Extreme Programming with Visual Age for Java
By Joshua Kerievsky

I used to be a die-hard user of a programmer's editor. You know the type: I used my editor on every project, for a wide variety of languages and could often be heard saying things like "IDEs are for wimps", "my editor handles *any* language" and "who needs a debugger? The code I write isn't very buggy." Today, I'm a changed man. My tool of choice for solo or team-based Java programming is IBM's Visual Age for Java Enterprise Edition (version 3.5). I've found that VAJ is currently the best tool on the market for doing Extreme Programming (XP).

Before I explain further why VAJ is so good for XP, I'd like to get a few things out of the way. 1) I have no connection with IBM so this article isn't a disguised marketing message. 2) This isn't a comprehensive look at all that VAJ provides – it provides a lot, but I'm just interested in showing you how it helps us do XP efficiently. 3) You may've tried VAJ and disliked it. That happened to me too, in 1998. But the folks I was programming with back then didn't really know how to use the tool effectively, so my perspective on VAJ was warped. When I later programmed with folks who really knew how to use VAJ, I was stunned to discover how great it was. So perhaps my experience will help you approach this tool with an open mind.

VAJ is particularly great at version control, which is vital for Extreme Programmers since we collectively own our code (anyone can change anything at anytime), continuously integrate our code (many times a day), and mercilessly refactor our code. With changes happening so frequently, it helps to be able to quickly, easily and safely view, compare, or revert to different versions of a method, class, package or project. You can do that with nearly all of the popular version control packages, some of which even integrate into your editor or IDE. But using those tools is awkward, compared to what you get with VAJ. VAJ makes version control easy by providing good, built-in version control tools that you can access with a few mouse clicks. I recall being stunned the first time I discovered that VAJ was saving every version of a method I was working on (see figure 1). This feature allowed me to comfortably make changes to the method, knowing that I could easily revert back to earlier versions with a minimum of effort. Merging code is also easy in VAJ, since it provides convenient comparison views of different versions of a package. That makes it easy to do something hard, like figure out how your code changed from the latest version of the code and how it changed from the version you started with.

VAJ's version control is quiet sophisticated, with capabilities to support many logged-in users, locking for this or that artifact, and tracking to know who worked on what. But on VAJ XP projects, we only use a fraction of these features. Because we practice collective code ownership, everyone logs in to VAJ using the same user name (typically "XPUser"), we never lock any section of code, and, most importantly, we only version at the project and package level, letting VAJ automatically assign version numbers to our classes and interfaces. This simple way of using VAJ's version control has worked great both for me as a solo programmer and when I've programmed with teams from 8 to 25 in number.

Let's now talk about incremental compilation. In most IDEs or programmer's editors, you edit a file, save it, and then invoke some command to compile the code or re-build everything that needs to be compiled. In VAJ, every time you save a method, the compiler kicks in automatically for you and does its job. So there is no "build" menu item in VAJ, no "rebuild" option, not even a "compile" option. Incremental compilation just happen every time you save a method, variable declaration, class declaration, import statement, etc. This feature enables time-savings magic. For example, say I save something which the incremental compiler sees has a problem. VAJ warns me and gives me the option of fixing the problem now, or of adding the problem to the Problems page and fixing later. If I choose to fix the problem(s) immediately, VAJ attempts to suggest possible solutions (like including "import java.util.*" because I'm using the ArrayList class and haven't provided an important statement for it). If I choose to ignore the problem for the moment, VAJ saves the code but places an X icon next to the name of the field, method and/or class containing the problem. These Xs make it easy for veteran and less experienced programmers to immediately see how one change to some code can effect other parts of the code. Compare that to what happens if a programmer makes many changes to some code, the saves, and then compiles, only to find a mountain of errors. Does that sounds familiar? Incremental compilation embodies XP's value of "feedback" – it gives programmer's feedback as fast as possible.

Incremental compilation also does more. Let's talk about how it impacts debugging. When I was still a fairly new VAJ user, I was pair-programming with a women at Evant, an ASP in San Francisco. We were running some JUnit tests, one of which had just failed. A quick look at the code did not reveal the solution, so we decided to use the debugger. We set a break point in our test code, told JUnit to re-run the test and in a few seconds we were positioned on our breakpoint, ready to debug. We stepped into a method, then another method and a few steps later we were in the method that contained the bug. We were on line 4. One of us spotted the problem on line 2. That's when my partner simply fixed line 2, saved (think incremental compiler doing its magic), and then choose a wonderful little option in the debugger that allowed us to re-enter the method as if we had not entered it yet! Our test code ran successfully and I was stunned. We had just changed a line of code *while debugging*, re-run only the method we had changed (not the whole program) and seen JUnit report that our test ran successfully. It's worth noting as well that VAJ has excellent support for debugging servlet-based code, since it can create two separate JVMs, one for the client and one for your servlet.

Refactoring – or improving the design of code without changing its behavior -- is something Extreme Programmers do all the time. But doing it manually can slow you down. So VAJ has an excellent refactoring plug-in tool called jFactor (by Instantiations), which seamlessly integrates with VAJ (and Borland's Jbuilder). When you click on a class, or method or highlight some chunk of code, this tool will provide you with a list of possible refactorings. Choose one refactoring, and the tool will ask you for some information and then safely perform the refactoring for you. The only downside to this tool, at the moment, is that it doesn't support undo and isn't cheap. Nevertheless, I'm addicted to using it since it speeds my refactoring work.

VAJ has excellent searching capabilities, including the ability to easily find all callers of a method or find the methods and fields referenced by a method. And, like so

many IDEs today, it has a code-assist feature to help you remember which method to call, and what parameters that method expects.

Ok, so I've talked a lot about how VAJ helps us efficiently do XP. But what are its downsides? Well, the enterprise version, which gives you team-level version control, isn't cheap. True, but you have to look at the time you waste by not using a powerful tool like VAJ and how much that costs you (which is the same argument you must make for a tool like jFactor). VAJ also likes to have a lot of memory well – and that can increase your hardware budget. VAJ isn't a speed demon at running Java. So if you need super-fast execution *during development*, you have to make some choices. On one Extreme Programming project, we found that our tests were taking nearly 40 minutes to run. Granted, we had a lot of test code, but this slow execution time was unacceptable since it was causing cues to form at our integration machine. We got around this problem by writing some VAJ code to export any changed code from our VAJ projects to the file system, where we could then run the tests under a fast JRE. That substantially improved performance. Another downside is that VAJ locks you in to a certain level of the JDK. So if you need some new feature that Sun just added to a latest release of the JDK, you are out of luck. I found that this was a problem in VAJ's early days, when for instance, it didn't even support the JDK 1.1 inner classes. But these days, not as much is being added to the JDK, so it hasn't been a problem on projects. Another shortcoming of VAJ is that it alphabetizes your code. So if you want all of your constructors and field declarations to be at the top of your class, they won't be if they don't all start with the letter A. This is quiet annoying, particularly when you export code from VAJ to share it with others.

Fortunately, IBM has been addressing nearly all of the above shortcomings with an exciting new project and product called Eclipse (http://www.eclipse.org). Eclipse, which has been programmed purely in Java by Erich Gamma (of the Design Patterns Gang-of-Four fame) and a team of talented programmers, is a *free*, sophisticated IDE and an open-source platform on which the public may create their own tools, IDEs, or Eclipse plug-ins. You are no longer locked into any particular JDK and Eclipse includes VAJ's sophisticated version control features, but it adds these features on top of popular packages like CVS or ClearCase (instead of on IBM's proprietary code). Erich and company have already written an excellent JUnit plug-in for Eclipse, and common refactorings, like Extract Method, are included, are undoable, and, because of the open-source nature of this project, should grow in number. Code-assist support is improved over the current VAJ and general editing is good, though definitely not at the level of a tool like IntelliJ. Speed of the tool is comparable to most, though I don't have a lot of experience using it yet. Overall, I'm very excited about Eclipse as an Extreme Programming development tool and a successor to Visual Age. The free price tag also sounds nice.

TABLE
Best Practices for doing XP with VAJ

- Have everyone log into VAJ with one user name (like "XPUser")
- Create one project for your code and one project for your test code
- Create open-editions of your projects at the start of an iteration and version them at the end of the iteration

- Version at the project and package level – don't' version classes or interfaces (let VAJ do so automatically for you)
- If your tests aren't running fast enough, run them externally from VAJ using a fast JRE (Java Runtime Environment). Export from VAJ only your changed code before running the tests (you'll need to write some VAJ code to have it only export your changed code, but this isn't that hard to do and will ultimately save you time).
- Don't rely too much on VAJ's dialog boxes for creating fields or methods –for example, if you're currently positioned on a method and want to add a new method to a class, just delete the text from the current window, start typing the new method and save. I find that VAJ's dialog boxes are useful sometimes, but often just slow me down.
- Use an automated refactoring tool like jFactor – to speed up your refactoring work