoring of the mapping.

The EoD approach is critically deficient with regard to defect analysis due to the direct effects on development and defect occurrence in a given iteration caused by the previous iteration. Thus, while a reasonably consistent set of assumptions between the first iteration and a “mini-waterfall” may sometimes be made, subsequent iteration increasingly deviate from waterfall assumption due to activity and inherited defect issues from previous iterations. Although the notion of inherited portions of a system may be philosophically related to software reuse assumptions the operational result is often quite different. The EoD measurement and analysis approach seeks to overcome these deficiencies by directly targeting the trans-iteration defect phenomena and using the result to a software project’s benefit.

5. Employing Defect Measurement in the Iterative Software Development Lifecycle

The approach of employing defects, as opposed to only removing, is an effort to use discovered defects to a software development project’s advantage by:

- Utilizing defects as a source of in process knowledge acquisition for the current project.
- Referencing historical defect trends for use in quantitative prediction of expected defect for future projects.
- Anticipating defect phenomena based on historical baselines.

5.1. Measurement for Defect Employment

Toward this end the EoD approach advocates a series of necessary measures. (See Table 1) The first measure is the Defect Artifact’s Creation (DAC) in both logical (DACL) and regular (DACR) time. DAC refers to the time when the artifact where the defect is found was created. Thus, DAC will be recorded prior to the discovery of defect. It is the discovery of a defect that triggers the creation of a Defect Record (See Table 1). At this time the pre-existing DACL and DACR values are entered into the record as appropriate. A defect record captures all the information relevant to a given defect for this measurement approach.

Most configuration management (CM) systems record a date of creation and change history that can be used to obtain (DACR). DACL can either be added as an item to record during development or may possibly be derived from CM information. If the CM system contains a data item that can link an artifact to a given iteration, version-numbering protocols are sometimes useful, then breakpoints for phase within an iteration may be derived from the regular-time date of a major review/milestone occurrence that traditional end a given phase for the organization. Thus all artifacts developed during a given phase could then be grouped and identified. Defect Discovery (DD) refers to when
the defect was discovered in logical (DDL) and regular (DDR) time. Similarly, Defect Repair (DR), Defective Artifact's Repair (DAR) refers to when (logical and regular time) the defect repair was complete and when a repair to a given artifact was completed. It is important to note that there may be multiple DARs for one DR. Thus, the DAR times, DARL and DARR, of the last artifact repaired would be equal to DRL and DRR respectively because the final DAR would complete the DR. If a defect’s repair is deferred to a later phase/iteration of development the Defect Deferred To (DDT) fields would reflect the logical (DDTL) and regular (DDTR) times when the defect repair is planned to take place. While DDTL is recorded at the time the decision to defer a defect is made DDTR is derived later as the start of the phase in the future iteration in which DR is planned. If a defect is not deferred then the fields contain a null value. A Learning Defect (LD) refers to a defect that requires artifacts that were developed in previous iterations or phases to be repaired. Thus, the value of the LD field is implicit in that it can be derived from DD and DAC using the rule:

\[
\text{Within a single defect record} \\
\text{if} \\
(\exists (DAC_j) \text{ earlier than DD}_j) \\
\text{then} \\
LD = \text{Yes} \\
\text{else} \\
LD = \text{No}
\]

Similarly, Regular Defect (RD) is also implicit and obeys the rule:

\[
\text{Within a single defect record} \\
\text{if} \\
(\exists (DAC_j) \text{ earlier than DD}_j) \\
\text{then} \\
RD = \text{No} \\
\text{else} \\
RD = \text{Yes}
\]

This rule states that RDs are defects where only artifacts that were developed in the current phase/iteration need be repaired. It is believed that LDs are associated with a higher probability of the existence and number of Related Repairs (RR) than RDs. RR are records of new defects discovered as a result of repairs associated with the defect in the current defect record. Therefore, the RR field(s) would be filled in (added to) after the defect in the current record has been repaired. If/while no RR exist the field contains a null value.

5.2. Analysis for Defect Employment
Recall from section one that, in addition to defect tracking and removal, defect employment seeks to:
- Make use of defects as a source of systematic learning throughout the iterative lifecycle.
- Capture characteristics pertaining to developmental flexibility in JPL’s iterative software development lifecycle and establish statistical rules and thresholds to maximize benefit and minimize dangers.
- Leverage existing technology/models for defect prediction such as the COQUALMO defect prediction model.

The measures discussed in section 5.1 were formulated specifically for these purposes. This section presents opportunities for analysis of the proposed metrics to achieve the stated goal above.

5.2.1. Defects as a Source of Learning
During iterative software development defects discovered in artifacts being developed in the current phase may prompt changes in artifacts developed in previous phases and iterations. This situation may be precipitated by one of two phenomenon 1) the previous artifact is/ was defective despite sufficient knowledge to avoid the defect in question or 2) the previous artifact was developed “correctly” based on information available at the time; information that latter (currently) has been learned to be incomplete or incorrect. In the initial stages of this effort at JPL defects resulting from the situation above will be considered part of one category (LD). Investigation to determine which of the two cases above a caused the defect is not feasible as an in-process activity by developers during development. However discrimination by an independent source or as a postmortem effort can be used as a source of information for root cause and process improvement purposes.

The increased probability that artifacts will be developed based on information that is incomplete, incorrect or not fully understood is an inherent cost of developing software in an iterative paradigm. The result is often the discovery of defects in later
phases/iteration of software development. Knowing this uncertainty exists and planning in order to manage it is necessary for successful development of high quality software within programmatic constraint. The EoD approach offers analysis to aid in managing the uncertainty accompanies development of software iteratively.

Analysis includes the use of the DAC, DD, RR and LD (implicit) measures. The first point for analysis is to examine the relationship between the defective artifact’s time of creation (DAC) and repair (RR) and the time when the defect was discovered (DD). Since a DAC exist for each RR performed in response to a defect discovery and correcting one defect may require multiple RRs, the earliest DAC associated with the defect in question is used for this analysis. Then defects are discriminated by DD (defects in each phase/iteration) where the DAC is earlier that DD implying the defect is an LD. Within the set of LDs, a point system is derived where each development phase between an injection (earliest DAC) and discovery (DD) of a defect is given a value of one. Further an extra point is added to the count whenever an iteration boundary is included in the time span. An iteration boundary exists between the test phase of one iteration and the requirements phase of the next iteration. (See Figure 1) Therefore, high scores in the point system are associated with more development based on a faulty foundation. This in turn raises the likelihood that the repairs will be more expensive and time consuming.

F1:

For each DD = i

\[ \frac{|DD|}{\sum_{j=1}^{n} (LDI_j \ points) / |DD|=i} \]

Graphically the metric for each DD = i forms an

Figure 1: LDI Point System Example

Figure 2: Ideal Average LDI Points Profile

X-axis corresponding to the time progression though the phases and iteration of the software development. The Y-axis represents the value of a metric for each phase that expresses a combination of the number and impact of LDs per phase. The impact of an LD refers to the amount of software development that has occurred between the “insertion” of the defect and its discovery measured here as the number of phases and iteration occurring.

Ideally, one would expect low average LDI points in early phases/iterations because little of the system is developed and thus has a short history of artifacts in which to find RRs. One can then expect the average LDI points to increase dramatically in following phases because details of the system are becoming relevant causing a great deal of learning to take place necessitating adjustments in previous artifacts. Finally, Average LDI points should ideally decrease in late phase/iteration of software development because a good understanding of the system should be in place and remaining defects should be localized defects in new functionality. (See Figure 2) If LDI point averages remain high moving into later phases it is a leading indicator that the current schedule and/or quality may be in jeopardy.

A second analysis relates the LDI points of a defect from the earliest DAC to DD to the number of RRs associated with that defect. The intuitively expected result is that the number of RRs will be positively correlated with LDI points at a high
level. This is because a trend of higher LDI points indicates that more of the system has been built based on defective assumptions from previous phases/iterations. Therefore the number of software repairs as a function of iterations/ phases will start small in early phases/iterations increasing rapidly in middle phases and leveling off in late phases. (See Figure 3) The phenomena of the number of defects stabilizing in late iterations despite decreasing average LDI points is due to the increased discovery of localized defects due to end-of-development testing and “clean-up development”. Clean up development refers to items such as cosmetic enhancement of interfaces, output formatting etc… that tend to produce RDs as opposed to LDs but require repairs none the less.

5.2.2. Defect Management Flexibility Analysis
While defect as a learning source (Section 5.2.1) focuses mainly on the period from defect insertion to defect discovery with repair as a corollary, defect management analyses are more focused on the repair of the software after defect discovery. However, the two issues are so closely related that this organization may be regarded as a philosophical issue more than a technical issue. The analyses associated with defect management and the flexibilities inherent in an iterative software development lifecycle make use of the DD, DR, DAR, DDT, RR, LD (implicit) and DR (implicit) measures. The first two analyses relate the time period from DD to DR accounting for DDT periods to:
- LDs VS RDs
- Number of RRs

The period from DD_R to DR_R represents the traditional aging metric of defects. While useful, this aging fails to account for various flexibilities in defect management during development of software in an iterative manner. Aging of defects in the EoD approach considers the period from DD_L to DR_L to account for various factors such as overlap of phases/iteration, as is common in an iterative environment, non-sequential development of a given functionality to completion or environmental factors such as work stoppages. In conjunction with the factors above, the Adjusted Defect Removal Period (ADRP) from DD_L to DR_L minus the period of defect deferral (date DDT is recorded to DDT_L), if it exists, during the overall interval. Defect deferral represents time in which a defect is purposely not addressed for strategic reasons. Thus, removing it from consideration in some cases gives a more accurate view of defect repair effort/productivity by considering only the period where developers are burdened with the DR task.

Examining average ADRP / defect broken down by LDs and RDs over software development phase/iterations gives insight into productivity of defect repair as the iterative software development lifecycle progress. When combined with a predictive technology for defects such as COQUALMO this information can be used for schedule planning purposes. A number of competing factors contribute to the productivity of defect repair including but not limited to:
- The number/timing of defects discovered which is directly related to the amount of testing, inspection etc…
- The staffing level of personnel dedicated to discovering/repairing defects including shifts in personnel during phases of the development process.
- The policy for deferring defect to later phases/iterations

At this time there is no clear justification for a given expected profile of defect repair productivity at JPL. However, current metrics efforts are, among other goals, attempting to generate a baseline profile and identify critical factors that explain the phenomena in the profile(s).
When factoring the number of RRs into analysis of DR productivity via the ADRP, the insight into DR offers additional insight into LDs. LDs with a high number of RRs represent critical phenomena that greatly inhibit the development of high quality software on time and within budget. Thus, the potential benefits of this type of analysis include determination of software development process weaknesses that offer high leverage for improvement.

Finally, by tracking DARs and their association with defects overtime, a measure of the amount and rate of change a given artifact undergoes may be examined. This allows software development personnel to target problem portions of the software system for:

- Additional scrutiny in process to resolve problems at the earliest possible time.
- Trend analysis over time to guide software managers in planning additional verification activity and resources in areas that are traditionally troublesome for the organization.
- Identify areas for process improvement to preemptively address persistent software development problems.

### 5.2.3. Defect Prediction in the Iterative Lifecycle

In conjunction with the EoD approach and the JPL Software Quality Improvement Project's cost estimation efforts, defect data is being categorized in terms of requirements, design, code and test, along with the COQUALMO drivers, for both the EoD approach and the COQUALMO defect prediction model. [6] The new approach here augments COQUALMO by adding an iterative dimension to the data that COQUALMO currently lacks. The employment of defect approach offers the ability to investigate defects within the iterative development paradigm and COQUALMO offer predictive capabilities. Conversely, COQUALMO offers predictive capabilities that are currently lacking in the EoD approach discussed in this paper.

Future work at JPL will yield a base of defect data that has the potential to relate COQUALMO to JPL's specific iterative phenomena. Defect data collected over JPL past projects' records from integration testing show a significant correlation with COQUALMO's prediction defect introduction on a percentage basis. (See Figures 4 and 5) The combination of defect employment in the iterative lifecycle with COQUALMO defect prediction will allow increased predictability using information that can be more readily estimated during initial stages of project planning. The resulting benefits will include better control over uncertainties pertaining to defect occurrence and management.

### 6. Conclusion

The EoD approach to measuring and analyzing defects identifies trends and phenomena that result from the use of an iterative software development lifecycle. In particular the effects, from a defects
standpoint, from between one development iteration and the next are examined. Trend analysis is used to provide software managers with quantitative data to aid in ascertaining the status of the software during a given iteration as compared to historical experience. Further, in conjunction with predictive models such as COQUALMO, the EoD approach will allow a software manager to anticipate likely defect phenomena (rates, repair effort etc...) based on information gained from the defects themselves in early software development iterations.

7. Acknowledgement

The research described in this abstract was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

8. References


9. Biographies

John D. Powell holds a M.S. in Computer Science from West Virginia University and is a software quality assurance researcher at the California Institute of Technology's Jet Propulsion Laboratory (JPL) in the Quality Assurance office. Currently he performs research in the area of Quality/Cost Estimation and Prediction as well as Formal Methods research for efforts at JPL. Prior to his work at JPL/USC, John worked as a System Software IV&V Analyst for NASA's prime IV&V contractor (Titan-Averstar) performing IV&V analysis on the Redundancy Management and Control systems for the Space Shuttle's Checkout Launch Control System (CLCS). Prior to that, at the NASA Goddard IV&V Facility, John performed research under the Intelligent Systems Initiative exploring alternatives to traditional model checking in conjunction with West Virginia University's Software Research Laboratory (SRL). His publications include a master thesis, papers at ICSE, Trans. on SE, CSEE&T, ISRE, IEEE WETICE, ISPA and various NASA Conferences/Workshops and technical reports.

John N. Spagnuolo Jr. is a Senior Staff member at the Jet Propulsion Laboratory / California Institute of Technology in mathematics / algorithm research. He graduated from Clarkson College of Technology and subsequently obtained an MA from the University of California at Los Angeles. After that he received a PhD from Rensselaer Polytechnic Institute in Mathematics. While at JPL, Dr. Spagnuolo has done work in artificial intelligence, parallel computation (Time Warp), coding theory, sensor modeling, simulated annealing, look-ahead planning, computer directed battle management, expert systems, neural networks, clustering and solar physics. He has won numerous NASA Tech Brief Awards for Significant Contributions and Publications and 2 Space Act Awards for innovation in technology. His present research interests include the merging of Kantian philosophical inquiries into the nature of human thought with computer graphics, neural networks and other requisite mathematical formalisms to simulate the reasoning process on machines.