Improving your Agile Continuous Integration Build Process

A technical whitepaper demonstrating the benefits of using Openmake Meister in agile development.

Table of Contents

Problems with Scripted Builds in Agile Development

The heart of agile development is a continuous build.

Agile development implies an iterative development methodology. The basic concept is that you build and deploy applications quickly and on a frequent basis. These mini, iterative releases allow for each developer's coding change to be compiled into a complete application on a continuous basis. The benefits include identifying early problems due to source code branches, deviations in shared components as well as end enabling testing and review by end users.

At the heart of an agile development process are continuous builds. The purpose of a continuous build process is to assemble the application as a complete unit, integrating source code changes as soon as possible. For this reason, builds are executed continuously on a remote machine and initiated when new source code modules have been committed to the shared source repository.

Meister supports agile development by providing adaptive and dynamic build services.

Because agile development results in an application that is changing at a rapid pace, the need to update and maintain redundant manual build scripts becomes a core part of the agile process. *The problem is clear; the agile development process is highly adaptive and dynamic. However, the heart of the agile process, the continuous build, relies on manual redundant build scripting*.

Build scripts are static documents that are snapshots of an application at a single point in time

Because they are manually written there is redundancy within the scripts themselves. A single team member called the "Build Meister" is responsible for managing the continuous build. Developers copy scripts between themselves and edit the scripts to reflect their unique requirements. The Build Meister will create and maintain one large script that calls each sub script sequentially or will write a single script to handle the build of the entire application. Redundancy is found at multiple layers. Each developer may maintain a script as well as the Build Meister. This scripted process was born from the waterfall approach to application development and does not retrofit well into agile development.

Build scripting is not adaptive or self documenting

When a developer updates or fixes a script, the Build Meister must discover that a change in a script has occurred to ensure that it is incorporated into the continuous build. In addition, when the build breaks due to a scripting error, fixing one script may not fix the build as the script may have been copied and used by other developers. A build break may mean re-visiting many scripts in order to completely fix the problem. Removing the redundancy between the scripts minimizes broken builds and useless

script coding. A community driven continuous build does not need to rely on repetitive manual scripting. This whitepaper will explain how Openmake Meister better supports the needs of agile developers by automating many aspects of the build process that would otherwise be manual.

This whitepaper describes how Openmake Meister:

- *Performs build integrations between the individual IDE build and the team CI build.*
- *Speeds up builds using Build Avoidance*
- *Supports a community developed knowledge base for eliminating redundant tasks.*
- *Creates an adaptive code re-factoring process outside of the IDE*
- *Automates and manages on demand, continuous or scheduled builds*

Build integrations between the Individual IDE Build and the Team CI Build

Meister synchronizes the IDE and CI builds for continuous, non-stop build processing.

IDEs allow developers to make rapid changes to source code. While coding, developers use their IDEs, such as Eclipse and Visual Studio, to point and click their way through the process of calling build engines to assemble their code into binaries. However, as they integrate their code with the other developers, they must leave the comfort of the IDE and move to a "command line" build process outside of the IDE. To do this they must manually create build scripts to call the IDE build engines in an attempt to repeat or mimic the build that was performed by the IDE.

In today's fast changing development environment where agile methodologies and continuous integration encourages change, this static, non adaptable build process has problems. First, agile developers update source code rapidly inside the IDE, but the static build scripts that exist outside the IDE have no reference to the changes made within the IDE. The build scripts themselves represent what the source code looked like at a particular point in time and become immediately out of date as source code quickly changes. A simple code refactoring within the IDE can cause a major re-write of the supporting static scripts. Secondly, as applications grow and complexity, the ability to manually define how the code should be assembled becomes more and more complex. For these reasons, an automated process that connects the development IDE automatically to the build scripts is necessary for keeping the build scripts themselves up to date. This automation allows developers to meet the demands and tight release schedules more quickly they face each day, without being bogged down by manual scripting.

OpenMake Meister solves these problems by mashing up the individual developer's IDE build with the builds executed outside the IDE. Meister uses innovative Build Services in conjunction with the IDE project file to generate the build scripts and to mash-up the IDE build with the CI build - automatically. Meister's Build Services are templates that are completely customizable and can be written to support any development requirement. The Meister community developed knowledge base includes standard templates that have been written and shared by the Openmake Meister User community.

Because Meister manages the build engines, such as MSBuild and Ant, it can also tightly manage what is occurring in the build process itself. This includes what is called Build Forensics where each artifact used in the build is logged, even when the artifact is not managed by a source code control tool. Build Forensics provides the most detailed build auditing possible, providing developer's complete transparency as to what source code and libraries were used for any one build. This same Build Forensics is also used during the build to support Build Avoidance, meaning that when a build executes, objects that are up to date or not rebuilt, providing the fastest build speeds possible.

Improved CI Build Speeds through Build Avoidance

Meister knows what to build, and when to build it.

10 minute builds are possible with Meister.

Improving the speed of builds increases the number of CI builds that can be performed during a single day. With applications growing in size and complexity, build times are increasing as well. Builds that run for 2-3 hours are no longer unusual. In fact, some builds run well beyond the 2-3 hour norm.

If your working in a continuous integration build environment, you need your builds to execute on an iterative basis, knowing what to build and when to build it. If objects are up to date, then they should not be rebuilt. This is what we call build avoidance.

Meister supports a build process that is as iterative as your agile source code development process. Using an innovative technology that performs deep dependency discovery and build forensics, Meister can accurately re-build an application without the need for rebuilding all components each time. This means that a build that normally takes 2-3 hours to build can be reduced to just a few minutes.

By allowing developers to execute CI build using build avoidance, the build process becomes as agile as the source code development process. Developers can make updates to source code and compile them immediately against the CI build.

Community Developed Build Knowledge Base – Eliminating Redundancy

Meister eliminates the redundancies commonly found in build scripting.

Agile developers can eliminate build script redundancy using a community developed build knowledge base. Using a centralized knowledge base provides a many to many communications process that eliminates the redundancies commonly found in build scripting.

The three most redundant components of a build scripts are:

- compile flags/parameters
- directory locations
- source code dependencies

A change in any of these three components can require that each script be revisited. When these components are not carefully orchestrated between scripts, the result is an inconsistent continuous build. An inconsistent build is a serious problem for the agile development process, as the continuous build represents the heartbeat of agile development.

Meister allows everyone the ability to address build problems as needed to ensure that the continuous build does not break.

Compiler Setting – Think Global Work Local

Compiler flags and parameters are normally coded as part of a build script and are referenced in multiple locations within each script. The problem with this ad hoc scripted method is that it requires visiting each script to understand what flags are used and how they are defined. A better method is to eliminate the redundancy within the scripts and store this build metadata in a centralized location. This allows everyone on the team the ability to understand the build configuration, address the problems as needed and ensure that the continuous build does not break.

Meister creates this "think global but act local" environment by managing and exposing the critical compile information outside of the script. Using Meister, developers can quickly review the globally used compile parameters and determine if their local change may result in an incompatible parameter that may cause a dependent object to break. Developers can manage their local parameters with the full understanding of what is needed to create the application overall. Viewing and updating compile parameters with Meister is as simple as generating a report or using the Meister interface to access the knowledge base data. **Figure 1** shows a sample report that reveals the global compile parameters for the "Java" Meister Build Types.

BUILD TYPE	OPTION	PARAM	DESCRIPTION	INHERITANCE	TASK	OPTION REQUIRED	USE PARAM	PARAM REQUIRED	OPTION SET
+ Available Tasks Templates									
- Java Compile									
	debug=	true	If true, compile source with debug information.	Build Task Options	Ant Javac	FALSE	TRUE	FALSE	DEBUG
	destdir=		Indicates the directory, relative to the build directory, where the class files should be stored. If it is not used, the class files will be stored in the build directory.	Build Task Options	Ant Javac	FALSE	TRUE	FALSE	RELEASE/DEBUG

Figure 1 – Java Compile Parameters

Coordinating the use of compile parameters creates a build process that is as iterative as your development process.

Online Collaboration of Object Directories

Agile developers make the best use of shared technology. From the use of SOA objects or SOA dependencies such as AXIS (Apache SOAP Communication) agile developers understand the importance of developing their applications on a common framework. What is often overlooked is that the development versions of the 3rd party objects must match what is running in production, otherwise a production release will fail. For this reason, it is important to eliminate the redundant references to the location of these objects.

In Java development most of these objects are found via the CLASSPATH definition set for each developer machine. For SOA development, WSDL (Web Services Definition Language) definition files, used to generate source class files that define the transaction services that interact with the database, are found in local or shared directories referenced explicitly in scripts or defined inside of an IDE.

Meister provides an easy way for agile developers to stay in sync.

Challenges in managing object directories are not completely obvious. First, the CLASSPATH can be defined in as many as three different locations for each developer The definition can be set at the machine level as an environment variable, via an IDE such as Eclipse or hard coded inside the Ant script itself. Because of the redundant ways the CLASSPATH can be defined, it is very difficult to ensure that the CLASSPATH definition is the same for all developers.

As for the use of objects such as those created by SOA WSDL, each developer points to a directory that contains the versions of these WSDL generated source class files via the IDE or within their script. Similar to the CLASSPATH setting, it can be difficult to ensure that each developer points to the correct directory, and in some cases, this may not be possible as each developer updates the WSDL as required.

Meister provides an easy way for the agile development team to stay in sync by removing the redundant references to the CLASSPATH and WSDL source

Meister supports a pro-active process for resolving build issues before they occur.

directories. Meister creates a central on-line environment for defining the CLASSPATH and WSDL directory locations. For developers using the Meister IDE plug-in, their build will automatically use the centrally defined CLASSPATH and shared object directories. It will also allow them to reference first their private directories if they themselves are creating new objects that are eventually to be included as shared objects. When the automated continuous build is executed, the same process is repeated ensuring that the exact objects that were used in the local build are used in the global build.

This centralized, community developed process of defining the directory locations of shared objects eliminates build problems that are found when developers use many definitions stored in multiple locations. Meister further supports this pro-active process of resolving build issues before they arise through the process of gathering and reporting on the build dependencies themselves.

Bulletproofing Dependency Management

The holy grail of any solid build process is the ability to quickly resolve the exact objects that went into the build. The ability to perform accurate dependency analysis is claimed by many Source Code Management (SCM) and build tools. However, the only proper way to perform dependency analysis is through code scanning and compiler "listening". Traditionally, dependency management was done manually with the "Build Meister" coding both high level (.c) and low level, embedded (.h) dependencies into the make script. Programs such as "make depend" provided some help by scanning the source code and automatically generating the low level portions of the make file. As make became replaced by Ant, Build Meisters either explicitly coded the dependencies or chose to use wild cards (*.class), to replace the manual effort of hard coding the dependencies.

The holy grail of build management is dependency gathering.

Build processes which include an interface to SCM solutions can generate a "bill of material" reports that list all the files that were checked out of the SCM tool, or were residing in the directory where the build actually occurred. This process is a "best guess" method as there is no way of actually knowing if the files checked out or residing in the local build directory were actually used by the compiler. The build script itself often defines the directories to be used in the build with no reference to what is actually stored in the SCM tool.

Some dependency gathering solutions attempt to solve the dependency gathering process by providing "listeners" that work at the file system level. These listeners watch to see what files have been opened during the build and are far better than a simple check-out "bill of material" report. The problem with these listeners is that they work only on a virtual file system. This means that in order to get an accurate listing of dependencies the agile development team needs to have all of their directories managed by a virtual file system. If a directory is included in the build, but not managed by the virtual file system, the dependency is left unexposed.

Meister solves this problem in two ways. First, Meister goes back to a more traditional approach like "make depend". Regardless of the development language, Meister performs file scanning, mining a dependency list at the lowest level. Secondly, because Meister does not rely on ant or make scripts to actually perform the build, it has the added benefit of being able to watch the compiler activity and confirm what files are being used to assemble the final objects, even when the files do not reside in an SCM repository or on a virtual file system. This level of dependency scanning gives you a complete and 100% accurate picture of all the dependencies used to create the deployable objects. This information can be linked into the binary as a footprint, or viewed in an online report that can be checked into your SCM tool along with the production ready binaries.

The magic behind Meister is its reusable, adaptive scripts.

Meister's build services create the adaptability needed to support a quickly changing agile development process.

Dynamic Continuous Build Scripts

The magic behind Meister is its reusable, adaptive scripting delivered through Build Methods and Build Services. In addition to Meister's community developed knowledge base for the management of build meta data, Meister provides the framework for agile developers to reuse build scripts. These reusable build scripts are generated based on standard templates and the build meta data that is stored in the knowledge base. Once generated, the script can be executed across the enterprise on any machine, on demand, scheduled or continuous.

Writing and maintaining non-reusable build scripts is a redundant and non-adaptive process. An update to a local script does not equate to an update to the global script. Finding and fixing a problem in a single script that had been copied and used by another developer means that each copied script must be re-visited.

Meister's build services eliminate script redundancy by allowing you to write a script once, for a particular type of compile, and reuse it for every build for that compile type. Meister uses the reusable script to generate what would normally be coded using Ant/XML or Make. This means that a correction in the global script results in a correction to the local script. In addition, Meister uses the community developed knowledge base to include in the generated script the correct compile flags, override compile flags as defined by each developer and the community shared source code directory locations. The reusable PERL provided by Meister supports this build reuse. The reusable PERL modules can be written for any type of compile or build action. Meister's build services create the adaptability needed to support a quickly changing agile development process.

Adaptive Code and Package Refactoring using Meister

Meister simplifies code re-factoring via its IDE plug-ins.

Changing a package or moving classes between packages is referred to as code or package re-factoring. When this is done through an IDE such as Eclipse, the moving and renaming of all objects is performed for you. However, when building outside of the IDE, the build scripts still reference the old package structure. If the build script is written using wild cards, this problem is minimized. However, using wild cards increases the potential of using obsolete or incorrect java source. For this reason, many agile teams use wild cards at a lower package level to produce a more precise build result; therefore package and code re-factoring must be dealt with at the build.xml level. Re-factoring a build.xml file requires that the build.xml script be edited and updated manually by each developer and the Build Meister with the new package structure. This process is far from the dynamic adaptive methods for which the agile developer strives.

When a developer is using Meister's adaptive script generation, once the re-factoring has been completed inside the IDE, the developer simply uses the Meister plug-in to update the build meta data that results in a new build.xml. This is done dynamically without the need to manually visit any ad hoc Ant/XML scripts.

Build Workflow Management

Meister provides full automated build management to support an infinite number of ways to execute the build workflow.

Builds incorporate many tasks and steps. Even though the core of the build is compiling and assembling the binaries themselves, there are other pre and post tasks that need to be incorporated in a continuous or team level build. In addition, these "build workflows" need to execute on multiple machines, coordinating machine hardware and resources, calling on other lifecycle tools such as source code control, testing and deployment.

In addition to executing steps that surround the build, builds are launched in different ways. For example, builds are executed on a shared location and are performed continuously with a build listener executing a build when fresh code has been added to the shared location. Builds of course are executed on demand at the local developer level and finally builds are scheduled to run at different times of the day on dedicated build machines.

These options and process, how the build is launched, what machines are used and the workflow of the individual build steps, are often referred to as build management. This level of control offers the benefit of centralizing the activities around the build and provides a single location to report on the success or failure of the build. Cooperation between development, testing and production release is simplified as all team members go to one location for the unique answers they need.

Meister provides full automated build management to support an infinite number of ways to execute the build workflow. Meister supports remote builds and remote build workflows with scheduling capabilities. In addition, these workflows are developed through Meister's community developed knowledge base. Developers can define build workflows and can share them across the team as 'Public' workflows.

The build workflows themselves can be coordinated and managed by chaining the steps and defining dependencies between the steps. And for developers who must rely on manually developed Ant or Make scripts, Meister will execute the scripts providing bill of material reporting, build difference reporting and build performance reporting.

From continuous build support to a production level build workflow executing across multiple remote machines, Meister provides the tools necessary to manage build automation from start to finish.

Conclusion

Meister offers agile developers maximum protection from failed builds and releases by providing improved traceability and adaptability.

Meister provides build services that allow a dynamic and adaptive method for managing builds from the most detailed level of compile management to the broader Application Lifecycle workflow requirements. Through its build services, Meister eliminates the redundancies normally found in ad hoc scripting. For developers who are working within agile methodologies, Meister's community-developed knowledgebase provides reusable templates which are sustainable roadmaps for the software build challenge; without relying on static, homegrown scripts. Meister offers agile developers maximum protection from failed builds and releases by providing improved traceability, adaptability, impact analysis and audit control. In summary, Meister improves software quality and speed by leveraging reuse and consistency in the build-to-release process.

For more information, visit us at www.openmakesoftware.com or reach out to us at request-info@openmakesoftware.com

Company Overview

OpenMake Software started the evolution of builds in 1995, serving mainly the financial community with the mission of delivering a 100% insulated build process that were also fast. The OpenMake Software team understood the ins and outs of software compiles and links, and how easily a build could be the bottleneck of the software delivery process and be easily compromised on accident or on purpose. With this mission in mind, OpenMake Meister was created and has been serving large enterprises for over 25 years, the longest serving solution in the DevOps ecosystem. Meister has been sold and distributed by Broadcom for over 20 years.