Managing Process Diversity While Improving Your Practices

Michael Deck, Cleanroom Software Engineering, Inc.

In recent years, the industry has gradually moved toward implementing standard processes across large organizations. The benefits of doing so include simplified accounting, measuring, and managing. Implementing organization-wide processes also makes it easier to judge the capabilities of an organization as a whole, which is important in large-scale software development where organizational structures can have a profound impact on success.

But there are also drawbacks to process uniformity, including cases where clean process documentation hides chaos in the trenches. Another drawback is the difficulty to modify an organization-wide process that has been carefully vetted by committee after committee. An organizationally uniform process can prevent smaller software teams working within large organizations from adopting a process tailored to fit their needs.

I define software practice improvement as a set of goals that sits between individual programming practices and organization-level software process improvement. By focusing on improving software practices, we stand a better chance of achieving success using well-understood techniques such as risk analysis, incremental development, and team ownership.

I define process diversity as using several different process variants simultaneously. This article describes a project in which localized software practice improvements led to process diversity. It describes certain problems that we encountered and resolved during the effort. It also presents some simple techniques that any project—small or large—can use for managing diversity. The techniques I discuss in this article evolved in the course of one multiyear project.

If you do not carefully plan and manage process diversity, it quickly becomes unmanageable. I used to advocate process uniformity, warning against the management problems caused by diversity, but recent experiences have suggested more effective ways to address these problems. This article explores these alternative strategies, which include adopting a basic set of practices that are uniformly enforced across all development steps, using risk analysis as a technique for selecting targets for software
process improvement, and planning for process diversity.

**A software practice improvement story**

The project I explore here started with an enthusiastic team of six relatively inexperienced developers, plus a technical lead who had significant domain expertise from previous similar efforts. We had to develop a platform for real-time control of several optomechanical instruments. Several different missions were the customers for this platform, including two space missions and one ground-based observatory. We sized the project initially at approximately 70,000 lines of C and C++ code, most of which would run in the VxWorks operating environment.1

The project proved to be a good environment for trying out advanced software engineering techniques. With a small team, we didn’t expect management issues to swamp the technical issues. And, because the developers hadn’t learned to succeed by working within an existing flawed process, they were willing to try new ideas. The project’s developers had almost no software engineering training or experience, although they had been programming for some time and were domain experts in real-time controls, optics, and electronics. The distinction between programming and software engineering is intentional because the latter also includes elements such as requirements analysis, metrics, team organization, project estimating, planning, conducting reviews, and developing design documentation.

We were determined to try new software processes and practices to avoid mistakes made on previous projects. However, we also planned to experiment with certain practices that the organization as a whole (a large government lab) had not yet approved. Risk analysis played a central role in defining our specific software practice improvement activities.

**Initial risk analysis**

We began our software practice improvement effort by analyzing risk. We used several sources to devise a set of typical project risks, but our main source was the Software Engineering Institute’s Taxonomy of Risks.2 We supplemented the SEI source with a risk-guided analysis of several current quality programs, including the SEI’s Software Capability Maturity Model (CMM) and the Malcolm Baldrige award criteria. The assessment process included a questionnaire followed by meetings with the team.

We determined that approximately 12 areas could be classified as high risk, nine as medium, and 11 as low. These numbers reflect the combination of over 100 individual risk criteria. Among the high-risk areas were several existing issues, including having

- no established process for requirements tracking and
- insufficient confidence in the requirements process.

There were also several additional potential risks, including

- inconsistent planning for and documenting of project activities and commitments;
- uncontrolled change in processes; and
- instability of requirements, specifications, and designs.

We explored in detail some of the potential effects of these risks. For example, the lack of an established process for requirements tracking led previous projects to discover missed requirements late in the process. We were also concerned about requirements stability, because the project had many sponsors with very different needs.

**A single improvement path**

We developed an improvement plan to treat these risks in approximate descending order of importance and integrated the practice improvements into an incremental development model. Our idea was to deliver increments of product code that could be integrated piecemeal with the target hardware (also under development at the time). In each development increment, we would further improve and formalize the practices.

Our increment model is similar to that used in the evolutionary (EVO) model,3 except that our increments were somewhat longer than EVO might recommend; they are on the order of three to six months each. The incremental-evolutionary model is key to our topic of diversity because such a model (as opposed to a waterfall approach)
permits controlled evolution of processes as well as products.

The improvement plan set a high initial standard for software engineering practices. These practices included several techniques drawn from the Cleanroom software engineering approach:\textsuperscript{4,5}

- formal behavior specifications using a notation based on the denotational semantics of Harlan Mills and his colleagues;\textsuperscript{6}
- reliance on team verification review rather than on testing for most requirements-coverage analysis; and
- numerous documents that were new to the project, including a detailed incremental project plan, user-view requirements, and formal specifications at the system, component, class, and method levels.

This plan was intended to apply uniformly to all project code, an aspect that did not seem significant at the time. We embarked on this learning process enthusiastically.

Initial success

The first increment’s goals were reasonably ambitious. The technical goals included proving the feasibility of numerous real-time control and scheduling algorithms, installing and learning a new compiler and runtime environment, and debugging a new optomechanical apparatus. Management goals included filling gaps in the domain knowledge of some team members, identifying significant gaps and inconsistencies in requirements, and identifying design trade-offs and alternatives.

The first increment achieved all of these goals. The mandatory documentation and code reviews contributed greatly to cross-training team members in domain knowledge. The formal specification with Box Structures uncovered numerous significant architectural and requirements problems that would likely have arisen much later in a traditional process. We found the documentation useful in focusing discussion on design trade-offs and alternatives.

That the project also delivered significant functionality in the first increment demonstrated the capability of both the process and the team to meet critical user requirements. The first increment created an unexpectedly robust and maintainable architecture. Although little of the first increment remains in today’s product (due to a switch from C to C++), most of the architectural elements that were present in increment two remain in increment seven after almost two years of additional work.

The steep learning curve did delay delivery of the first increment, though there were a number of unplanned outside events that also delayed the schedule. Perhaps most importantly, the team that developed the first increment was extremely enthusiastic about the Cleanroom process they had used and were committed to continuing its use.\textsuperscript{1}

Other risks emerge

Unfortunately, the initial risk assessment did not account for all the actual risks that were to appear. Much of our inability to account for all the actual risks can be explained by pointing to changes in the project’s environment, but the fact remains that by increment two there were significant new risks. These risks did not fully manifest themselves until increments three and four, which is another reason why we did not address them quickly enough.

The most significant change in the project’s environment was in the area of staffing. As the project grew, additional personnel joined the team. Most came from other projects that had used the organization’s traditional process. Many were more experienced, and more senior, than the original team.

The arrival of the newcomers led to two kinds of problems. First, the original project team members had to train the newcomers in their practices before the newcomers could be productive. This lowered overall productivity at a time when schedule pressures were already high. Second, the senior transfers were comfortable with their traditional processes and were highly resistant to trying new techniques. Endless debates further reduced productivity.

We also underestimated the impact of not having the team trained in basic software engineering principles. A team that had some training in the basics of inspections, requirements, and formal testing would have been more comfortable with verification-based inspections, formal requirements specifications, and stochastic testing.
We also had to spend more and more energy explaining and defending our process to senior staff and upper management. We waged battles in email and hallway discussions. In the end, we could not enforce process compliance and still meet our aggressive delivery schedule.

Resolution

After an increment of partial anarchy, a consensus began to emerge around the possibility that a single software process and a single improvement path might be unrealistic for the project. In particular, we observed that there were really three kinds of activities going on at any one time.

Some of the code was being built to support critical elements of space-borne scientific instruments. This code had to meet the highest level of quality and reliability but could be commensurately more expensive to build and document. Some of the code was being built to support noncritical components and ground-based instruments. This code did not have to meet the same reliability requirements as the critical software.

Finally, there were a lot of necessary experiments to run. For example, would a particular memory-management strategy meet performance objectives? The simplest way to find out was to write a little code and run an experiment, but nobody wanted to subject the code to a formal process of specification, design, review, testing, and documentation. The result was a diverse set of processes in simultaneous use, each of which was tailored to meeting key needs of a software subset.

Adopt basic practices

Our experiences with this project led us to believe that no process, no matter how well-intended, can satisfy the needs of every project. Process diversity is, therefore, a fact of life. What we must do is manage that diversity. First and most importantly, we learned to define a set of foundation practices upon which there is near-universal agreement. Although we did do this in our project, we didn’t consider the pool of individuals who were likely to transfer into the project in the near term.

Before embarking on a software practice improvement effort, the team must agree on—and ideally document—an initial set of universally accepted practices. If the initial set of practices isn’t broadly accepted, adding advanced practices will be more likely to fail.

Consider the analogy of a house built in an earthquake zone. If the foundation is strong, the earthquake will just jar loose some trim. But if the foundation is weak, the entire structure will crumble. In the absence of broad, grassroots acceptance of basic processes, proposed improvements will not survive the inevitable project stress events such as turnover, reorganizations, requirements changes, and “crunch” mode (that painful time of extreme schedule pressure).

The foundation practices must be part of the organization’s fabric so that people no longer question the practices’ worth or try to eliminate them in rounds of cost cutting. We recommend defining foundation practices in the following areas:

- requirements management,
- software project planning,
- peer review,
- quality assurance,
- configuration management,
- metrics, and
- defect prevention.

Risk analysis will also guide the definition of further foundation practices, as will analysis of prior problems and crises. It can be challenging to define a set of consensus practices in some areas. It might require baby steps initially. For example, although defect prevention is an advanced practice of SEI’s Software CMM, most projects can find consensus to hold a regular informal discussion of significant software bugs, perhaps at the regular staff meeting. These sessions can start with an analysis of every bug that took more than two hours to fix. The discussion will naturally flow to how that bug or similar ones might have been prevented. After the meeting, it is likely that programmers will return to their desks and look for similar problems in their code.

One of the questions not fully answered by the projects on which this experience report is based is who must participate in consensus-building. In these projects, we sought universal consensus only within our project, but that turned out to be insufficient because of personnel flow between the project and the rest of the organization.
On the other hand, seeking consensus and defining processes beyond a single project was perceived as being outside the project’s scope of responsibility. Indeed, the teams sought outside consulting advice because they were unsatisfied with weak organizational processes and sought to institute stronger local practices. Although educating the organization as a whole would have been useful, there were significant benefits to stealth, lest local decisions be overruled by the organization’s process engineers.

Use risk analysis to select practices

Once the foundation practices have become a comfortable part of everyday project operations, you can begin to identify targets for improvement. If you select the wrong targets for software process improvement, you will waste valuable resources fixing the wrong problems. For example, one project in a different organization was canceled because the team did a poor job prioritizing risk. The team spent months learning object-orientation so that their code would be extensible and maintainable, but their greater risk lay in not being the first product on the market. Understanding the threats makes it possible to avoid them.

For this reason, our second diversity-management recommendation is to use risk analysis to identify process improvement opportunities. After agreeing on what the most serious risks are, software process improvements can be more focused, addressing the most significant problems rather than those that require the coolest tools.

Risk analysis, prioritization, and process improvement must continue throughout the project, so that the process always tracks the current situation. Informal risk analysis means thinking about all of the ways projects have failed in the past and asking whether any of those factors are present in the current project. A risk taxonomy or risk assessment process can be easily tailored to your project.

Armed with the risk profile, you then evaluate tools and techniques. The highest risks will be expensive to address, so they must be subdivided into manageable units. Figure 1 shows two examples of how it can be done.

Risk analysis is by no means the only way to identify software practice improvement targets, but it does lend itself to a greater or lesser degree of rigor. It is also one that few small teams think to apply, even though it is common practice at the level of large projects and organizations. Used informally at the team level, it can be a valuable guide to practice selection.

Plan for diversity

The only way to avoid process diversity is to adopt the perfect process and then never change it. Failing that, the rest of us will be trying out practice improvements on a pilot project or component while leaving other parts of the project or organization alone. Some of the practice improvements selected through risk analysis might become part of a revised foundation process; others will be applied in special circumstances, and yet others will be rejected as inappropriate or unsuccessful.

One approach is to segment your processes based on the software quality factors or “ilities” such as capability, reliability, and usability. The project I described here segmented its processes based on reliability: the flight software had higher reliability requirements so it needed a tougher process. You

Example one

Past situation: A new project member needed weeks of training and code-reading before being able to make even a small change.

General risk areas: documentation, design, and training.

Specific risks: The current design documentation is organized to be more useful as a reference than it is for learning.

Possible near-term practice improvements: Create road map and survey documents that don’t duplicate content but serve as guides for learning. Produce examples of typical tasks. Investigate documentation-management tools that support version control.

Success criteria: Significant decrease in amount of time between joining the team and making useful contributions.

Example two

Past situation: You communicated requirements changes to a junior staff member without discussing the changes with the team; the implementation led to errors.

General risk areas: Requirements stability, requirements traceability, design communication.

Specific risks: Requirements changes are accepted without project review.

Possible near-term practice improvements: Create a simple database or spreadsheet that lists all requirements, even if vaguely stated. Convince requirements-givers that every change must be logged, then use reviews to approve updates to database.

Success criteria: All requirements changes are logged for review and analysis.

Figure 1. Two examples of subdividing into manageable units
could also base your decisions on something like usability: certain projects will have mandatory usability testing, while others will not.

Another approach would be to focus on maintainability: certain projects will produce code that is used and maintained over a long time; others produce one-off, short-term solutions. This is not to say that singling out one software attribute is always the right way to segment processes. The important thing is to consider the impact of each process or technique on all of the software quality factors.

A technique that has worked for us is to define all of the possible process attributes but then identify tiers or strata that define different practice groups for different types of code. Table 1 shows a segment from a multitier process guide.

Table 1 indicates that every code unit must be checked into the CVS repository. It is universally acknowledged that CVS will be the repository and not some other product. Furthermore, everyone agrees that each module’s header comment will conform to a particular standard. We have also decided that buddy reviews are a technique that might be useful on some projects, so we list that in our process database. Then we determine that the foundation process should consider that technique to be optional, while the standard process should always use that technique. The advanced process will also be required to conduct formal inspection, but pre-inspection buddy reviews are recommended.

The process tiers are not strict subsets or supersets of each other-numbers 2 and 3 in Table 1 define variants of the same practice area. In the project described here, we applied the foundation process to one-off experiments and simple feasibility studies, whereas we applied the standard process to noncritical software and the advanced process to critical sections and controls.

There are two important attributes to managing diversity within the parameters of this kind of plan once you have defined each process’s and each segment’s required, recommended, and optional elements. First, you must clearly define which products will use which processes—and who will decide. If you do not clearly define this in advance, crunch mode will reduce every project to the least-strict process.

Second, you must have a mechanism through which one process’s work products can be used by another process. For example, code developed as part of a one-off experiment should not be used in the spaceborne software kernel without some additional reviews and testing. However, we shouldn’t mandate that all experiment code be thrown away either. The right thing to do is to treat code developed under a less-strict process as raw material that must be documented, reviewed, and tested within the more-strict process if it is to be incorporated into a component that requires it. Enforcing this requirement is a key element of day-to-day technical management.

Unfortunately, moving work products from one process to another is not always a black-and-white activity, and its management is not always easy. We found numerous examples of work that drifted across the boundary, mostly through code reuse by engineers.

### Table 1

<table>
<thead>
<tr>
<th>Description</th>
<th>Foundation process</th>
<th>Standard process</th>
<th>Advanced process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Use CVS as repository of all code</td>
<td>Required</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>2 Each module must have conforming header comment</td>
<td>Required</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>3 Use formal ‘B’ language specifications for every method</td>
<td>Optional</td>
<td>Optional</td>
<td>Recommended</td>
</tr>
<tr>
<td>4 Conduct ‘buddy’ review of all code created or changed</td>
<td>Optional</td>
<td>Optional</td>
<td>Required</td>
</tr>
<tr>
<td>5 Conduct formal inspection of all code created or changed</td>
<td>Optional</td>
<td>Optional</td>
<td>Required</td>
</tr>
<tr>
<td>6 Track all labor hours spent in review/inspection</td>
<td>Optional</td>
<td>Recommended</td>
<td>Required</td>
</tr>
</tbody>
</table>
Frameworks (which penalize organizations for locally improved processes).

Organizations that seek higher CMM levels will need to upgrade the minimal process level continually to include new areas of consensus. When seeking CMM compliance, you must evaluate all processes against the relevant key process areas. In the case of ISO, each process must be well defined within the ISO framework with the appropriate documents and records. Each project team member must be aware of which process they are using for each work effort.

For a project to make long-term progress toward improved quality and productivity, all team members must agree on a set of basic engineering practices and use risk analysis to identify advanced techniques. The resulting diversity must be carefully managed to prevent chaos and backsliding.

Acknowledgments

I thank Brad Hines of the Jet Propulsion Laboratory for his help and encouragement, and the anonymous reviewers for suggesting many content and presentation improvements.

References


About the Author

Michael Deck is a software practice consultant at Cleanroom Software Engineering, Inc. His research interests include applying Cleanroom models to object-oriented development, real-time systems, and embedded software. He specializes in training project teams to tailor and use software engineering practices to solve real-life software process problems. He received an MS in Computer Science from the University of Maryland, College Park. Contact him at deckm@cleansoft.com.