Lessons Learned from Tool Adoption[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

Software engineers talk a lot about using tools to help them perform development, project management, and quality tasks. They evaluate quite a few tools and buy a fair number, but they seem to use few tools in their daily work. Selecting and installing a software tool is the easy part. It is much harder to make effective tool usage an integral part of your group's software engineering practices.

Tool adoption initiatives often stumble because people don't fully appreciate the technical, managerial, and cultural factors that influence success. Here are eight lessons I've learned from groups attempting to use a variety of software development tools. Some of the lessons apply to any kind of tool adoption; others are most pertinent to computer-aided software engineering (CASE) tools for systems analysis and design. Keep these lessons in mind to increase the chance that your next tool investment will yield the desired payback.

General Lessons for Tool Adoption

Most tools that are available to the contemporary software developer or project manager belong to one of these categories:

- ***Project management tools***: These help your team estimate, plan, and track schedules, resources, effort, and costs.
- ***Analysis and design tools***: These help you document, analyze, and manage requirements or create design models.
- ***Coding tools***: These include code generators and reformatters and code analysis and reverse engineering tools.
- ***Quality improvement tools***: These include test planning and execution tools, and static and run-time code analyzers.
- ***Configuration management tools***: These let you track changes and defects, control access to files, and build your product from its components.

Most tools save time and reduce errors by automating a part of your development or management process. Tools that let you iterate on designs or accurately repeat tests can lead to higher quality and projects that are more successful. Defect-tracking tools ensure that problem reports do not get lost, and source control tools prevent one programmer's changes from obliterating another's. However, the first lesson to keep in mind is:

---

[1] This paper was originally published in *Software Development*, October 1999. It is reprinted (with modifications) with permission from *Software Development* magazine.

A tool is not a process.

For example, development groups sometimes believe that because they are using a tool to track defect reports, they have a defect-tracking process. In reality, a tool merely supports a process. The process defines the steps that you must perform to carry out some activity correctly. Team members must share a common understanding the steps and the roles and responsibilities of the process participants.

I once implemented a new change control system in a web development group. First, I wrote a change control process, which the team members reviewed and accepted. We then beta-tested the new process using paper forms for a few weeks while I selected a commercial issue-tracking tool. Feedback from the beta helped us improve the process, and then I configured the tool to support the new process.

Introducing tools into a group can be disruptive. New ways of working demand new ways of thinking, and most people aren't eager to be forced out of their comfort zones. To succeed, the team members must agree on some conventions for using new methods and tools. This transformation requires both culture and technical changes. This brings me to lesson two:

Expect the group to pass through a sequence of forming, storming, norming, and performing when it adopts new tools.

Forming, storming, norming, and performing describe the stages through which a new team progresses. These terms also characterize a software group trying to apply new approaches. During forming, the team selects a tool. During storming, team members debate how to use the tool, whether or not it is a good investment, and how to interpret the rules embodied in the tool or in its underlying methods. If you stop at storming, your team will not evolve to an improved future state that includes the tool. Storming is part of the learning curve, the cost of investing in new approaches that promise to yield long-term benefits.

Guide your team through the storming stage, relying on the team members' professionalism and willingness to compromise. It should then enter a norming stage, where the team agrees on how to apply the tool. Some individuals won't always get their way but everyone must be a bit flexible to support the common good of improved team performance. The ultimate goal is performing, at which point the team achieves better results with the new tool than without it.

Any time someone is asked to change the way he or she works, you can expect to hear the question, "What's in it for me?" This is a normal human reaction; no one wants to go through an idle change exercise just because someone else had a cool idea. Sometimes, though, the change doesn't offer an immediate, obvious benefit for everyone. The lesson here is:

Ask "What's in it for *us*?" not "What's in it for me?"

Anyone attempting to introduce new software development methods or tools should be able to articulate the overall expected benefits to the project team, the organization, and the customers. For example, developers might not see the need to use code analyzer tools. After all, it might take them longer to complete their work if they have to run the code through a tool and fix any bugs it finds. However, explaining that it costs several times more to fix defects when the system testers find them than to use the tools during coding is a compelling argument. Quality practices that take one individual more time than his or her current work methods usually have a high return on investment downstream. This benefit might not be obvious to those who feel someone is making their jobs harder.

Tools are sometimes hyped as silver bullets for increasing productivity. Some tools do provide a substantial productivity benefit, as with automated testing tools that can run regression tests far faster and more accurately than a human tester can. However, the benefits from most tools come from improving the quality of the delivered product. Productivity gains come over the long term, as tools help you prevent defects and find existing bugs early and efficiently. Improved initial quality reduces late-stage, defect-repair costs. Any action that reduces rework frees up developer time to create new products, rather than rehabilitating flawed ones. Keep lesson four in mind as you explore increased software development automation:

Tools affect quality directly, productivity indirectly.

I have heard development managers balk at acquiring development tools because they are expensive. "I can't afford a copy of Tool X for every team member at $2,000 per license," they protest. However, remember lesson five:

Weigh the price of the tool against the costs of not using it.

A good example here comes from requirements management tools. Licensing a powerful requirements management tool for a team of 10 software and quality engineers might cost $15,000. However, you don't have to go very far wrong on the project's requirements to waste $15,000 implementing unnecessary functionality or re-implementing poorly understood and ill-communicated requirements. You cannot predict exactly how many errors such tools can help you avoid. Nonetheless, your tool acquisition decisions should consider the potential costs of not using a tool, in addition to the size of the check you'll have to write for it.

Lessons from CASE Tool Adoption

I worked with several small software development groups that used a number of CASE tools over several years (see *Creating a Software Engineering Culture*, Dorset House, 1996). CASE tools let you draw models of requirements and designs according to a variety of standard modeling notations and languages. Commonly used models include data flow, entity-relationship, state-transition, class, sequence, and interaction diagrams. When they first came onto the scene in the 1980s, vendors claimed fabulous benefits from CASE tools. But some of the groups I have seen use CASE tools were simply documenting completed systems. This is useful, but it doesn't help you improve the software you build, only to understand software you've already built. Lesson six is:

Use CASE tools to design, not just to document.

Use the tool's built-in methodology rules to validate diagrams and detect errors that are difficult for people to find. Your team should use the tools to iterate on design models before they start cutting code, because developers will never conceive the best design on their first attempt. Iterating on requirements and designs, rather than on code, is one way to improve quality and reduce product cycle times.

Our team had some energetic meetings in which we haggled over exactly what rules we should follow for various design models. Resolving these issues was critical to successfully implementing CASE in our group. We finally agreed on some conventions we could all live with. Lesson seven from our experience is:

Align the team on the spirit of the method and the tool, not on "The Rules."

Even if the developers do a great job on design, programs rarely match their designs exactly. You need to decide how to reconcile inconsistencies between design models and delivered software. If you want the CASE models to provide long-term benefits to the project, heed lesson eight:

Keep the information in the tool alive.

Ideally, you will update the models to match the reality of the system as it was built, and you'll keep the models current as you modify the system. This approach demands considerable effort and discipline. If the system evolves while the models remain static, inconsistencies between code and designs can cause confusion, wasted time, and errors during maintenance. If you decide not to update the design models, either discard them once they have served their purpose, or clearly identify them as representing the initial design, not the current software implementation.

Some CASE tools permit round-trip engineering—generating code from a design and regenerating design models automatically from source code. This approach guarantees the correctness of the as-built design documentation. You can repeat the model generation any time code changes are made to create a new set of accurate models.

Fitting Tools into Your Culture

A cultural transformation takes place as a team moves from manual methods to the discipline of using structured processes and tools. Facilitate this change by:

- Articulating why you selected the tools
- Acquiring management commitment
- Selecting appropriate tools for your organization's culture and objectives
- Training your team
- Setting realistic expectations of the new technology's future benefits.

Some developers will balk at using new tools, maintaining that they can do better work with their current approach. For example, programmers who use interactive debuggers to perform unit testing (an inefficient process) may resist using test coverage tools. Some skeptics can be swayed when they see their teammates getting better results with the new tools, while others will never accept that there's a better way. Be alert for recalcitrant team members who sabotage the improvement effort by "proving" that the tools are worthless. Look for allies among the developer ranks, early adopters who are willing to adjust their current approaches to try new tools. Merge the tools into the way your team works, rather than reshaping the culture to fit the tool vendor's software development paradigm.

Educate your managers about the value of investing in tools, so they understand the tools aren't just toys to amuse those technology-drunk software geeks. Obtain their commitment to spend the money you need to license the tools, train the team, and accept the short-term productivity hit from the learning curve. Ask management to set realistic and sensible expectations about how the team will incorporate tools into routine project activities.

The best chance for successful tool adoption comes if the tool capabilities will attack some source of pain your team is experiencing. For example, if developers spend countless hours on

tedious searches for memory leaks or pointer problems, they might be willing to try run-time analyzers such as Rational's Purify or Compuware NuMega's BoundsChecker. Begin by identifying the areas where improvements are needed, then go to www.methods-tools.com and search for tools that might fill the bill.

Incorporating tools into a software organization requires more than dropping a user manual on every engineer's desk. Support the new tools with training in the underlying approaches, such as testing concepts or project estimation principles. Don't rely on tools to teach your developers the fundamental methods, any more than you can learn arithmetic by using a calculator. The individuals who I've seen use tools most effectively had a solid understanding of the methods and principles implemented in their tools.

As you pursue the effective use of new software development and management tools, respect the learning curve, which will likely make your first attempts to use new tools actually take longer than your previous methods did. Share your successes and failures with the team, so everyone can learn from the experiences of others. Work toward a long-term objective of providing each of your developers with a robust, integrated automation suite that will increase their productivity and the quality of the software they deliver.