Project LEAP: Lightweight, Empirical, Anti-measurement dysfunction, and Portable Software Developer Improvement

Philip Johnson

Department of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
(808) 956-3489
(808) 956-3548 (fax)
johnson@hawaii.edu

April 30, 1998

Abstract

Project LEAP investigates the use of lightweight, empirical, anti-measurement dysfunction, and portable approaches to software developer improvement. A lightweight method involves a minimum of process constraints, is relatively easy to learn, is amenable to integration with existing methods and tools, and requires only minimal management investment and commitment. An empirical method supports measurements that can lead to improvements in the software developers skill. Measurement dysfunction refers to the possibility of measurements being used against the programmer, so the method must take care to collect and manipulate measurements in a "safe" manner. A portable method is one that can be applied by the developer across projects, organizations, and companies during her career.

Project LEAP will investigate the strengths and weaknesses of this approach to software developer improvement in a number of ways. First, it will enhance and evaluate a LEAP-compliant toolset and method for defect entry and analysis. Second, it will use LEAP-compliant tools to explore the quality of empirical data collected by the Personal Software Process. Third, it will collect data from industrial evaluation of the toolkit and method. Fourth, it will create component-based versions of LEAP-compliant tools for defect and time collection and analysis that can be integrated with other software development environment software. Finally, Project LEAP will sponsor a web site providing distance learning materials to support education of software developers in empirically guided software process improvement. The web site will also support distribution and feedback of Project LEAP itself.

1 Motivation

This proposal for Project LEAP arises from our experiences with research and practice of formal technical review methods, affiliations with industry software quality improvement groups, and research and practice of individual improvement methods such as the Personal Software Process [9]. These experiences lead us to the following observations:

FTR best practice is "heavyweight" and difficult to adopt. Current industry best practice with respect to formal technical review emphasizes management commitment, extensive training, detailed measurement, structured data entry forms, prescribed roles for participants, and precisely specified and constrained phases and constraints on interactions [4, 5, 20, 21]. As a concrete example, industry best practice suggests that review meetings should focus on issue recording and not "digress" into issue resolution. We term such precisely specified and constrained methods "heavyweight", since they require a large and ongoing investment of time, energy, and oversight of both developers and managers to adopt as well as to maintain over time.

Our close affiliation with SQA leaders at many industrial organizations, interviews with industry managers and developers, and first-hand observation of industrial review practice have convinced us of two things: (1) successful adoption of current review best practice does improve the efficiency and effectiveness of review, and (2) current review best practice is tremendously difficult to successfully adopt. In addition to our own observation, we conducted an informal poll using the internet newsgroup comp.software-eng requesting information on review practice from the respondant's organization. Although the results are not statistically meaningful, it is still interesting to note that approximately 80% of the respondants either did not use formal review practice or had tried and abandoned it.

Software review is susceptible to measurement dysfunction. Recent research by Robert Austin investigates the phenomenon in software development organizations of "measurement dysfunction": a situation in which the act of measurement affects the organization in a counter-productive fashion, and which leads to results directly counter to those intended by the organization for the measurement [1]. Such dysfunction occurs because many process measures have two potential applications: (1) to provide *information* about the organization and (2) to support *performance evaluation* of individuals and groups. It is impossible for an organization to guarantee that a measure, once collected by the organization, will never be used for performance evaluation. Thus, individuals in an organization may tend to operate under the assumption that any measure may eventually be used for performance evaluation, regardless of the stated intention of the organization with respect to that measure at the time it is collected. Austin's research provides examples of how this awareness can result in individual and group behavior that produces good values of the measure for performance evaluation, but undermines the measure's value for process information and improvement.

Although most FTR training materials caution against the use of review measures for individual performance evaluation [5, 4, 20], Austin's research reveals that the *potential* for such use is sufficient to induce measurement dysfunction, regardless of whether the organization ever actually uses the measures in that fashion. Our involvement with industrial development groups have provided us with first-hand experiences of measurement dysfunction. For example, at one company, defects discovered during design review are routinely classified as "enhancements", because recording the

defects as defects could cause them to miss an upcoming milestone, with implications for subsequent performance evaluations.

Software review measurements are extremely susceptible to measurement dysfunction. We have catalogued many forms of FTR measurement dysfunction induced to satisfy organizationally mandated goals [10]. Selected examples include: "defect severity inflation", where minor defects are reclassified as major to improve review effectiveness measures; "artificial defect closure", where defects are either reclassified as enhancements or reclassified with a lower severity level to improve defect closure rates; "preparation time inflation", where reviewer preparation time is inflated to improve review quality measures; "defect discovery rate inflation", where preparation time is deflated to improve review efficiency measures, and "defect duplication inflation", where reviewers compare notes and submit duplicate defects to improve their personal defect discovery efficiency.

Software review technology neglects downstream support. A variety of review support tools have been developed, including: ICICLE [2, 19], Scrutiny [8, 6, 7], ASSIST [13], Hypercode [18, 17], INSPEQ [11, 12], CAIS [14, 15], review tools based upon electronic meeting rooms, and our own tool CSRS. All of them, including CSRS, focus on "upstream" review issues—issues involving the conduct or disposition of a single review—such as definition of review processes, hypertext annotation of the work product, voting on issue disposition, and synchronous or asynchronous communication among developers as they carry out preparation or review meeting activities.

None of these tools provide explicit support for "downstream" review issues—issues involving the analysis of results from multiple reviews in order to better understand the impact of reviews on the development process and to discover potential process improvements. Although researchers may argue that such analysis is more properly done as part of general process improvement analysis and activities, our own industry experience leads us to conclude that if review technology does not incorporate explicit, easy to use mechanisms for analysis of review data, that analysis will simply not be done. We know of one industry site where over five years of detailed review data has never been touched by analysis.

Top-down process improvement is difficult to adopt. Top-down process improvement initiatives are those resulting from a high level management decision to invest time, resources, and training to improve development practice through some combination of changes to the organization's structure, staff, practices, and/or development environments. Examples of top-down process improvement include Motorola's "Six Sigma" initiative and the Capability Maturity Model [16].

While successes of these initiatives are widely publicized, problems with their adoption are becoming more visible as well. For example, Yourdon exhorts software project managers to rebel against the "Methodology", "Metrics" and "Process police" in their organizations [22]. Tom DeMarco likewise portrays top-down initiatives like the Capability Maturity Model in an unfavorable light [3].

One structural problem with many top-down process improvement initiatives is that the direct benefit to a given individual may be deferred, missing, or even negative. For example, design staff might be required by management to insert formal technical reviews into their schedule because reviews have been shown to decrease test time, yet the design staff is not provided extra schedule time or resources for review (because time-to-market is critical and management wants to see cod-

ing begin as soon as possible). As a result, individual designers incur new direct costs while the major direct benefits accrue to the managers and testers. As another example, top-down initiatives may require staff to fill out additional forms so that the organization can assess process compliance, effectiveness, and efficiency. By the time this data is analyzed, the staff who collected the data have been reorganized and no longer a part of the group affected by the analysis. Indeed, given the current level of churning in the industry, many developers can reasonably expect to have changed employers before process improvements take effect. Finally, top-down process improvement data, like review data, is subject to measurement dysfunction.

Bottom-up process improvement has many benefits. An attractive alternative to top-down process improvement is bottom-up process improvement, best exemplified by the Personal Software Process developed by Watts Humphrey [9]. In the Personal Software Process, developers carefully collect data on the defects that they make and the time they spend in each phase of development. Over time, developers build a "personal database" of measurements that they can use to support design, planning, coding, testing, and process improvement activities. Humphrey's text provides an incremental introduction to personal software process improvement that, with effort, can be accomplished in one semester. He has collected data from his students at Carnegie-Mellon as well as from industry sites that appears to show significant improvements in software quality and project planning accuracy resulting from use of the PSP.

Bottom-up process improvement, such as the PSP, has many advantages over top-down initiatives.. First, a "PSP-enabled" developer does not need management approval or resources to maintain PSP data; it is simply a part of her daily development practice. Second, the investment made by an individual into PSP data collection and analysis directly benefits that individual. Third, PSP data is "portable": a developer can take PSP data with her as she moves within or across organizations. Fourth, PSP data is not normally susceptible to measurement dysfunction since the organization does not normally have access to PSP data.

We now have two years of direct experience using PSP both in a classroom setting, where the principal investigator has taught PSP for three semesters, and in a research setting, where PSP methods have been adapted to support not only our software development activities but student thesis development as well. Our results are mixed. On the one hand, we have found that bottom-up process improvement does have the clear advantages over top-down initiatives noted above. Evaluations of the course and the PSP methods by students have been overwhelmingly positive. Our student data appears, at first glance, to confirm all the claims made for the PSP. On the other hand, after gaining experience with teaching and use of the PSP, we started to become suspicious about the quality of PSP data. Did the data and analyses really reflect the practice?

Bottom-up process improvement can suffer from poor data quality. To investigate this issue, we conducted a case study during a recent semester while teaching PSP. Prior to the semester, and based upon our experiences teaching PSP, we implemented three extensions to the standard PSP curriculum in order to improve data quality. First, we added formal technical reviews and checklists to each project. Prior to data submittal, all PSP data was subject to a thirty minute group review. The review followed a checklist (which was made available at the beginning of the assignment) to check whether data and calculations were accurate. Second, we distributed customized spreadsheets to each student to support data analysis, and Java applications to calculate software size. Third, we

added new forms to the process to clarify problem areas (such as selecting the correct method for size estimation) found in previous semesters.

During this semester we collected 90 PSP datasets from the students: 9 projects from 10 students. After the semester, we began a detailed retrospective analysis. We implemented a database system to cross-check all calculations and expose six classes of data defects. Next, all individual data values from the 90 PSP datasets were entered into the system. Finally, we ran our checks over the system. Given our level of commitment to high data quality during the semester, and the inclusion of additional procedures designed specifically to support high data quality, we were astounded by the results of this analysis.

Frequency	**Type**
699	Incorrect calculation
262	Missing field
175	Transfer of data between projects
146	Entry error
100	Transfer of data within a project
88	Impossible values

Figure 1: *Summary statistics for error frequency and type in a PSP class*

As Figure 1 illustrates, we found that almost 1500 errors in data collection or manipulation were made during the case study semester. (Our values are conservative, because there are other important classes of errors, such as failing to log a defect on the form, which we could not detect through our analyses.) Furthermore, we did not see a sharp drop-off in errors as the semester proceeded, which might indicate that the errors were mostly an artifact of the learning process and would not persist into the student's professional environment. The errors were distributed between analysis and recording and across all students in the class.

Although this data is preliminary and our analyses are still ongoing, this study clearly calls into question the quality of bottom-up process data, at least when collected and analyzed in the manual fashion used by the PSP. Although we are sure that engineers do exist who can flawlessly record, analyze, and interpret their personal process data in a manual fashion, we believe that the required time, energy, and attention to detail imposes severe burdens on more typical engineering staff. We conjecture that a radical simplification of data collection and analysis must occur before personal process data can be effectively and accurately manipulated by a large segment of the development community.

2 Proposed Research

As discussed above, our prior research raises concerns regarding heavyweight review methods, the possibility of measurement dysfunction, the lack of downstream analysis support, the need for extensive management commitment to and investment in top-down process improvement, and the data quality problems present in otherwise promising bottom-up process improvement approaches

like the PSP. The four principal requirements underlying Project LEAP's approach to software developer improvement derive directly from these concerns.

- **L**ightweight. Project LEAP methods must be "lightweight". In other words, they must involve a minimum of process constraints, be easy to learn, easy to integrate with existing methods and tools, and require minimal investment and commitment from management. If a LEAP method imposes new overhead on the developer, then that effort must yield a short-term, positive return-on-investment to that same developer.

- **E**mpirical. Project LEAP must have a quantitative, as well as qualitative dimension. A lightweight orientation cannot be gained at the expense of high quality collection and analysis of data. Developer improvement should be observable over time through measurements of effort, defects, size, and so forth.

- **A**nti-measurement dysfunction. Measurement, while an integral part of Project LEAP, must be carefully designed to minimize dysfunction. Yet the most simple solution to dysfunction, making all data totally private, is incompatible with the benefits to the organization of sharing certain kinds of data and process improvements. Project LEAP is designed to find a suitable balance between totally public and totally private measurement data.

- **P**ortable. Useful developer improvement support cannot be locked to a particular organization such that the developer must "give up" the data and tools when they leave the organization. Rather, this support should be akin to a developer's address book; a kind of "personal information assistant" for their software engineering skill set that they can take with them across projects and companies.

These four criteria, when composed together, create additional requirements. For example, we hypothesize that extensive automation is required within any method that is simultaneously lightweight, empirical, and anti-measurement dysfunction. On the other hand, automation clearly does not guarantee lightweight processes or meaningful empirical evidence of improvement. As an example, a repeated criticism of the CSRS automated review system was that its extensive measurement system would lead to dysfunctional behavior in an industrial setting.

These four criteria, when composed together, preclude existing approaches to bottom-up or top-down process and/or software quality improvement. Approaches such as CMM or ISO 9000 are empirical, but not lightweight or portable and they are prone to measurement dysfunction. The PSP is portable and not susceptible to measurement dysfunction since there is no necessity that PSP data be shared. (In fact, Humphrey recommends against sharing PSP data, at least initially.) As the results of our case study revealed, however, PSP data is highly susceptible to low data quality, and our own experience teaching and using it indicate that it is most definitely not lightweight. Finally, formal technical review is empirical, but neither lightweight, anti-measurement dysfunction, nor portable.

The above discussion may leave the impression that satisfying the LEAP criteria is impossible. Actually, we conjecture that many LEAP-compliant methods for software developer improvement can be designed, once these requirements and their implications are understood. The next section presents an overview of one such method and associated tool support for defect entry and analysis under active development in our research group. It represents an appealing synthesis of our previous research on Formal Technical Review and the Personal Software Process.

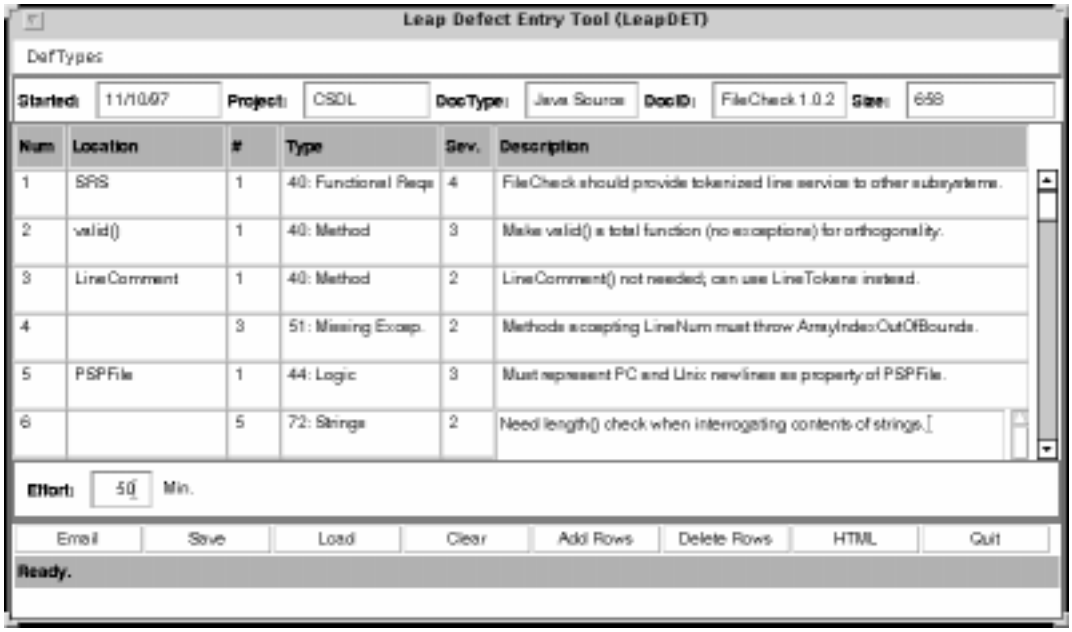

Figure 2: *A LEAP-compliant defect entry tool. The system enables reviewers to efficiently raise issues about work products, distribute the results to other members of the review team, and save the data for later empirical analysis using the Leap Defect Analysis Tool.*

A LEAP-compliant method and toolset for Defect Entry and Analysis

A fundamental premise of both formal technical review and the Personal Software Process is that collection and analysis of defect data can provide useful insight into software and process quality. In FTR, defect data is generated through the combined efforts of a group of developers, using a structured process to generate a composite defect report which the author uses as the basis for rework activities. In the PSP, defect data is primarily recorded by the author as she discovers defects in her own work products. In the past, we have collected defect data through FTR and PSP and analyzed the data separately according to these two paradigms. Currently, we are deploying an early release LEAP-compliant method and toolset within our research group that supports both of these modes. This toolset is written in Java using COTS components for tables and graphing and runs on all major platforms. The toolset supports a simple four stage method for defect processing: (1) Entry, (2) Distribution, (3) Rework, and (4) Analysis.

In our example method, two tools are used to radically simplify most aspects of defect collection, analysis, and their application to software developer improvement, whether the user is participating in a group review or in PSP-style individual defect collection. The first tool is called LeapDet (for LEAP Defect Entry Tool). Figure 2 shows a screen image from this tool. The second tool is called LeapDat (for LEAP Defect Analysis Tool), and is illustrated in Figure 4.[1]

LeapDet supports review of arbitrary document types by focusing on high efficiency entry and distribution of structured defect data between members of a group. To support as wide a range of formal and informal review methods as possible, LeapDet makes no assumptions about the

[1] Although this toolset is still in an early stage of development, it is functional and has already been installed at one industrial site for evaluation. Though not yet mature enough for public release, interested readers of this proposal are welcome to examine the toolset, installation instructions, and user guide at ftp://ftp.ics.hawaii.edu/pub/csdl/leap.

```
!         NAME            COMMENT
Project: "CSDL"           "CSDL research projects"
!         NAME            COMMENT
DocType: "Java Source"    "Java source code"
!         DOCTYPE         NAME                    COMMENT
DefType: "Java Source"    "10: Syntax"            "General syntax errors"
!         NAME                SIZE   COMMENT              DOCTYPE          PROJECT
DocID:  "FileCheck 1.0.2"     658   "File Syntax Checker" "Java Source"    "CSDL"
```

Figure 3: *Example toolset declarations. Lines beginning with '!' are comments and are included as documentation for the fields in the next line's declaration. These declarations are used to improve the efficiency and effectiveness of both the Leap Defect Entry tool and the Leap Defect Analysis Tool.*

actual review process in use in the organization. All fields are optional, and the tool can be easily configured to provide the defect entry data fields required by a given organization.

Rather than focusing on process representation as is done in other review tools, this toolset concentrates on support for defect representation. Upon invocation, both tools read configuration files that declare the projects, document types, defect types, and document IDs currently defined within the organization. Figure 3 shows examples of the four major declarations provided in this toolset. (Our sample leap definition file comes with example declarations for six project types, over 100 defect types, and several Project and Document IDs.)

While using the toolset, individuals and development groups incrementally build and refine this representation of the set of projects they have worked on, the types of documents they have produced, the types of defects that have occurred for each document type, and the work product instances for which defects have been detected.

Explicit declaration of projects, defect types, document types, and document IDs facilitate review and process improvement in three important ways. First, it enables each reviewer to work more efficiently. Information about the document under review (the top line in Figure 2) is automatically specified via a common .config file provided at the start of review. In addition, pop-up menus allow users to mouse-select structured information such as defect types rather than type them by hand, although users are always free to invent new defect types on the fly. The goal is to minimize or eliminate manual entry of information whenever possible. Second, explicit representation of project, document type, and defect type information facilitates consistent use of this information among developers. A development group will typically maintain one or more public Leapdef files with common or group-wide definitions of projects, document types, and defect types, while still allowing individual to add personal extensions through additional private Leapdef files. Third, the declarations support downstream analysis. Effective defect data analysis requires the ability to categorize defects based upon the document type and defect type. Explicit declaration enables users to effectively analyze data across projects and from different reviewers.

As groups and users evolve, so too will the set of projects, defect types, document types, and document IDs. Clearly, the addition of new declarations poses no problem to the toolset. However, users will also need to modify existing declarations, such as splitting one document type ("Java Source") into subtypes ("Java GUI Source", "Java Network Source", "Java File I/O Source", etc.).

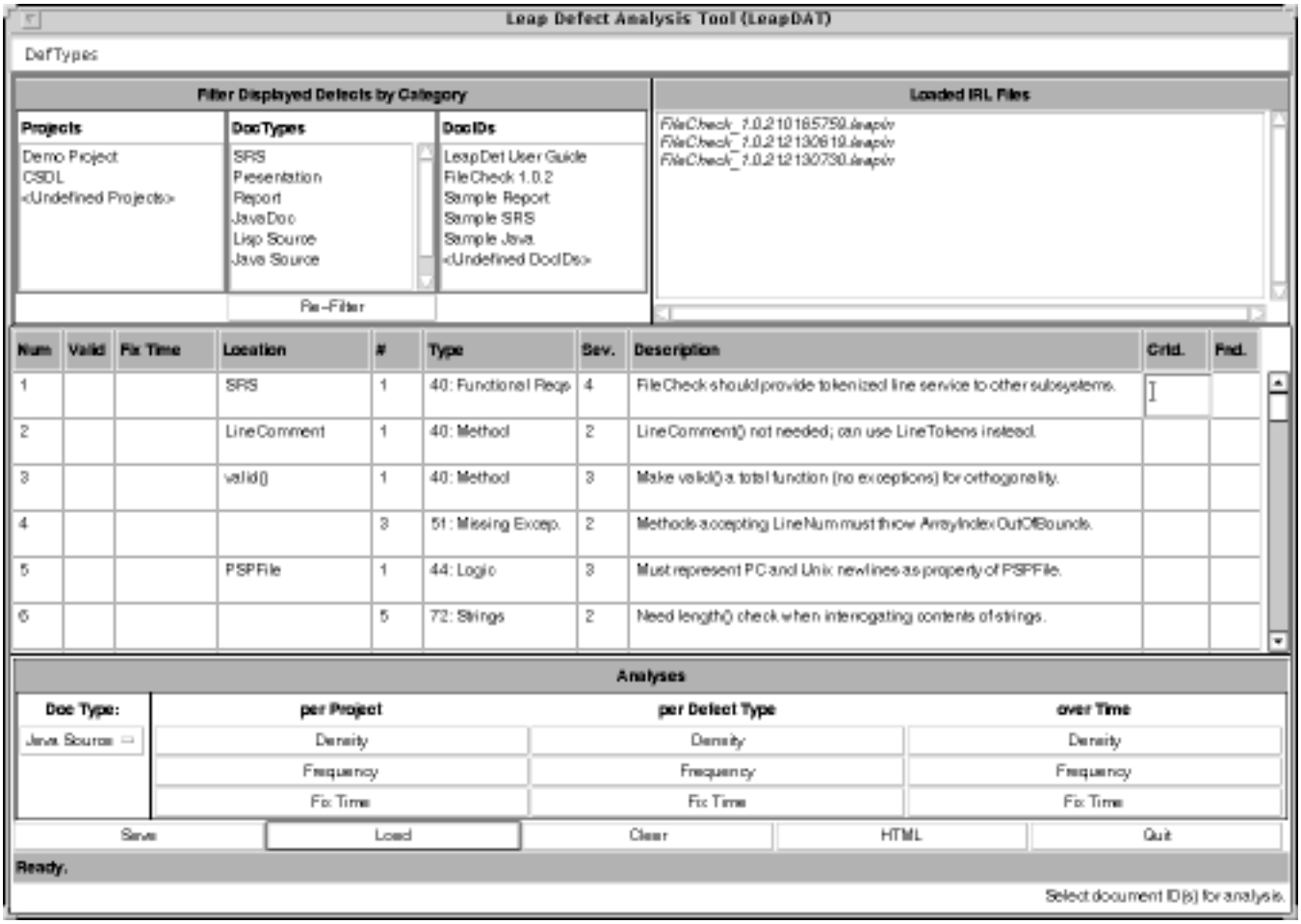

Figure 4: *A LEAP-compliant defect analysis tool. This tool allows a work product author to display, sort, and annotate review comments sent via email from other reviewers. Analysis options allow developers to enter defect data gathered over multiple projects and perform various analyses to support software developer improvement.*

We have found that the set of defect types is particularly volatile when a user first begins collecting this information, and major reorganizations occur frequently. The toolset is designed to accommodate such evolution. While the declared types facilitate organization and analysis, it is not syntactically or semantically illegal for non-declared types to occur within data files. In addition, filtering functions allow users to find those defects with obsolete type information and update them to the current set of declarations. More sophisticated support for evolution is possible, but we have not yet found it necessary.

Efficient entry of defect information is the first step toward LEAP compliance. The second step is efficient distribution. In LeapDet, users can quickly email defect commentary to others at any time via a button which brings up a dialog box requesting the To: field and a brief comment. The defects entered are then sent to all the recipients as a MIME-encoded file which can be decoded by any standard mailer. By default, LeapDet automatically assigns a file name consisting of the document ID followed by an eight character time-stamp (DDHHMMSS). The time-stamped filename minimizes the possibility of duplicate file names, while still keeping them together within a

directory. So that users can easily distribute an updated version that is intended to replace a previous version, this time-stamped filename is generated the first time a set of defect data is emailed or saved to a file and then persists through all subsequent updates.

The design of this mechanism was influenced by our previous experience in CSRS, where we implemented a much more sophisticated, heavyweight, object-oriented persistent store with socket-based notification services. We found that the sophistication of data storage and distribution posed a barrier to industrial adoption of this technology. We expect that the current file and email-based approach to storage and distribution will remove this barrier to adoption.

After defects have been entered and distributed, the immediate next step is to collate together all of the individual defect files, examine the resulting aggregate set of defects, and use the combined data to improve the work product. The LeapDat (LEAP Defect Analysis Tool) supports this process. LeapDat allows the user to load multiple defect files and sort them according to location, severity, number of occurrences, severity, and type. Defect data can be added, deleted, or modified at this time. New fields are provided that enable the author to add information about the validity and fix time associated with each defect, if desired.

Figure 4 illustrates LeapDat after the user has loaded in three files containing defect data sent by reviewers. The LeapDat interface is divided into three sections. The top section provides a listing of all currently declared projects, document types, and document IDs. It also lists all of the files containing defect data currently loaded into the system. Since the amount of defect data maintained by a developer can be large, this top section allows the user to filter the defect data displayed by selecting one or more projects, document types, and document IDs to display. The middle section contains a table-based interface to individual defect entries. The filtered defect data appearing in this table can be sorted by type, severity, fix time, location, or other attributes.

This toolset is designed to improve the efficiency and effectiveness of both review methods with meetings and those without meetings. If a review meeting is used, then LeapDat can be used to interactively display the results of preparation using a projector. The moderator can sort the defects collected prior to the meeting by importance or by location and guide the meeting discussion appropriately. The scribe can enter changes or new defects discovered during the meeting directly into LeapDat. The outcome of the meeting can be distributed either electronically or as an HTML document generated automatically via LeapDat. If a review meeting is not used, then the author alone would use LeapDat to organize the review results.

The bottom section of the LeapDat interface provides a set of built-in analysis mechanisms for historical defect data. To our knowledge, no other review tool exists with this explicit support for analyzing historical data. While we intend this toolset to add value to conventional review processes by improving the efficiency of defect data entry, distribution, and rework, a more fundamental benefit from the standpoint of this research is this support for analysis of defect data, both in aggregate and as trends over time.

The current toolset is designed to analyze three basic measures: defect density, defect frequency, and fix time and provide data about these measures over a series of projects, stratified by defect type, and as trends over time. Figure 5 shows the analysis obtained by pressing the LeapDat button marked “Frequency” in the “per Defect Type” column for one user. This chart shows that two defect types, File I/O and String, account for most of the errors found in documents of type “Java Source” by this user. Armed with this knowledge, the user employs the filtering functions of LeapDat to display only those defect entries and look for process improvements that can reduce the frequency of these defects in the future. For example, if further examination revealed to the user that end-of-

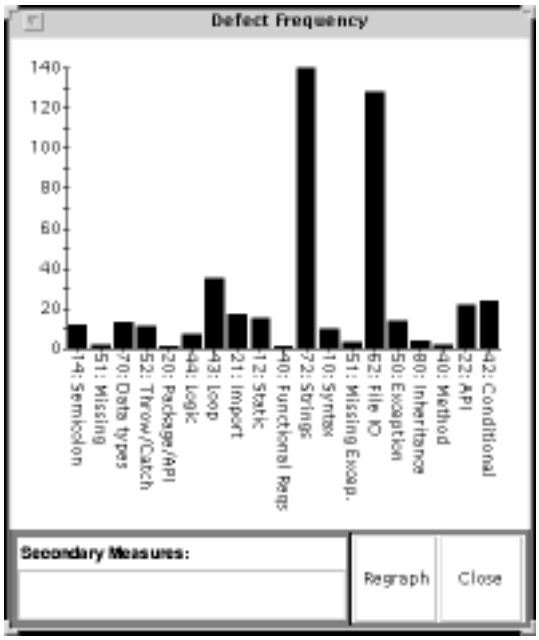

Figure 5: *A defect data analysis, showing defect frequencies for one document type (Java Source), categorized by defect type. Two defect types, File I/O and String, account for the vast majority of this user's defects.*

file errors occur frequently, then a possible remedy would be the use of a checklist that focuses her attention on this issue. Perhaps the user discovers that the erroneous use of the "==" with string objects causes many errors. In this case, the introduction of a lint-type tool might improve the developer's efficiency.

One final LeapDat analysis feature is worth noting. We encourage users to distribute work product defect data not only to the work product authors, but also to the other reviewers. If this is done, then reviewers can compare the defect data generated by their own review to the defect data generated by other reviewers, and determine what defects they missed during the review. Such feedback can help reviewers improve their own reviewing capability by exposing "blind spots". The feedback also indicates likely defects the reviewer might make when they create work products of that type, and thus support developer as well as reviewer improvement. No other review technology includes this empirical support for reviewer improvement based upon comparative analysis of defect data.

To conclude this example of LEAP-compliant support, we return to the four principles and show how this example toolset and method satisfy them:

- **L**ightweight. This method and toolset is designed to be easily adopted by any individuals or groups who have an interest in defect entry, distribution, rework, or analysis. On-line, context sensitive help enables "just-in-time" learning for most of the tool's capabilities, and initial training and installation is minimal. The tools and methods complement existing approaches to review and defect collection, so users can leverage any prior training in review or PSP, though this training is not a prerequisite to effective use of the tool. Finally, extensive buy-in from either management or even fellow development team members is not necessary for

individuals to obtain value from the tool. In the worst case, a developer could generate defect data through self-review.

- **E**mpirical. This method and toolset is clearly empirical in nature, collecting data on defect types, their frequency, and other properties and providing built-in analyses. As analysis is entirely automated, the quality of data should be reasonably high. Of course, the quality of analysis ultimately depends upon the completeness of the data. This motivates the toolset's focus on simplifying defect entry, storage, and retrieval.

- **A**nti-measurement dysfunction. This method and toolset explores a middle ground between PSP data, which is typically completely private (and thus not prone to measurement dysfunction), and FTR data, which is typically collected and analyzed by management (and is thus quite prone to measurement dysfunction [10]). In this toolset, all defect data is distributed *anonymously*: once the file is decoded and separated from its email wrapper, no more identifying information about its author exists.[2]

  As an additional precaution, although LeapDet enables reviewers to provide information about review time, this time-related information is omitted from the emailed data file. We have found that effort data is so susceptible to dysfunction that even sending anonymous effort data is too risky.

- **P**ortable. This method and toolset is intended to be portable on several levels. First, it provides portability through its choice of Java as the implementation language, so it runs on all platforms. More interestingly, the use of declarations and data types allow users to take data with them as they move across organizations within a company or across companies. Finally, this feature enables the methods and tools to be useful for an individual working at multiple companies at the same time under a contractual or consultant arrangement.

3 Anticipated Results

The previous section illustrated that LEAP-compliant tools and methods can be designed by presenting a description of an early version of a toolset and method for defect analysis. In this section, we detail the research directions we will pursue during the course of Project LEAP to gain greater insight into the costs and benefits of the approach. We expect Project LEAP to contribute to the current state of knowledge through the following five major results:

Result 1: Validation of PSP data quality findings

Our current data strongly indicates that manual approaches to PSP data collection lead to serious data quality problems. Our preliminary analysis also indicates that appropriate automated support, similar to the Leap toolset, could dramatically improve data quality.

[2] This safeguard is not perfect. It is still possible for management to decode the file once obtained, go into LeapDat, and add an annotation containing the author name to the description field of every single defect. By repeating this for every single defect, one could eventually build a profile of individual developer defects. However, this effort is large enough that virtually any manager would simply ask developers for copies of their analyses. At that point, if necessary, developers could even generate a "cooked book" with dysfunctional defect data to satisfy management while still retaining an accurate copy for their own records.

The first result of Project LEAP will be a thorough experimental investigation of data quality issues in the PSP. To accomplish this, we will conduct a controlled experiment over four semesters, where our current data counts for semester 1. During Spring 1998, we will teach the PSP in an advanced software engineering class and replicate the current study's format, data collection, and analysis. During Fall of 1998 and Spring of 1999, we will teach the class twice more, this time using appropriate designed LEAP tool support for defect and time data collection and analysis. We will then analyze our four datasets for differences in data quality over the four semesters and for differences when the semesters are combined into "manual" and "automated" treatments. Of course, we will also be attentive to unexpected side-effects of the introduction of automated technology.

As we progress with this research, we will make the experimental design, instruments, and protocols for the experiment available to our ISERN (Internation Software Engineering Research Network) partners and to the entire software engineering community. This could lead to additional replication of the study, improving the generality of the results.

Result 2: Industry case studies of LEAP adoption

A primary finding from our ongoing industrial collaborations is that software review best practice and software review technology are quite difficult to adopt. Project LEAP is our response to this problem. We hypothesize that lightweight, empirical, anti-measurement dysfunction, and portable methods should be substantially easier to deploy within an organization.

We will test this hypothesis through case studies at various industrial sites. Currently, we have commitments from development groups at Tektronix, Inc. and Digital Equipment Corporation to evaluate this technology. (Indeed, a preliminary version of the toolset was already installed and evaluated at Tektronix in early November, 1997.)

Clearly, the design of this case study must be carefully chosen so as to not re-introduce the very measurement dysfunction we have tried so hard to prevent! Therefore, although it is tempting, we cannot ask developers to provide us with their LEAP data files. We will, however, conduct structured interviews with developers at regular intervals during the study to evaluate the use of the tool. With the developers' permission, we may be able to obtain general empirical data, such as the number of LEAP data files they have saved. However, we will rely primarily on qualitative data for insights on the strengths and weaknesses of the LEAP methodology and to enable us to partially test our hypothesis.

Result 3: Component-based infrastructure for LEAP developer improvement

The current toolset is implemented as a suite of Java applications. Over the course of the study, we will continue to evolve and improve the initial set of functionality demonstrated in this proposal. We will also extend the toolset to support collection and analysis of time-related data.

As the toolset functionality begins to stabilize, we will first investigate transition away from an application-based implementation and toward a browser-based implementation. The browser-based implementation can simplify installation of the tool and support training activities at the educational web site, to be discussed further below.

Our second investigation will be the decomposition of LEAP tools into building blocks that can be integrated into other systems using component technology such as Java Beans. The ultimate goal is to provide infrastructure that enables integration of LEAP technology with any other element of

the software development environment, from code editors to verification tools to GUI interface builders to change control systems.

As we did with the Egret and CSRS systems, we will make high quality, well-documented releases of all of these technologies freely available to the software engineering community through the Internet.

Result 4: Distance learning resources

Although the LEAP tools are designed to be easy to adopt, they do not eliminate all of the complexity associated with empirical software process improvement. Once developers begin to collect data, they will start to have questions such as, "What kinds of valid inferences can I make based upon this data?", or "How should I collect future data in such a way as to test the hypothesis I have generated from analysis of the current data?" To support developers in this process of empirically guided self-discovery, we will develop a distance learning program that will discuss these concepts. We will employ browser-based versions of the LEAP tools as an integral part of the learning process. For example, we can provide the user with a browser interface to the Leap defect analysis tool that is pre-loaded with defect data and use this data set to guide them through various empirical analyses.

We do not expect all of the information transmission to be one-way; rather we expect the LEAP user community to contribute to this on-line resource as well. For example, users can contribute defect type declaration sets for new or existing document types. Users can also contribute checklists, review methods integrating LEAP technology, or other process improvements they have found useful.

Result 5: Standards for defect representation and analysis

Finally, the LEAP toolset embodies a representation for defect and other process representation and a set of associated analyses. We intend to work with other groups to help develop a standard representation for this information so that all tools manipulating this data can freely interoperate.

4 References Cited

[1] Robert D. Austin. *Measuring and managing performance in organizations*. Dorset House, 1996.

[2] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: Groupware for code inspection. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work*, pages 169–181, October 1990.

[3] Tom DeMarco. *The Deadline*. Dorset House, 1997.

[4] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews*. Little, Brown, 4th edition, 1990.

[5] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[6] John W. Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney, and Gerard Memmi. Scrutiny: A collaborative inspection and review system. In *Proceedings of the Fourth European Software Engineering Conference*, Garwisch-Partenkirchen, Germany, September 1993.

[7] John W. Gintell and Roland F. McKenney. A proposed structured collaboration architecture derived from Scrutiny project. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work: Workshop of Software Architectures for Cooperative Systems*, Chapel Hill, North Carolina, October 1994.

[8] John W. Gintell and Gerard Memmi. CIA: Collaborative Inspection Agent. In *Proceedings of the CSCW'92 Workshop on Tools and Technologies*, September 1992.

[9] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, January 1995.

[10] Philip M. Johnson. Measurement dysfunction in formal technical review. Technical Report ICS-TR-96-16, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, 1996.

[11] John C. Knight and Ethella Ann Myers. Phased inspections and their implementation. *Software Engineering Notes*, 16(3):29–35, July 1991.

[12] John C. Knight and Ethella Ann Myers. An improved inspection technique. *Communications of the ACM*, 11(11):51–61, November 1993.

[13] Fraser Macdonald. ASSIST v1.1 user manual. Technical Report EFoCS-22-96, Department of Computer Science, University of Strathclyde, UK, February 1997.

[14] Vahid Mashayekhi, Janet Drake, Wei-Tek Tsai, and John Riedl. Distributed, collaborative software inspection. *IEEE Software*, 10(5), September 1993.

[15] Vahid Mashayekhi, Chris Feuller, and John Riedl. CAIS: Collaborative asynchronous inspection of software. *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, 19(5):21–34, December 1994.

[16] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1995.

[17] James M. Perpich, Dewayne E. Perry, Adam A. Porter, Lawrence G. Votta, and Michael W. Wade. Anywhere, anytime code inspections: Using the web to remove inspection bottlenecks in large-scale software development. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 14–21, Boston, MA., May 1997.

[18] Dewayne E. Perry, Adam Porter, and Lawrence G. Votta. Evaluating workflow and process automation in wide-area software development. In *Proceedings of the Fifth European Workshop on Software Process Technology*, October 1996.

[19] V. Sembugamoorthy and L. Brothers. ICICLE: Intelligent code inspection in a C language environment. *The 14th Annual Computer Software and Applications Conference (COMPSAC '90)*, October 1990.

[20] Susan H. Strauss and Robert G. Ebenau. *Software Inspection Process*. McGraw-Hill, 1994.

[21] David A. Wheeler, Bill Brykczynski, and Reginald Meeson. *Software Inspection: An Industry Best Practice*. IEEE Computer Society Press, 1996.

[22] Edward Yourdon. *Death March*. Prentice-Hall, 1997.