IDEs & Build Scripts

Working together in harmony

by Steve Berczuk

ISTOCKPHOTO

Interactive development environments (IDEs) are powerful productivity tools for software developers. While Java developers often work in Eclipse or IntelliJ IDEA (two popular IDEs), many projects also use automated integration build tools, such as Maven. The use of two different tools can cause inconsistencies if not well managed. This article discusses the risks that can occur when IDEs and build scripts diverge and provides guidelines for keeping the two views of the development project consistent while maintaining the relative benefits of each.

Let's begin with an example scenario:

Jim is very productive with Eclipse, his favorite IDE. To implement a new feature, he makes a change to a Java source file, runs the unit tests through the IDE, and sees a green bar indicating that all the tests passed. Since everything seems to have compiled and the tests ran, Jim uses the IDE to commit the changes to the source code repository and then takes a break for lunch.

When he returns from lunch, Jim notices an email from the integration build system saying that there was a compile error. Looking at the build log on the integration machine, Jim realizes that he forgot to update his IDE configuration to reflect the team's decision to update a third-party library. The method calls in the code that he commited to the repository were deprecated (declared obsolete) in the prior version and are no longer valid in the current version. Simple enough to fix. Too bad that anyone who updated his workspace while Jim was at lunch had to debug a broken build.

Across the hall, Mary is engaged in a heated discussion with Phil, the release engineer. Phil wants to use Maven's resource filtering to configure a properties file at build time. Mary is arguing that this is a bad idea, as the resource filtering puts a meaningless token in the source configuration file, which makes her IDE-based debugging settings not work.

In this scenario, there is a mismatch between the configuration of the developer's IDE and the configuration of the integration build. In Jim's case, the IDE settings were not updated to show the current state of dependencies as reflected in the Maven build scripts. In Mary and

Phil's case, an idiom (resource filtering) that works quite effectively in the integration build environment doesn't match the way that a developer works within an IDE. These are two examples of how the differing ways that developers configure IDEs and configure build scripts can cause disfunction and conflict in the team. Fortunately, there are approaches to reconcile these differences.

The IDE

IDEs can increase productivity for individual developers by enabling them to maintain *flow*, a hyper-productive mental state in which a person is fully immersed in what he is doing. Development practices support flow by working to focus the team on tasks that add value and to eliminate distractions, such as non-productive work.

IDEs help maintain flow by allowing developers to focus on the intent of their development tasks rather than the mechanics. This gives the developer more rapid feedback on the results of tasks such as refactoring, which enables him to perform refactoring tasks such as "move method" by indicating the code to be moved rather than having to perform the mechanical copy, paste, and search for compile errors elsewhere in a project. IDEs allow you to code without thinking too much about coding style conventions. A properly configured IDE can either format the code as you type or let you format it afterward. You need not worry about tabs, spaces, alignment, or other peripheral tasks, allowing you to maintain focus and avoid changing contexts to perform your primary tasks. IDEs also interface with other supporting tools, such as software configuration management and issue-tracking systems, via plug-in mechanisms to simplify their use.

IDEs improve feedback, identifying errors in the code as you type and 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, it will interrupt your thinking and disrupt flow if it takes longer than a few seconds. 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, with an IDE using an incremental compiler, or different paths to resources, for example. 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 keep them in sync to avoid the "works in my IDE but not in the integration build" scenario.

The Integration Build

An automated integration build verifies that your code is working after every set of changes to the source code tree, allowing you to identify issues quickly so that they can be fixed closer to the introduction of the problem.

You see the real benefits of an integration build by using continuous integration, which enables you to increase the rate of feedback so that you can detect problems quickly after a change is made. A continuous integration environment polls the version repository every few minutes—waiting for a short period of time after the changes to let the system settle—and performs a build if files have been updated. This avoids problems where, for example, someone forgets a file during a commit, remembers it afterward, and then adds that file. This short wait reduces the chances for false-negative build failures.

A build on an integration server has several advantages over a build on a developer machine:

- The build takes place in a controlled environment, so you know that the build works in the "defined configuration." Developer machine configurations have the potential to vary minute by minute.
- The build integrates changes directly from the source code repository, validating that the build reflects all changes up to the time of the build. For example, if a developer forgets to commit a change to a file, the code in his workspace will build, but the integration build will fail. Or, if a developer makes a change that conflicts with another developer, the integration build will quickly detect the problem.
- The integration build can be a source for identification (build numbers, build labels) that can be applied and referred to consistently.

The integration build provides an additional protection against developer mistakes and misunderstandings. When a team practices continuous integration, "working code" means "works on the integration build" in that the code compiles in the integration environment and passes all unit and integration tests specified to run as part of the build.

Even with this definition of "done," compiling and testing in an IDE isn't enough to guarantee that the build works; the IDE configuration must match the build configuration.

Overlapping Functionality

Because of differing requirements, IDEs, such as Eclipse, and build systems, such as Maven, have different configuration mechanisms. An IDE typically maintains project files in its internal format. Maven defines projects using an XML-based project-object-model (POM) file. The information is similar but stored

in different places. While some IDEs can derive their configuration from the Maven project files, inconsistencies can still remain. To ensure that the code is consistent with the build tool, always use the command line to build. However, it isn't practical to use the command line to build the entire project after an edit.

Because development teams will want to work inside an IDE, it is essential for project-related configurations to stay in sync between IDEs and build scripts. The configurations that need to be synchronized between IDEs and build tools include:

- Inter-component dependency information—for example, that the "UI" project depends on the "model" and both depend on the "core"
- External dependencies (name, version, etc.)—for example, knowing that the core project depends on "Apache Commons" version 2.4
- Build-time prerequisites—for example, if the "UI" project depends on a "persistent-model" that is based on code generated from meta-data

Keeping these dependencies synchronized not only helps avoid checking in broken code but it also allows the IDE to provide accurate feedback when checking syntax and executing tests.

Synchronizing

Modern IDEs, such as Eclipse and IDEA, can synchronize with Maven project files to ensure that dependency information is correct. Unfortunately, even after synchronizing with a project file, there are other settings, such as coding style preferences and debug and run configurations, that must be maintained manually because they may be customized to the user's environment. A challenge for teams is to decide whether to maintain IDE configurations in the same way that build project settings are maintained—under version control or not.

If your team uses a single IDE, maintaining project files under version control sounds appealing. It's easy to get a new developer workspace up and running by simply checking the IDE configuration out of your source code management system along with the source code. If you version your IDE configuration files, you need to decide how to maintain IDE configuration. Two choices are:

- Ad hoc—The developer changing a build script also changes the IDE. This approach helps ensure that there is responsibilty for updating the IDE configurations. However, since there is no automated way to verify the IDE configuration's correctness, a forgetful person has no backup.
- Assigned responsibility—Have a toolsmith be responsible for updating the IDE configurations. While this approach helps to ensure that the configurations are kept up to date, the toolsmith is essentially manually translating the information into build script in the IDE configuration. This involves duplicate work that can be of low value.

In many cases, adding a requirement that one IDE configuration work for everyone means that there are restrictions on how developers can customize their IDEs. While it's useful and appropriate for teams to develop standards for style and how code should function, developers can be most effective when they are allowed to decide how to work. Since IDEs are individual productivity tools, you might find yourself reducing productivity in an attempt to increase it by maintaining a common configuration.

On some teams, IDE configurations are checked in for convenience and developers check them out, customize them, and then make an effort never to check them in again. The IDE configurations are only in the source code management system for reference. While this seems to address the question of how to get a developer up and running quickly, asking team members to use as a starting point a resource that is not maintained can cause more problems than it solves. Any file in your source code management system that isn't used by everyone as the definitive source for information should be evaluated for marginal value.

Since an IDE is a key part of a developer's toolset, it is not essential that IDE configurations be turnkey. As long as the IDE can import project dependency information from a build configuration, a developer will quickly learn to configure any other settings he needs to be productive.

Balancing Tools

Another common issue in teams that heavily use IDEs is that techniques that work well in a build environment may not work transparently in an IDE. For example, if your project uses code generation, you need to generate the code (using the build tool); otherwise, your IDE will give errors when you reference the generated code. While the default build mechanisms for IDEs work well, IDEs also allow you to customize how projects are compiled. You can add a step to the build to run a Maven phase before building your code. For example, Eclipse lets you define project builders and IDEA lets you define Maven goals to execute before running the debugger.

Remember that an IDE is a tool you use to create an application, but the final application will be constructed using a build tool. Don't sacrifice functionality and value to the automated build because it doesn't work immediately in your IDE.

What To Do

Since the definition of "done" is "works in the integration build," use the build configuration as your primary source for project definitions and don't maintain derived sources in version control.

Use an IDE that allows you to define a project in terms of the build script mechanism. IDEA and Eclipse allow you to import a Maven project file. Alternatively, there are tools that create IDE configurations from build configurations. The Maven-Eclipse plugin and Maven-IDEA plugin allow you to generate Eclipse and IDEA project files from your Maven POM files.

While some configurations, such as those related to debugging, may not be derivable from the project definition used in the build, it still may be easier to provide guidelines for developers to customize their environments once they have imported the build configuration. Even developing internal tools to set up an IDE configuration from a project definition is more cost effective than manu-

ally maintaining two definitions. Since IDEs are extremely customizable, make the requirements of the automated build environment primary.

As a final step, establish a policy in which each developer is required to run a command line build before committing changes to the repository. This step helps ensure that any discrepancies between the IDE and build environments are detected immediately rather than after the commit.

While IDEs are useful tools—the tools that developers spend the majority of their development days working in—developers should be aware of the capabilities of their build tools and use their IDEs to support integration build, rather than optimizing their process for what works best in the IDE.

Postscript

In anticipation of making some code changes, Jim updates his source tree and opens up his project in his IDE, which is configured to synchronize from the Maven POM file. He makes his changes and runs tests, but before committing his changes, Jim runs a final build using Maven from the command line. Since his IDE was configured from settings in the build scripts, it works perfectly. Jim checks in his code, waits for the message from the integration machine that all is well, and then goes to lunch.

After talking to the release engineer, Mary learns how to have her IDE run the Maven process-resources goal before a build. Having learned the power of resource filtering, she gets to work on making her code more modular and configurable.

By viewing the build tool as the primary source of definition for a project and using the IDE as a developer productivity tool to keep the build running, teams can work effectively and reduce duplication of effort. **{end}**

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- Continuous integration
- Tools