Software Process Definition & Improvement: An Industry Report

Michael Jester, Herb Krasner, and Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
mgibjester@hotmail.com, {hkrasner, perry}@ece.utexas.edu

ABSTRACT
We present highlights of an process improvement project at a software center in preparation for CMM Level 3, with applicable examples given that relate to CMM level 2, level 3, and the effort needed in transitioning between level 2 and 3. For the efforts taken at this particular software center, the work documented is divided into two parts: the process definition and the process database. Finally, some qualitative and quantitative analysis, suggestions for future work, and conclusions regarding overall lessons learned from the experience are provided.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: D.2.9 Management

General Terms
Process improvement, CMM, Process Experience

Keywords
Software Engineering, Process Improvement

1. INTRODUCTION
For the sake of non-disclosure, the specific details about the company and the projects and technologies being developed in them are intentionally left vague and ambiguous. What can be said is this: the company is not involved in the business of selling mass-marketed software to the general public. Their primary customers are other corporations, and while software is not their end-product per se, the technologies developed by the company almost always require a strong software component. As a multinational corporation, this particular company has several "centers of excellence" in different parts of the world, each with individual focus. The focus for this particular center is the development of software that interacts with or supports the tools and technologies developed at other centers.

The center is a relatively new center (compared to the other centers within the company), having been established in Beijing, China in 1997 (starting with only a handful of engineers, but growing steadily in numbers). The typical project engineer is relatively young in their career (typical work experience of 2-3 years, with several fresh graduates), and many of the projects require a high level of interaction with other centers around the world. For these reasons (among others), having a viable software process is a center-wide objective, with formal CMM level assessment being used as the yardstick to measure progress toward that objective (in fact, every employee's year-end bonus in 2004 was contingent upon passing the CMM Level 3 assessment). Upon joining the software center in August of 2003, certification for CMM level 2 had already been obtained (in the previous calendar year) and efforts were already underway to receive CMM level 3 certification by the end of 2004.

Some of the challenges related to transitioning from CMM Level 2 to CMM Level 3 included:

- Defining new procedure documents and templates related to CMM level 3 KPAs
- Revising existing process documents and templates (related to CMM level 2) in order to comply with CMM level 3 requirements
- Establishing a process database (to use historical data for project planning, among other reasons)

These challenges will be explained in turn in more detail in the subsequent sections.

2. LEVEL 2 BASES
There are several important KPAs that are critical for moving to Level 3: software quality assurance, project planning and project tracking.

2.1 Software Quality Assurance
Software Quality Assurance is one of the KPAs for CMM level 2 and essentially is a mechanism that helps ensure each individual project is following the defined process the way they are supposed to [7,8]. It is a sort of "police system" for checking up on project managers and their teams. In getting prepared for the level 3 assessment, the process-related strengths and weaknesses as reflected in SQA reports served as an important indicator for how prepared the center was for the assessment (and what gaps had to be filled before the assessment).

Selection and General Duties of SQA Engineers

The software center had approximately 20 software projects under development organized under three different areas (where each area produced related software products). For each project, an engineer from another project would be assigned to act as the "SQA engineer". In general, the duties were not supposed to take more than two calendar days per month (since the SQA engineers still had their own projects that they were responsible for). General SQA duties included participating in peer review (including pre-reviewing documents before wide distribution and serving as a moderator during review meetings), moderating particular milestone meetings, performing periodic (i.e., monthly) checks to identify issues (and following up on the project team to ensure they were resolved properly), and (occasionally) helping the project team perform testing.

Playing the role of an SQA engineer brings to mind the phrase "it's a tough job, but somebody's got to do it". Much like software testers, SQA engineers typically only brings bad news to the project managers (in the form of a list of process non-

compliances). This was not particularly enjoyable for either the project manager or the SQA engineer.

SQA Reporting

At one point during the process improvement project, it had been decided to merge the previously mentioned tailoring form and the SQA report into one template (See Figure 3). More than just a move to reduce the number of documents by one, the idea was to minimize the confusion between SQA engineers and project managers as to what was considered "fair game" for the monthly SQA audit and what was not. For example, if the approved version of the tailoring form reflected that a particular artifact needed a formal review (rather than an informal review which was used for shorter and/or less "critical" documents), then the SQA engineer would know from the form that they should look not only for the document itself but also the review record associated with the document (as well as checking up to see that all issues uncovered during the review were properly resolved). Alternatively, if an artifact was "waived" by the manager (e.g., a design document was waived because it was combined with an architecture document), then the SQA engineer would not need to waste time looking for an artifact that did not exist and was not supposed to exist. In short, combining the tailoring form and SQA report helped project managers know what SQA engineers would be checking for and help SQA engineers ensure that the project was following the project-specific (tailored) process.

Phase	Artifact	Required/ Waived	Review Type	Reason/Comments	Properly Archived and available from Website?	Up-to-Date?	Properly Reviewed?	Review Issues addressed?	Other Comments
Inception	Vision Statement	Waived	n/a	continuation of existing product (no new vission statement needed)	n/a	n/a	n/a	n/a	
	Initial High-level Requirements	Required	Key Stakeholder approval	just need key domain expert to look at it and approve	yes	yes	yes	non-compliance	no record that issues raised by reviewer were addressed
	Prototype	Required	Walkthrough	have key stakeholder see a demo	yes	yes	yes	yes	
	Initial Project Plan	Required	Manager approval	no need for formal review of initial plan	yes	yes	non-compliance	non-compliance	no record of manager approval or review issues addressed

Figure 1. An example Tailoring form/SQA Report (only Inception phase shown). A complete report would list artifacts from all phases, although only artifacts from the current phase would be audited.

Role of the SQA Coordinator

At any given time one of the two full-time process engineers acted as an "SQA coordinator". The responsibilities basically included making sure that the SQA engineers were trained properly, collecting and reviewing monthly SQA reports, and helping follow-up on unresolved process non-compliances from the previous month (for example, if a reported process non-compliance was not resolved within the specified time frame agreed upon between the SQA engineer and the project manager, the SQA coordinator might escalate issues by going to the project manager and/or their supervisor).

The general goal was that the SQA coordinator would be able to have a compiled list of non-compliances from all projects in the center and try to prioritize them. Much like a software product manager tries to systematically organize and prioritize software defects so that the most important ones are fixed in a timely manner, the SQA coordinator needs to have the judgment to do the same with software process defects (again, Osterweil's claim that "Software Processes are Software Too" applies). In this regard, the SQA coordinator often acted as a resource to the SQA engineers whenever there was disagreement or misunderstanding as to how a process non-compliance should best be handled.

2.2 Project Planning and Tracking

Project planning and tracking spans two KPAs within CMM level 2. Besides being critical for CMM level 2, project planning and tracking is important for CMM level 3 in the sense that estimates of effort and schedule had to be made on historical data (there is where a project database comes in, to be discussed in more detail later). In order to have a consistency of data quality, having a standardized template for collecting project effort and calendar data was critical. A template using an excel spreadsheet (portion of it shown in Figure 1 and 2) had been used and was in circulation but had been deemed by most project engineers as being less than user-friendly. To help facilitate this, a new template was developed using Microsoft Project.

Phase	Task Number	Task	Category	Effort (Estimated)	Effort (Actual)	StartDate	EndDate	Responsibility	Dependency
Development	37	Write Design Document	DESIGN	16 hours	20 hours	2/1/2004	2/5/2004	Engineer1, Project Manager	
	38	Review Design Document	SQA	4 hours	4 hours	2/15/2004	2/15/2004	Engineer1, Engineer2, Engineer3, Project Manager	37
	39	Update Document	DOCUMENTATION	2 hours	4 hours	Required	Walkthrough	Project Manager	38
	40	Code Module 1	CODING	12 hours	10 hours	Required	Manager approval	Engineer1	39
	41	Code Module 1	CODING	12 hours	8 hours	Waived	n/a	Engineer2	39,40

Figure 2. Portion of an example list of tasks for a Project Plan Template (MS Excel shown above; both Excel and MS Project were implemented).

Several "pilot projects" starting using Microsoft Project before it was distributed center-wide (in order to minimize deployment issues in tailoring the template to be useful across the whole center). The template had all of the tasks dictated by the official process as well as important associated subtasks. If the project's tailored process had fewer required artifacts (and thus fewer required tasks) specified in the afore-mentioned tailoring form, the user would have to delete some tasks, but it was generally agreed upon that it was a better problem to have a project plan template with more tasks listed than necessary (and let users delete them as needed) than not enough (which would run the risk of having critical tasks overlooked). Note that the final version of the template was considered fairly complete, with over 100 tasks and sub-tasks listed in the template.

2.3 Other Key Process Areas

The above sections highlight some major efforts taken to prepare the center for the CMM Level 3 assessment. Note that not all of the Key Process areas from CMM level 2 and 3 are mentioned.

The omitted are not mentioned because they were either not applicable or needed very little adjustments. For example:

- *Peer Review* – mostly involved minor tweaking to the peer review templates and making sure people were following the peer review process properly
- *Requirements* Management—relatively few changes since CMM level 2 had been achieved.
- *Subcontract Management* – was not really applicable since there were not any outside subcontractors (and those working inside the center were treated as normal employees in terms of following the software process).
- *Software Configuration Management*—relatively few changes since CMM level 2 had been achieved.
- *Training Program*—was coordinated by the human resources group rather than the Software Process Group (with SQA training being a notable exception).
- *Intergroup Coordination*—had a procedure drafted to be reviewed by other centers but never reached formal agreement from the other centers that were involved in cross-center projects. Part of the problem was that we were the only center trying to achieve CMM. However, the fact that other centers were following the same high-level product development procedure did help with Intergroup coordination.

This concludes our discussion of the software process definition project, and we will now turn our attention to the process database.

3. PROCESS DEFINITION

The tasks for redefining the organizational process were organized as a project, with two full-time process engineers and some part-time help from some of the other project engineers. The major activities involved some definition of new processes and rework of existing processes. We will take a few of those CMM level 2 and 3 KPAs (Organization Focus, Software Quality Assurance, Project Planning and Tracking, and Peer Review) and discuss them in more detail [2,3].

3.1 Organizational Process Focus and Definition

Most of the documents revised or created during the life of the project were divided into three categories: high-level policies (broad in scope, giving the "big picture" view), procedure documents (providing a clear set of expected process inputs, outputs, steps, and measurements, if applicable), and templates (ready-to-go documents for project engineers to "fill in the blanks" with their project specific data). Of these documents, the most important were the templates in the sense that the templates were used the most by project engineers; the artifacts that were created during the life of the projects were based on the templates. The policies and procedures were mostly for reference. The templates were intended to be as consistent with the policy and procedure documents to the highest degree possible (the rationale being that taking this approach would lessen the learning curve required by project engineers to learn the new process; as long as they were filling in all the required fields in the templates, they would be in compliance with the official process).

Another purpose of the high-level policy documents was to serve as an interface between this software center and the rest of the company (spread worldwide). The company (which is a worldwide organization) had its own an official product development procedure that was imposed upon all centers; each center is required to have a process compliant with the company standard (note that not all of the other centers are software centers, or are trying to become CMM certified). Externally, the highest-level policies gave some evidence and assurance to higher-level organizations that the center was compliant with the company-wide standard. Internally, the policies served as a sort of "constitution" for all of the procedure and template documents to be developed; lower-level documents discovered to be in conflict with the high-level policies would have to be reworked (in practice this was rarely a problem, since the high-level policies allowed ample room for flexibility in process definition).

Although peer review is not a CMM requirement until level 3, a fairly mature and established review procedure was already in place (it had long been part of the company culture long before the center had any interest in CMM). This was a valuable asset when trying to revise the overall process, since it greatly facilitated valuable feedback from the "end-users" of the new process (i.e., project engineers, project managers, and resource managers).

3.2 Process Deployment

After process documents and templates were approved, they were made widely available via a website on the local intranet, where any engineer on the company network could view and download them. Ideally, a newly revised process document (whether a procedure or a template) would go through a "beta" phase before being made available for wide distribution. Specifically, a small number of projects would experimentally use a new template or procedure document, provide feedback, and necessary revisions to the document would be incorporated before widely publishing the new document.

Due to time constraints, this was not always possible (i.e., there had to be evidence of certain processes being in use over a certain period of time before the assessment). However, since many of the documents were simply newer versions of existing documents, the risk associated with not having a trial period was low. Also, the risk exposure caused by not having a "beta" phase for a given document was lessened by the fact that only a small percentage of projects would be in the same phase at any given time. For example, if major revisions were made to the project plan template, there might be only two or three projects in the planning phase at the time of deployment (who would often give feedback on the new processes). Furthermore, a website was provided where any engineer could log on and make a suggestion for process improvement (which was either an enhancement or a true defect). These suggestions would intermittently be examined, classified and prioritized by a process engineer or another AIT member (and if appropriate, incorporated into the process documents). The practice of reviewing the process defects and enhancement suggestions, very similar to the procedure followed for defects filed against an actual software product, brings to mind Lee Osterweil's contention that "software processes are software too" [4].

Besides posting new documents on the process webpage, weekly meetings served as a means to make center-wide, process-related announcements (e.g., if a new procedure was in place). While

these served a means for broadcasting information, often it was necessary to have follow-up sessions with a small group of engineers with more time for Q&A to effectively "train" engineers that were using a new process document (for example, making sure that a project team using a "beta" version of a document clearly understood the new procedure and the expectations associated with acting as a "pilot project").

In order to give the groups within the center more "ownership", two engineers from each discipline were assigned by their managers part-time (i.e. two days a month) to serve on the previously mentioned "Action Improvement Team". This essentially amounted to part-time supplemental workers for the two full-time process engineers and helped tremendously during process definition and process deployment. For example, several of the action improvement team members were project managers themselves, so having a deeper understanding of the process (by having helped contribute during the definition stage) greatly facilitated their ability to explain the process in turn to their team members.

3.3 Similarity to RUP

The process being followed was very similar to the Rational Unified Process (RUP—see figure 3). For example, there were very distinct phases throughout the life-cycle of each project (inception, elaboration, construction, and transition), each with an associated set of phase-specific activities, and demarcated with distinct milestones that marked the end of one particular phase and the beginning of the next. Within each phase, a certain set of artifacts had to be produced before moving on to the next phase; for example, having a formally reviewed and approved requirements document might be a requirement before moving from the elaboration phase to the construction phase.

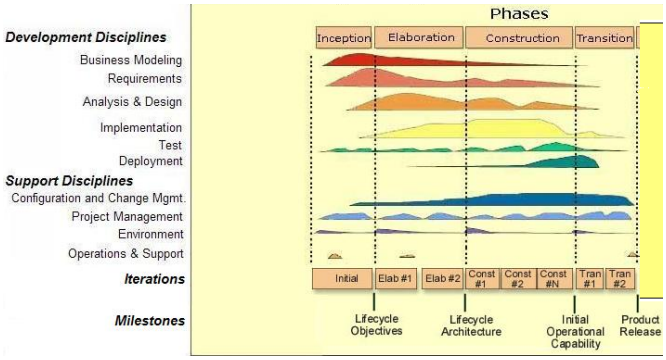

Figure 3. Model of the Rational Unified Process. Different disciplines have different levels of activity depending on the phase of the project.

Figure 4 shows a graph generated with historical data from an actual project using the process database. We discuss the process database in more detail in below; however, this graph is shown here to show the similarity to the RUP process, in that there are different disciplines (both development and support disciplines) spread across four different phases, with different levels of activity during each phase. For example, the R&S (Requirements and Specification) task type shows the highest amount of activity during the Inception phase, while COD (coding) shows the most amount of effort in the second and third phases (both of these examples are about what one would expect). Note that this type of view from the process database could give a higher-level manager an idea of how people were spending their time [5] (for example, if the inception phase showed little effort in Requirements and Specifications, and significant effort in coding, it could be an indicator that the project team was starting too early into development without a stable set of requirements).

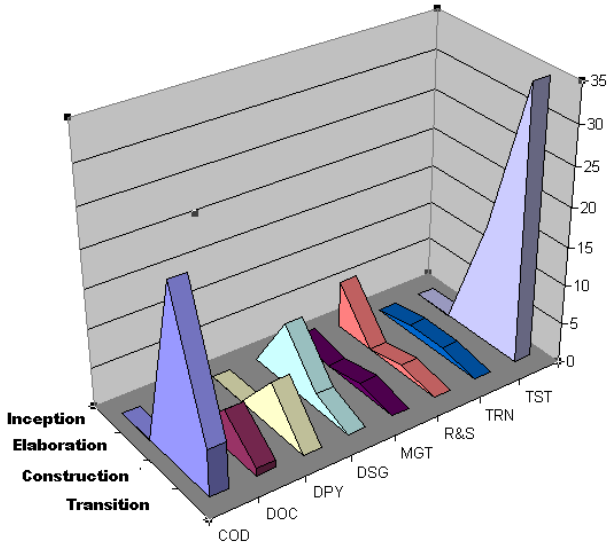

Figure 4. Task Effort (in person-weeks) for an actual project within the center. Phase (e.g. Inception, Elaboration) shown along the x-axis, effort (in person-weeks) shown along the y-axis, and activity type (e.g. coding, documentation, etc.) shown along the z-axis.

3.4 Tailoring Form

One of the high-level policy documents would specify which artifacts were absolute requirements for each phase and which were "optional". Typically, while a given project was still in inception or elaboration phase, a resource manager would mutually decide with the assigned project manager what artifacts would be necessary for the given project and document them via a "tailoring" form. For example, a manager might require a larger project to have more milestones and artifacts produced than a smaller project (i.e., the larger project required a "heavier" process than the smaller project). In this way the tailoring form served as a type of contract recording the agreement of what the project needed. In addition, it helped serve as a barometer for project tracking, since all items agreed to be produced within a given phase had to be produced before officially moving to the next phase. Furthermore, the tailoring form served as a means to provide some degree of flexibility between different projects. There are some tradeoffs to be managed between creating a process that satisfied the "organizational focus" requirement of CMM level 3 and allowing some flexibility to allow project managers to tailor the process to the specific needs of a project. A single process to be applied for all projects might end up being so over-specified that it would be unwieldy for a high percentage of projects. On the other hand, making the process too loosely defined would make it difficult to have an easily understandable and repeatable process, and also might cause a low score on the organizational focus requirement of the formal CMM assessment.

Providing a tailoring form with some required and some optional items helped manage these tradeoffs, giving a sufficient amount of structure to every project, while at the same time having some flexibility to modify to process to the specific needs of the project.

4. PROCESS DATABASE

The purpose of creating a process database was to create a central repository of historical project data off which to base estimates for future projects. Pankaj Jalote in "CMM in Practice" [3] often gives examples of how the existence of a database proved helpful in project estimation. Besides project estimation, project engineers within the center hoped to used the process database for tracking (during the lifetime of the project) and post-mortem analysis (after project completion).

The process database had three main types of data which will be discussed in turn: task effort data, defect data, and code size data.

4.1 Task Effort Data

A consultant who helped part-time during the assessment preparation had stressed that the task effort data was the most important in terms of being prepared for the assessment; one could not claim that project planning estimates were based on historical data without task effort data (even if defect and product size data were ready available). It was also observed that the historical data collected on task effort was the least reliable of the three types mentioned. This is presumably because the collection process was the most "human intensive" of the three; code size data was calculated automatically as developers checked in new revisions of code, and defect data, while requiring human input into a database once found, was more or less limited to particular phases of the project (mostly coming during alpha and beta testing), whereas task effort data required much more day-in, day-out attention to detail on the part of the project manager.

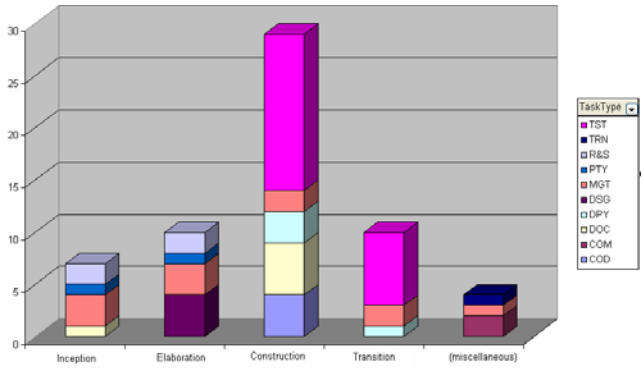

Figure 5. An example task effort breakdown by phase and by task type (phase on x-axis, effort in person-weeks on y-axis)

Besides the extra effort associated with data collection, standardization appeared as an issue. For example, what level of granularity was most appropriate for reporting data? Some project managers were reporting time in terms of hours (e.g. 3 hours coding, 2 hours documentation, 3 hours testing, etc.) while others were in the habit of reporting in days (e.g. 0.5 days coding, 0.25 days documentation, etc.) Since different reports used different granularities, all effort data was "normalized" to person-weeks when archived in the process database, where a 40-hour work-week was assumed. This method still leaves room for discrepancies; namely, a team member who worked 60 hours in a given week, whose time was recorded at the hour level of granularity, would have the effort for that week calculated as 1.5 person-weeks of effort before being archived in the database. Meanwhile, a team member on another project, who also put in 60 hours but whose time was recorded at the "days" level of granularity, would only have 1 week of effort recorded, unless the project manager had the habit of taking overtime into account and recording the effort value for that week as 7.5 days (or 1.5 weeks, equivalent to 60 hours) of effort (an unlikely scenario, since discussions with project managers led to the belief that those interested in taking note of how much overtime was worked were more likely to record time in hours rather than days).

This situation seemed unavoidable for data that was already collected. Since this type of standardization had not been important previously (not a requirement for CMM level 2), the best method of dealing with it was to make people aware of possible discrepancies of older data, make them aware of the standards that would be reflected in the process database and encourage everyone to move in the same direction (for example, upon mutual agreement of project engineers, the newer versions of the Microsoft project templates had "days" as the default unit of time).

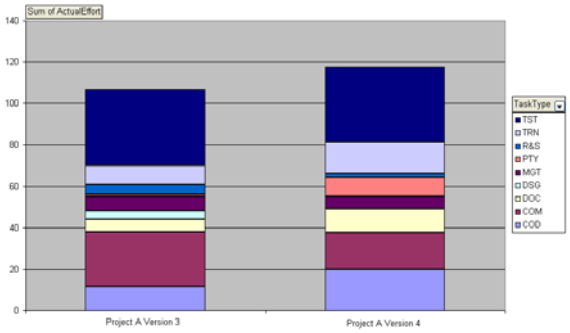

Figure 6. Comparing task type breakdown for two versions of "Project A". Note the relative similarity in percentage allocations for each task type.

Another problem with standardization centered around task type. For example, if two hours were spent reviewing a requirements document, a team member in one project might classify the effort as being a "requirements and specification"; in another project, a team member might reference it as a "documentation" activity; yet another engineer might reference it as "software quality assurance" since it was review. All this would lead one to expect more commonality of task type breakdowns for projects that were different versions of the same product (and often had the same project manager), and less commonality for projects working on different products (see figure 6 for an illustration). Similar to the issue of granularity previously mentioned, these sorts of discrepancies were not as much an issue when still at CMM level 2. Likewise, the goal was not so much to sort out discrepancies for data that had long ago been collected but to help ensure more standardization looking forward. This was handled by giving each default task in the Microsoft Project template a default value for the task type category, as well as having more precise

definitions included in the template that would help sort out ambiguous situations like the ones mentioned above.

While the task effort data was primarily designed for estimation (which obviously came in the early stages in the project), engineers were encouraged to use the available data in other ways if deemed helpful to their work. For example, at least one project engineer used the data for post-mortem analysis in comparing the task type breakdown in a newly released product with the task type breakdown for the previous released version of the same product (after looking for trends, he noticed a relatively low percentage allocated to design, with a high percentage spent on testing and debugging, and based on his experience with the project, felt that the former was probably a root cause to the latter).

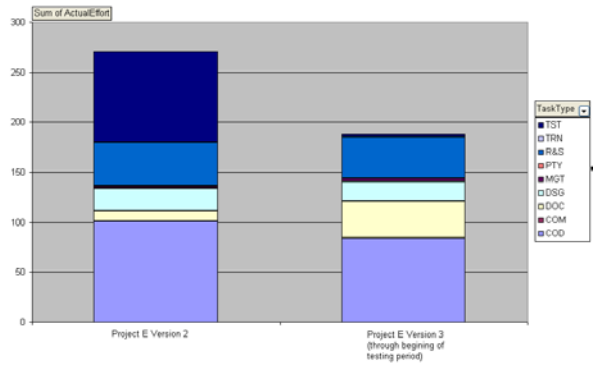

Figure 7. Comparing task type breakdown for two versions of "Project E" (bar on right-hand side only includes data through the beginning of the testing period, which is why the effort for the "TST" task type is low). Note the relative similarity in percentage allocations for each task type, but the difference with those shown in figure 6. This particular project manager only used six of the available task type classifications.

4.2 Defect Data

Like task effort data, defect data had to be manually collected; unlike task effort data, the tool used for data collection had been standardized globally (not just center-wide) for about a decade. Since use of this particular tool had been widely adopted for a relatively long period of time, the data quality was a bit more trustworthy than that of the task effort data.

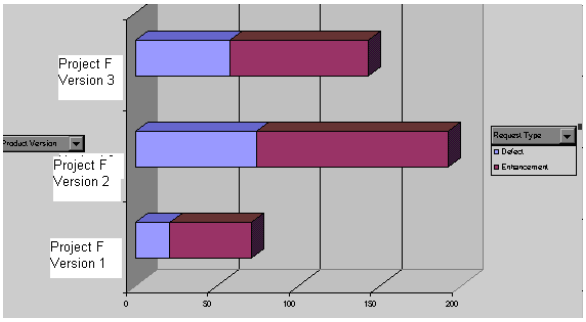

Figure 8. Number of defects (and enhancements) for successive versions of the same product.

Nothing changed regarding the data collection method a result of this project. The one thing that did change was the data presentation. Specifically, data was ported (once every evening) from a server located at a different center (that stored defect data for multiple centers worldwide) to a local server, thus allowing creation and manipulation of graphs similar to the ones shown for task effort data. While the other tool also permitted graphical report ability, the advantage of having a single tool for multiple data types was that it provided a "one-stop shop" for multiple types of project-related data, and also offered a lower learning curve (since the graphical query abilities of the local tool had the same look-and-feel, regardless of the type of data being viewed). Perhaps more importantly, having the data stored on the local intranet enhanced performance; even if the same type of data and graphs were already available on the previous tool, an engineer might use the tool accessing the local data simply because the web page could retrieve the data much more quickly.

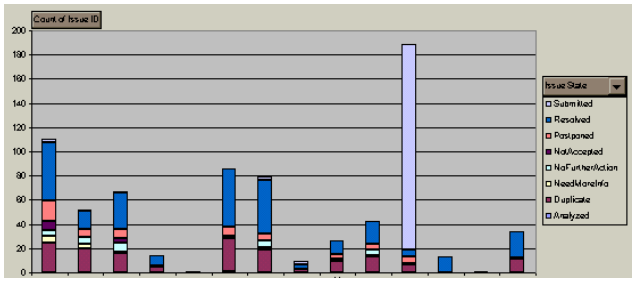

Figure 9. Defect Status versus Component (components on x-axis, with component names not shown intentionally)

During beta testing of the tool, defect data was used by one project team for post-mortem analysis and by another for project tracking (during alpha testing). The former involved using the tool to compare defectivity between different versions of an ongoing product release (see figure 15). The latter involved tracking defect status (e.g. submitted, resolved, postponed, etc.) for various components as the project team tried to resolve several hundred defects uncovered during alpha testing in a timely manner (figure 16). Due to the large number of defects to be resolved by a relatively small team (four to five project engineers), with status updates required more than once a week, the project manager found the process database to be an extremely useful tool to help analyze which components needed the most attention during bug-fixing.

4.3 Code Size Data

During requirements gathering for the process database, there was some debate over whether LOC was a viable metric for measuring growth of product over time. (Some engineers argued that "function points" would be a better metric for measuring product size). Nevertheless, most agreed that using lines of code, while perhaps crude, would be the easiest metric to collect; calculating these metrics involved writing a simple script that manipulated existing, readily-available metrics from the source code control system (the system calculated lines of code added, deleted, or changed for every revision; making these statistics available at a macro scale was achieved by calculating a sum of the individual values contained for all code revisions associated with a given product.

At least one project manager showed interest in looking at code quality (defects generated per LOC) and comparing it with figures published in industry (used for post-mortem project analysis). While the database did not automatically generate this type of information, it could be computed by collecting defect data and code size data individually, and then performing a manual computation. In preparing for beta testing of the product, product size tracking was the most anticipated use. For example, a given project might base a project schedule on the assumption that the product to be constructed was relatively similar in size and scope as a previously completed project of a known size; during the life of the project, the code size would be monitored month-to-month and reported in a monthly report sheet. If the code size exceeded a pre-defined threshold (agreed upon during early project planning), thus signaling that the project was perhaps larger than anticipated, then the project manager would need to take corrective action (hopefully getting a deadline extension for the project). A prototype for an excel spreadsheet with some background VBA code that could download this information automatically from the database to the spreadsheet was used for this type of project tracking. (The prototype was not made widely available due to some bugs found in beta testing, but the information could be gathered by looking at the code size graphs in the database webpage).

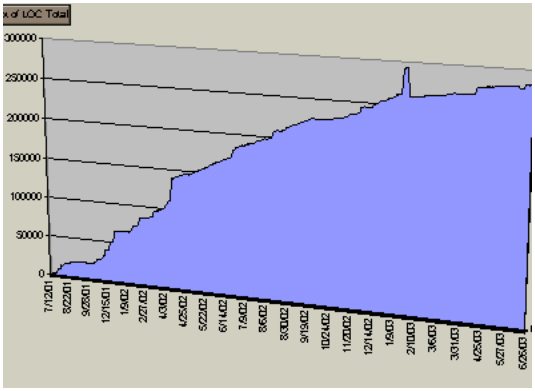

Figure 10. Code Growth over time for a project during entire product life-cycle (note the "leveling-off" effect that occurs over time).

Another way to get a feel for the project progress (based on code size data) was to look at the shape of the "net lines of code added" curve. In general, a graph that showed a relatively linear climb upward was indicative of a project that was not yet "code complete" (the pace at which features were being added was not showing signs of slowing down), while a more "mature" product had a graph that indicated a "leveling-off" effect (figure 10) over time in terms of the lines of code being added (most changes being made were presumably defect-fixing, refining and refactoring rather than adding major pieces of new functionality).

5. BEYOND LEVEL 3

While not required for CMM level 3, it is never too early to provide a quantitative bases for process management and continuous improvement.

5.1 Quantitative Analysis (See Figure 12)

The following section provides some quantitative analysis regarding data collected into the process database. Note that there are several project's worth of data that are not reflected in these tables and figures. In some cases, it was thrown out due to dubious quality. In others cases, the project data was simply not available (for example, projects that were completed more than two years previously might not have had good collection mechanisms at the time to help with data collection; similarly, project starting more recently were still ongoing by the time the process project was finished; thus any data generated would not reflect a "complete" project). However, there is still enough data available of reasonable quality to play around with and explore interesting trends. Some of these are discussed below.

Code Quality

The first analysis involved defectivity. Here we looked at the net LOC added during the life of the project compared with the number of defects found (similar to the common metric *Defects/Kloc*, except inverted). Eight projects (with between one and four version releases each) were examined, for a total of eighteen projects' worth of data.

The first thing to be noted is the variability in the rates of defectivity. A low value of 35.11 and a high value of 674.16 (more than a order of magnitude difference) is shown in the table, with an average value of 252.71. However, when looking at the project summaries, we notice that the variability is dampened substantially (a low value of 126.90, and a high value of 405.79). Note also that the standard deviation for the projects mentioned is also much lower than the standard deviation for the complete data set (106.52 versus 207.80).

Another interesting note came when looking at multiple releases. For example, ten product release transitions (e.g., transitioning from Project A version 1 to version 2, Project A version 2 to version 3, Project B version 1 to version 2, etc.). Typically, if the defectivity value of a given release was above the average value of 252.71, then the subsequent release had a value that was below the average value. The inverse also typically held true (i.e., if the given release had a below average value, the subsequent release had an above average value). Eight out of the ten transitions held to this pattern of "centering around the mean".

This trend seems akin to the studies of Lehman and Belady [6], who studied the dynamics of successive releases, finding (among other things) that one could predict by looking at the expansion of a system from release to release when a "clean-up" release would be necessary (in order to provide a more stable platform for continued evolution). Another interesting thing is that in four of the six projects, the defectivity increased (i.e. the net LOC added/defect decreased) from the first release to the second release. This could possibly explained by two phenomena explained by Brooks *Mythical Man-Month* [1]: a) the "second system effect", in which a designer's second system almost always has more bugs than the first, and b) the increased maturity of users of the system (attained by time spent using the product) and thus increased ability over time to find bugs (some of which may have existed in the first release but remained undiscovered until the second release).

Whether or not the two factors mentioned are the primary issue, or whether it is due to other causes, the applicability is the same: the defectivity of the initial release of a product can be misleading, and thus should be taken into account when allocating schedule time for testing and bug-fixing during the second release. One example of "bucking the trend": project C version 1 had a relatively high defectivity (low value of Net LOC added/Defect). From discussions with the project manager and the second-line manager, it was known that more serious "early-testing" efforts were put into that particular project, which led to higher end-user satisfaction. Thus, the low value of 142.39 is probably less indicative of low code quality and more indicative of a better job of finding all of the bugs in the early stages.

Productivity

The second metric we look at is productivity. The number we are most interested in is Net LOC Added per day per team member. Note that days is simply the number of days from beginning the beginning of the project till the end of the project (i.e., it is "calendar days" rather than "working days"). Note also that Net LOC Added is simply LOC Added minus LOC Deleted (including commented LOC).

Similar to what we discussed in the previous section, we notice that variability is much lower when comparing the "project subtotal" values—eight data points (shown in the green rows in Figure 20), each of which is a summary value of the given project—than when comparing all of the data points individually (again reflected in the a lower standard deviation value—35.74 versus 57.75). Also similar to the previous section is the fact that four of the six multiple-release products had a drop-off in productivity between the first and second release (presumably because they had their hands full doing bug-fixing in the second version, a trend discussed in the above "defectivity" section); of the two projects that did not hold to this trend, one of them (Project A) did show a significant drop-off by the third release (from 115.28 LOC/day/team member to 39.22). The other (Project E) had a productivity rating that was so low that one suspects that a significant percentage of the calendar time was taken up by requirements-gathering activities rather than development activities. Perhaps to avoid "comparing apples to oranges", it would be better to look only at the productivity for a given project during the development phase, or only during the development and transition phases; this would dampen the effect of larger projects (with longer schedules) needing more time to perform proper requirements gathering.

The applicability to these statistics are also similar to what we mentioned above in the sense that it is easier to base estimates off of previous releases from the same project rather than completely different projects. This is common sense, to some extent; having the same project team, the same manager, the same domain-specific and project-specific issues, as well as the same user base should lead to higher inter-correlation between data sets with the same project than between data sets of different projects. And it offers little help to project managers taking on the first release of a new product. However, perhaps the variability could serve as a reminder to both project team members and managers to be wary of making optimistic schedule projections for the first release (since every product is different) and for the second release (since there tends to be a drop-off in productivity, when one might expect to see the opposite).

5.2 Next Steps: Focusing on Measurable Benefits vs. CMM Certification

The fact that CMM Level 3 certification was achieved can be a double-edged sword: on the one hand, the project could be deemed successful; on the other, there is a viable danger that after achieving certification, complacency would set in and undermine the work done on process improvement (For example, [8] reports that two-thirds of software process improvement programs started in the late 1980's died sometime after a formal assessment was performed). If the entire motivation for software process improvement activities is simply a nice certificate to hang on the wall, then one could almost expect some sort of backsliding after the assessment. If, however, concrete numbers can be shown measuring the payoff that comes with the investment of a software process improvement program, a center has a much better chance of overcoming post-assessment complacency. Indeed, if numbers can be shown that convinces upper management of the added bottom-line value of a software process improvement program, certification almost becomes secondary. While measuring the cost of process is not a requirement for the CMM model until level 4, one could argue that it is in the best interest of organizations to start making these measurements much earlier.

For the center described in this paper, time effort data (like that shown in figures 11) was collected for process-related projects as well as normal software projects. This data should provide a good baseline for the cost of software improvement activities in terms of staff-hours. The next question is, how can one measure the savings derived from good process? [8] suggests at least one major (and measurable) benefit is the reduced cost of rework. In the previous section, we noted that time recorded in the TST (testing) category including not only testing, but also bug-fixing. If the results found here are a reliable indicator, then the center should probably see an overall decrease in the percentage of project time spent in the TST category over time; if appropriate, additional categories could be added to separate testing time and debugging time (or possibly other types of rework), if it helped to provide more precise measurements.

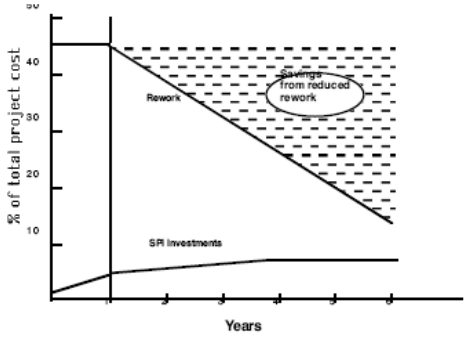

Figure 11. Cost of rework and Software Process Improvement (SPI) investments over time. Shaded area represents savings from reduced rework.

Another way to measure added business value can come from utilizing extra field entries in the defect database. The database entry form has fields for items such as "time to analyze", "time to fix" and "time to validate", along with the risk associated with implementing the fix for a given defect. Unfortunately, many of

these fields are not required entries (i.e., the user can leave these fields blank and still submit a defect report). Other fields exist (which are mandatory) that assign business value to a given defect or feature. (It can also be thought of as a "end-user pain index"; in other words, how unhappy will the end user be if a particular defect is unresolved or if an enhancement suggestion is ignored?) One software process consultant encouraged the use of these two sets of fields (possibly making the first set mentioned "required" fields) in conjunction in order to make more intelligent, business-based decisions for which defects got fixed for a given product and which did not (e.g., defects with low cost to fix and high business value were always fixed, those with high cost to fix and low business value were ignored, and the in-between cases would be judgment calls). If every project in the center recorded this type of data and made use of it for decision-making, one would expect an overall drop in time measured on bug-fixing and rework, while retaining (or perhaps even improving) the overall business value of the delivered product.

Yet another measurement that could be added would be defects found before coding or testing even began—for example, defects uncovered during the requirements or design phase. A simple way to measure this would be to take the defects recorded during peer review (as shown in figure 4) and include the data in the process database. Since peer review (including use of templates such as the one shown in figure 4) is already well incorporated in the company culture, one would expect the data to be of reasonable quality. Furthermore, by organizing this type of data centrally in the process database, one could compare projects and see if those projects with "good review" (i.e. high number of defects found in the early phases) had a lower percentage of project time spent on rework—a trend which, if backed up with numbers, would provide ample justification for the business value of process-related activities such as peer review, training (e.g., for requirements gathering and design), and software quality assurance audits, among others.

6. CONCLUSIONS

The project overall was deemed successful in the sense that CMM Level 3 certification was in fact achieved (and as a result, all the engineers were happy to receive their year-end bonus). At the same time, while much was accomplished, a lot of work remains in terms of "maintaining" the new process (to avoid the pitfall of achieving certification, only to regress in process maturity once the incentive to actively improve the process is removed). Making more measurements for the return on investment associated with process-related activities might be the best solution to this problem.

A number of important lessons emerged or were reinforced during the process project execution:

- Lee Osterweil's tenet that "software processes are software too" was reinforced throughout the project: defining, modifying, deploying, improving, and maintaining processes have many similarities to the process that one goes through for creating software products.
- Changing the culture takes significant time. Several process books indicate there is a long lead time associated with getting a return on investment from process-directed initiatives, and the experience of this particular project was no exception (take for example, the time it takes to generate each one of the data points shown in figure 11; even a product that has a relatively short cycle time of one release every six months will only generate 6 new sets of data points in three years, making some significant statistical analyses problematic).
- An established peer review process can be a valuable asset in the overall process of achieving CMM level 3.
- Risk exposure can reduced by introducing improvements incrementally to a few projects before full release
- Ownership of processes provides significant motivation for their definition and deployment success. Involving project management facilitates that success significantly
- Tailoring Forms were critical to avoid the one size fits all problem where processes are either so over-constrained as to be unwieldy or so under-constrained as to be unrepeatable.
- It is better to have a full project plan where items can be deleted in tailoring the plan to the needs of the project. Critical tasks are less likely to be overlooked this way.
- Standardization of project data is critical in achieving comparable results. Hence, use default measurements in the project template.
- The use of a single data presentation tool for "one stop shopping" provides a lower learning curve, uniform presentation and a higher likelihood of being used extensively.
- When frequently updated, the project database was found to be an extremely useful tool to help analyze which components need the most attention during the development process.
- Brook's prediction of a drop-off in productivity for the second release was reinforced throughout the project.
- Continuing process improvement is primary, certification is secondary.
- Finally, while measuring process costs is not required until CMM level 4, it is in the best interests of organizations to begin measurements as early as possible.

In summary, despite of the expense associated with investing in a good software process, it is still an investment worth making, knowing that the expenses resulting from not having a process are even higher.

7. REFERENCES

[1] Fred Brooks, The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition, Addison-Wesley, 1995

[2] Kim Caputo, CMM Implementation Guide: Choreographing Software Process Improvement, Addison-Wesley, 2003

[3] Pankaj Jalote, CMM in Practice: Processes for Executing Software Projects at Infosys, Addison-Wesley, 2000

[4] Leon J. Osterweil, "Software Processes are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9", ICSE 1997

[5] Dewayne E. Perry, Nancy A. Staudenmayer, Lawrence G. Votta, "*People, Organizations, and Process Improvement*", IEEE 1994

[6] M.M. Lehman and L.A. Belady, *Program Evolution: Processes of Software Change,* Academic Press, New York, 1985

[7] Herb Krasner, *"Using the Cost of Quality Approach for Software"*, Crosstalk: The Journal of Defense Software Engineering, November 1998

[8] Herb Krasner, "Accumulating the Body of Evidence for the Payoff of Software Process Improvement", Software Process Improvement, IEEE Computer Society Press, 2001

Project	Net LOC Added	Defect Count	Net LOC added/Defect	Project Length (days)	Net LOC added /day	# of team members	Net LOC Added/day/ team member
Project A.1	70570	141	500.5	168	420	4	105
Project A.2	83005	460	180.45	144	576	5	115
Project A .3	37806	211	179.18	241	157	4	39
Project A.4	146189	389	375.81	436	335	4	84
A subtotal	**337570**	**1201**	**281.07**	**989**	**341**		**86**
Project B.1	51393	90	571.03	102	504	2	252
Project B.2	3125	89	35.11	184	17	1.5	11
B subtotal	**54518**	**179**	**304.57**	**286**	**191**		**132**
Project C.1	102946	723	142.39	203	507	4.5	113
C subtotal	**102946**	**723**	**142.39**	**203**	**507**		**113**
Project D.1	60674	90	674.16	112	542	4	135
Project D.2	33488	652	51.36	199	168	4	42
D subtotal	**94162**	**742**	**126.9**	**311**	**303**		**89**
Project E.1	10117	*200***	50.59	325	31	2	16
Project E.2	283820	*649***	436.65	714	398	7	57
Project E.3*	86800	*400***	217	492	176	4	44
E subtotal	**380737**	**1249**	**304.83**	**###**	**249**		**39**
Project F.1	41339	71	582.24	115	359	4	90
Project F.2	45848	192	238.79	218	210	4	53
Project F.3	77562	143	542.39	327	237	4	59
F subtotal	**164749**	**406**	**405.79**	**660**	**250**		**67**
Project G.1	52968	361	146.73	459	115	4	29
Project G.2	14849	83	178.9	437	34	2	17
G subtotal	**67817**	**444**	**152.74**	**896**	**76**		**23**
Project H.1	154059	424	363.35	392	393	5	79
H subtotal	**154059**	**424**	**363.35**	**392**	**393**		**79**
Total	**1356558**	**5368**					
Mean			**252.71**		**294**		**78**
Std Dev			**207.8/106.52**		**131**		**58/36**

Figure 12. Net LOC added/Defect for various projects (*Project E Version 3 shows data through beginning of testing period only). Net LOC added/Defect for various projects (*Project E Version 3 shows data through beginning of testing period only; **estimated values for these individual product releases; the total number of defects for all releases in the project is accurate).