DavidChappell
& Associates

TOOLS FOR TEAM DEVELOPMENT: WHY VENDORS ARE FINALLY GETTING IT RIGHT

DAVID CHAPPELL

SPONSORED BY MICROSOFT CORPORATION

Most software development is done by teams of people. Yet even though tools to support team-based development have been available for some time, they weren't always as useful as they might have been. Today, the vendors who create those tools have reached a consensus on what the real problem is, and they're providing tools to solve it. The result is team development tools that focus on the right thing: optimizing the end-to-end development process.

THE ORIGINAL PROBLEM: COMBINING TOOLS

To understand what today's consensus is and why it's important, it's useful to look first at the history of development tools. Figure 1 illustrates how the tools developers use have evolved over the last few decades.

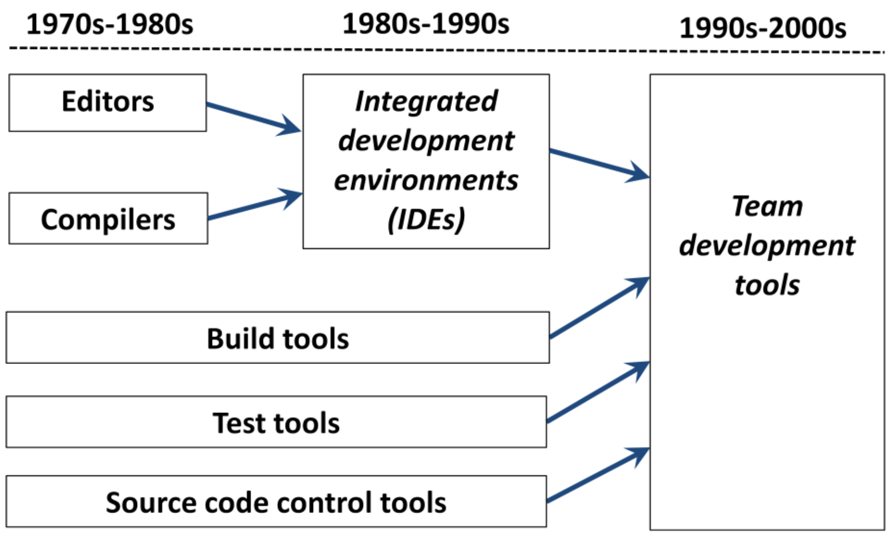

Figure 1: What began as separate tools have evolved into unified team development tools.

In the 1970s, all of the functions performed in the software development process were supported by different tools. Developers created code using an editor, then explicitly invoked a separate compiler when needed. Similarly, the tools used to build complete executables, test code, and manage versions of that code were all distinct from one another.

Over time, these separate tools for software development have been combined. In the early 1980s, editors and compilers were united to form *integrated development environments (IDEs)*. Developers loved IDEs—yoking these previously separate tools together meant, for example, that errors could be fixed immediately right in the source code—and so IDEs caught on quickly. Combining these two tools made sense, because it let developers be significantly more productive.

Over time, what were once separate tools for software development have been combined.

As build tools, test tools, and source code control tools became more widely used, it also made sense to combine them with one another and with IDEs. Just as integrating compilers and editors allowed things that weren't possible when the two were separate, grouping all of these tools together was also a step forward. The result was *team development tools*, an advance that began appearing in the 1990s.

Unlike IDEs, which caught on quickly, organizations were slower to embrace team development tools. This slowness was partly because this kind of unified tool is harder to adopt than an IDE. Moving from a separate editor and compiler to a unified IDE requires only individual developers to change—it's easy. Moving from separate tools for writing and compiling code, doing builds, testing, and source code control to unified tools is significantly harder. More people—and more tools—have to change. Just as important, the development process itself must change.

Making this kind of process change can be challenging, creating various kinds of resistance. People might be reluctant to give up a favorite tool in some area, for example. Another potential roadblock stems from an important benefit of team development tools: their ability to automatically track information about the development process, then generate reports on a project's progress. While this transparency is a great boon for the people who manage the project, it also means that people on the dev team have nowhere to hide. If a developer hasn't checked in any new code in the last week, this problem will show up quickly.

There's also another important reason why organizations have been slow to adopt team development tools: The tools weren't initially as valuable as they might have been. In fact, it's fair to say that, as in most new areas, the real problem wasn't fully understood at the beginning. Today, however, that's no longer true: The challenge has become clear.

THE REAL PROBLEM: OPTIMIZING END-TO-END FLOW

When vendors (and open source projects) first created team development tools, they commonly built or acquired the best tool they could for each area: development, testing, source code control, and so on. This kind of point optimization led to some excellent tools, but it didn't solve the right problem. In fact, the goal in team-based software development isn't creating great tools that optimize parts of the process. It's optimizing the process as a whole.

To see why this is so important, think of a manufacturing process where a part takes three hours to manufacture, then sits in a warehouse for three weeks waiting to be shipped to a customer. Cutting the manufacturing time in half won't make the customer any happier—what needs to be optimized is the end-to-end flow. Similarly, an organization that improves how its developers write and compile software won't see much benefit if, say, testing doesn't also get better. By providing a cohesive environment for creating software, team development tools can help optimize the entire process, not just its individual components.

This idea becomes even more important given how software development has changed over the last few decades. Developers once waited an hour or more to get the results of compiling new code, then waited days or weeks for meaningful test results. Today, creating, compiling, and testing code happen continually—the iterations are much shorter—and the transitions between them are more frequent. Making the process better requires making these transitions as smooth as possible.

Doing all of this requires tools that work together well, with a common way to share information across different aspects of the process. Just gluing together good tools in each area won't work. What's needed is an approach like that shown in Figure 2.

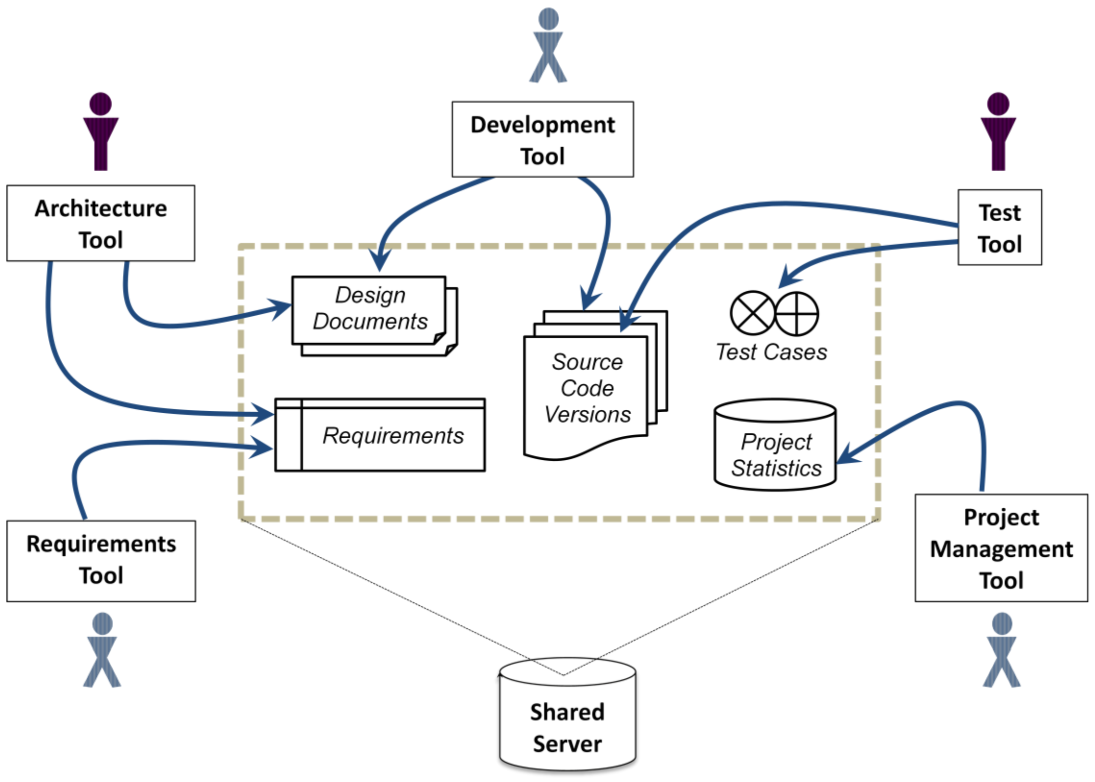

Figure 2: Optimizing the development process as a whole requires a unified approach to storing and working with the artifacts of that process.

Tools used for different purposes—working with requirements, specifying architecture, developing code, testing code, and project management—should all be able to work with a common set of interconnected artifacts stored in a common place. As Figure 2 shows, those artifacts can include requirements, design documents, various versions of source code, test cases, statistics about this development project, and more. This kind of integration allows all sorts of useful things: automatically recording code check-ins, associating test cases with requirements, generating historical reports of bug counts, and more. More important, it allows optimizing the entire process. The transitions between different functions all rely on shared information, and so making those transitions gets easier.

When our industry converges on an architecture, it means there's broad consensus about the best way to do something.

For the most part, the first generation of team development tools didn't take this approach. The vendors have learned from their experience, however, and team-based tools today can provide this broad integration. Microsoft's Visual Studio, for example, provides tools for architecture, development, testing, and more, all of which rely on Team Foundation Server (TFS) to store the artifacts they work with. IBM's Rational offerings take a similar approach, with different tools supporting different functions and Jazz Team Server acting in a role similar to Microsoft's TFS.

When our industry converges on an architecture, it means there's broad consensus about the best way to do something. This consensus has appeared in team development tools. The vendors have figured out that the real goal is optimizing the development process as a whole, not just individual parts of that process.

CONCLUSION

This is an important moment in the evolution of development tools. Since team development tools are now focusing on the right problem, they have more to offer than they did ten years ago. Organizations doing team-based development—a category that includes almost everybody—can benefit from taking another look at this style of tool.

Still, don't expect adopting team development tools to be as easy as adopting IDEs. The challenge of clearly understanding (and perhaps changing) your development process still exists. Nonetheless, just as IDEs made life easier for individual developers by combining what were once separate tools, team development tools can help developers and everyone else on a development team work together more effectively. Given how important software is to most organizations, improvements in how we create it are always welcome.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technology.