# Beware the IDEs: The Risks of Standardizing on IDEs

Articles - Agile CM Environments

Written by Steve Berczuk, Robert Cowham, Brad Appleton

Monday, 16 March 2009

*Beware the ides of March (Shakespeare Julius Caesar Act 1, Scene 2)*
*Beware the IDEs...* -Steve, Brad, and Robert
*Individuals and Interactions over Processes and Tools* -- The Agile Manifesto

Two topics that are likely to launch a development team into an impassioned discussion are development standards and development environments (IDEs, editors, etc). Combining the two topics into that of standardizing on development environments, is even more likely to spark debate. Decisions about development tools affect the day to day workings of each person on the team as well as the productivity of the team, and as such are important to discuss as a team organizes itself.

When making these decisions it's important to consider the reasons for standardizing, how much you need to standardize, and how to balance productivity in an IDE with accuracy in a build. There are three questions that a person who is concerned with configuration management issues needs to address when considering how IDEs fit into the development ecosystem:

- Does the team need to standardize on a specific IDE (if any) and related tool set?
- What, if any, IDE configuration artifacts should be placed under version management?
- What is the balance between the goals of a productive development environment the requirements of the integration and production environment?

This article will make some suggestions to help your team answer these questions and be more productive as a team and as individuals.

**Development Environments (Interactive and Not)**
There are many productive ways that developers work, ranging from a simple text editor plus command line combination (NotePad in Windows being one example), to highly customizable editing environments (emacs, with JDEE ) to standalone Integrated Development Environment (Eclipse , IDEA , NetBeans ). Developers in any of these camps can either work with whatever comes out of the box or with environments highly customized to their work style. We know developers who have personal emacs configuration files that have been maintained and moved from project to project for years, and developers who switch between IDEA and Eclipse at will.

A discussion on tool standardization applies to both text-editor based environments and IDEs. We'll focus the discussion on IDEs, as IDEs are typically more feature-rich, and are more likely to involve external configuration. The key question of how a development environment affects team and individual productivity remain the same.

**Productivity and IDEs**
IDEs can increase productivity for individual developers by enabling developers to maintain Flow. Flow is a hyper-productive mental state where a person is fully immersed in what they are doing. Agile development practices support flow by working to eliminate non-productive work and to focus the team on tasks that add value, and IDEs also support this goal.

IDEs help maintain flow by allowing developers to focus on the intent of their development level tasks rather than mechanics, giving

the developer more rapid feedback on the results of tasks such as refactoring -- enabling a developer to perform refactorings such as move method by indicating what code the change will affect rather than having to perform the mechanical copy, paste, and the search for compile errors elsewhere in a project--, and working with other, supporting tools such as SCM and Issue Tracking systems via plugin mechanisms. IDEs also allow you code without thinking too much about coding style conventions. A properly configured IDE can either format the code as you type, or it can let you format it after. You need not worry about tabs or spaces or alignment, or other peripheral tasks. This allows you maintain focus and avoid changing contexts to perform your primary tasks.

IDEs improve feedback, identifying errors in your code as you type, avoiding the interruption in flow that a full compile/build/test cycle can cause. While a full compile build test cycle will give you an accurate assessment of the state of your application, if it takes longer than a few seconds, it will interrupt your thinking. To allow for a more rapid incremental build compile and run scenario the IDE environment is not always exactly the same as the full build process that runs in your integration environment and which you use to deliver the application, and there are risks lurking in these differences. If your IDE uses a build configuration that is different from the one used in the Integration Build, you need to work to keep them in synch. You also need to be careful to avoid the "works in my IDE but not in the Integration Build" scenario, by running a Private Build in your workspace before committing changes, as we discuss in The Importance of Building Earnestly.

Given the benefits of IDEs, the fact that different tools have different advantages and disadvantages, and the desire to control the divergence of IDE and build configuration, many teams want to standardize on a single IDE. Standardizing on Development Environments raises some interesting challenges. Moving beyond the simple features that an IDE provides a team, individual productivity is closely related to how team members adapt tools to individual styles. IDEs allow developers who are skilled in the uses of a particular tool to learn how it works, and customize the tool to fit their work style.

For all the benefits of customizations, differences in configuration can make it more difficult for another individual to contribute to the same code. The temptation is to simplify the problem by standards. Since there are other standard tools on your project, would it make sense to add IDEs to list?

**One IDE for All?**

When considering development tools and processes, standardization often enables simplification. Some tools adress the constraints of the problem the team is working on. The set of languages and version of compilers, for example, may defined by the target environment. Some tool standards constrain the problems that a team needs to solve. Since no tool is perfect for all situations, a standard build tool set means that you can fix a problem once and have consistent results for all team members. Sometimes a tool standard comes out of simplicity. Picking Maven as your build tool for a Java project gives you a certain degree of integration between build scripts and IDEs that, say, Make may not. And having a standard compiler and build toolset give you a definition for "working software" which is "works in the integration build when you use these tools. The SCM Patterns, such as Private Build and Integration Build are based on being able to reproduce, with a reasonable degree of fidelity, an integration build in a developer workspace.

Some standards, like coding standards, don't have a large degree of impact on how a developer works, assuming that there is a mechanical way to apply them. Once the team agreed to standards they can be configured in an IDE and someone writing code can spend energy deciding how an algorithm should work rather than where the opening brace for a method call goes. Build tool and coding style standards simply define end points in an executable way (ie, tests) and it's possible to set up your surrounding tools in a way such that developers can work in whatever way is most effective, as long as their code passes these tests.

IDEs are a bit different. Having a standard IDE simplifies the problem of deciding what tool a novice developer on the team should use. When in doubt, just use the standard one. But what about the people on the team who are expert in a particular tool? If the point of an IDE is to improve productivity, forcing such a standard on everyone can result in a decline of productivity for someone who is an expert in the other tool. Often, when one switches between, say, Eclipse and Idea, there a period of time where it was difficult to work in either tool because of subtle differences in approach. This is surmountable, and learning a new tool is a useful experience, but it has a cost.

Adding to the complexity of IDE selection is that IDEs have a variety of strengths and weaknesses. Even if you use Eclipse for most of your work, you may find that the GUI builder in NetBeans is so good that you might want to use NetBeans for part of your Swing Development, or the profiler in NetBeans if that works well for you. *Requiring* a standard IDE for all work would cut you out of that

option.

If developers are considered experts in their craft, why impose a tool that affects something as basic as how a developer works with her code on a minute to minute basis. In other professions, experts use tools that they are the most comfortable with. Musicians are fussy about not using an instrument other than their own. Even musicians who play less portable instruments, Bass, Piano, will go to great lengths to attempt to use their instrument, or one similar. Why expect less of developers? If someone is hyper-productive on IDEA, what is the benefit of having them context switch to Eclipse? In Pragmatic Thinking and Learning , Andy Hunt comments about the difference between experts and novices as it related to standards and tools: Novices don't really care, and having a standard helps. Experts know their tools well. While a novice can select the *refactor->move method* context menu, an expert in IDEA will enter the shortcut (F6) getting to the interesting work more quickly. The same IDEA expert will still know to refactor a method if they are using another tool. It will just be more slow.

When trying to decide what tool is best we're often inclined to confuse "familiar" with "better." It's in the interests of everyone on the team to be cautious of this tendency.

Closely related to standard IDEs is the question of how quickly and reliably one can set up a new environment. Organizing your setup around a standard IDE seems simplest, as you can version the IDE settings, but that isn't always the best approach as we'll discuss next.

**The Build's the Thing**

Regardless of whether or not the members of your team use one or many IDEs, Integrated Development Environments and their configurations are not identical to the build scripts that teams use to run their integration builds. In Java environments, classpaths may differ, working directories for the IDE may be different than the build environment's defaults. In some cases, differences in the compiler mechanism may lead to different results that differ from the build. These differences are OK as long as you understand them, and treat the build script as the canonical definition of "it works," as we discuss in Building for Success.

How, you ask, do you reconcile the requirements that you be able to create (IDE) workspaces quickly and also maintain the fidelity of the build? There are a few approaches. The simplest one is to maintain two sets of configuration files. The build, and the IDE configuration. Maybe you have one person who maintains the IDE settings, (perhaps one for each IDE) and keeps the files in synch if there is an issue. This means duplication, and manual work. Fans of lean software development will see duplication as waste. Fans of any process will see a manual step as a place where errors can creep in.

Another, better, way is to generate the IDE build related settings from the build files, and not version the build files (with one exception). Many modern Java IDEs for example, can use generate their configurations from Maven or Ant build files. And there are Maven plugins to generate IDE configurations from Maven build files. Keep the build files as a primary artifact, and generate the derived artifact that is the IDE configuration.,

The nature of how teams work with IDEs may lead toward workspace configurations and configuration approaches that differ from those that make the most sense in a production environment. One useful aspect of IDEs is that they are customizable to fit the working style of the person using them. In many cases, individual customizations are not suitable for everyone on the team, but that may not matter.

**Performance, Individuality, and Agility**

In agile teams, people tend to work more collaboratively, in shared physical spaces, and in tighter cycles than on a traditional project, so being able to have similar environments may more important than in other environments. An agile Team using Pair Programming could potentially benefit most from a common tool set. So a natural question in the context of agile teams is whether pair programming requires that the team standardize on tool set. If any environment would require standardization, Pair Programming would, as there are many perceived efficiencies in moving between developers' workstations as if they were identical environments.

While in practice, some pair-programming teams do standardize on IDEs, many others don't require a single IDE. If you are following the Driver/Navigator model for Pair Programming, you use whatever IDE the driver prefers, and the second half of the pair will quickly pick up enough of how the IDE works to perform basic tasks. Even if you are trading the keyboard freely, the differences between environments still allows for basic collaboration, and the diversity of IDEs provides a chance for team members to learn about other

tools and provides a path for learning and improvement.

This raises a basic question about the tradeoffs between delivering value quickly as a team, and allowing individuals to work in the way that is best for them. Agile methods value Individuals, so it stands to reason that unless there is a very good reason for mandating a tool over one which a productive team member sees a strong advantage, one should allow for flexibility.

**What to Version?**

Everyone reading this, we hope, has their build scripts (pom.xml files if they are using Maven, build.xml if ANT, Makefiles, etc) under version management. In the interests of efficiency, you might suggest that IDE configurations should also be under version management. To see if this makes sense, let's think about the different types of IDE configurations: Build settings, and tool settings. Build settings are, in effect, what is in the build configuration files. maintaining these for IDEs is duplication, and it's better to generate them from the build.

Tool settings are a reasonable thing to maintain and version if there exists the possibility for user customization of settings that don't really matter to the whole team, but which improve developer productivity.

Since IDE files are often redundant to build scripts, if the IDE files are not maintained to be in snch with the build scripts, team members will learn to ignore the project files in version control. You want to avoid a situation where team members ignore versioned files because they don't want their files overwritten. This defeats the purpose of versioning the files and leads to confusion. Versioned files should add value to the project.

Don't version IDE configurations, that related to build path issues. Use tools to keep IDE settings in synch with the build scripts. Avoid any duplication. It may seem like a small amount of work, but manual effort to maintain IDE files is rarely high value. Maven provides tools to generate Idea and Eclipse configurations from POM files. Idea and Eclipse both have plugins that can work off of maven pom files to synchronize build and IDE settings.

Tool vendors (both IDEs and script based tools) should consider the ability to synchronize between IDEs and scripts to be a high priority as IDEs and other build tools serve complimentary purposes, and are of the most value when their configurations match as much as possible.

You can benefit by versioning configurations relating to style which can be universally used by all team members. But keep the build scripts the source of build and dependency configuration as much as you can.

**Bridging The Gap between IDE and Build Tools**

The Private Build pattern gives the most concise guidance for how to ensure that a build is synch: Run a build in your workspace with the same toolset that works in the Integration Build. And a Continuous Integration process, running the Integration Build Patterns, provides a backstop for when you goof. Incremental builds are fine in an IDE when you are coding, and the instant compliation that occurs after a change helps you to maintain flow when you code. But there can be subtle, and not so subtle, differences on the build environments, from the specific compiler implementation, to path configurations, that mean that what works in your IDE may not work in your build. And Broken Builds slow everyone down.

As long as the contract of the integration build being successful is honored, we should not need to overly constrain how developers work.

**What to Do?**

Interactive Development Environments (IDEs) are a common part of a developer's tool kit. IDEs are highly customizable, and increase productivity, and help enforce coding standards. Some developers have very strong opinions about what the best one is. Every team struggles with development standards. Standards affect communication (coding standards) and execution (tool standards). Standards can help a team be more effective by eliminating discussion on matters that are less important to the end result, and by enhancing communication by establishing a common style and language. Too many standards, or standards applied to appropriate concepts can have the effect, reducing productivity and causing the team to spend a disproportionate time coding to the standard rather than delivering value. Let's wrap up by offering some recommendations on the questions we raised at the start of this article.

We Recommend:

- Allow team members to work with whatever development environment toolset that is most productive for them, given the constraints of the project.
- Define your workspace set up process in terms of the least common demominator: getting the files on disk using command line tools for example. This will help establish consistency with the Integration Build environment. (more on this in our article Private Workspaces: where Development Process Meets CM Process) .
- Much like we would suggest for other requirements, focus on defining end results not mechanisms. If indentation is important, specify the rules, not the IDE. Recommendations are fine, but if someone is more productive with a different toolset than the recommended one, it should be fine.
- If something is really important, add it to the build. There are plugins that will run style checks (both cosmetic and functional) at build time, generating warnings or failing the build if something is amiss. The Integration build is the one point in the process where you are guaranteed a consistent environment, so use it as a gate.
- Version only primary artifacts, and use tools to generate secondary ones, such as build settings for an IDE, where feasible. Primary artifacts includes build files, but not IDE configuration files relating to build settings. If the IDE has the ability to version style related artifacts independently of the project configurations, these should be treated as versioned artifacts.

The measure of whether the toolset works is whether the code meets the standards set by the build, and whether people using other tools can work easily with the code. Don't require a specific IDE tool set as long as the core development standards can be supported by the developers tool of choice.

Vendors need to think about the individual and team aspects of IDEs --ensuring that personal SCM tool settings are not checked in for example to make this possible.

The question of tools and standards can be controversial. The simplest solution that balances team and individual joy and productivity is to focus on whether the standard leads to more customer value, and to define standards in terms success criteria (how quickly to set up a workspace, features of the code, etc) rather than mechanisms.

There is a difference between consistency in important things, which is valuable, and conformity, which is often mistaken for consistency. Focus on delivering consistent results, and respect that a team will know how to get there.

---

***Brad Appleton*** *is an enterprise SCM solution architect for a Fortune 100 technology company. Currently he helps projects and teams adopt and apply agile development & SCM practices. Brad also author's the Agile CM Environments blog, and is co-author of Software Configuration Management Patterns: Effective Teamwork, Practical Integration, the "Agile SCM" column in CMCrossroads.com's CM Journal, is a regular contributor to "The Agile Journal", and is a former section editor for The C++ Report. Since 1987, Brad has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at brad@bradapp.net*

***Robert Cowham*** *has been in software development for over 20 years in roles ranging from programming to project management. He continues his involvement in development projects but spends most of his time on SCM Consultancy and Training. He is the Chair of the Configuration Management Specialist Group of the British Computer Society, has a BSc in Computer Science from Edinburgh University and is a Chartered Engineer (CEng MBCS CITP). You can reach him by email at rc@vaccaperna.co.uk*

***Steve Berczuk*** *is an agile software developer and consultant.. He has been developing software applications since 1989, often as part of geographically distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book Software Configuration Management Patterns: Effective Teamwork, Practical Integration and a Certified ScrumMaster. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at steve@berczuk.com*

# Trackback(0)

TrackBack URI for this entry

# Comments (0)

Subscribe to this comment's feed

# Write comment

Last Updated ( Friday, 20 March 2009 )