Building Eclipse Java Applications with Meister®

A White Paper and technical overview of build management techniques using the Eclipse Java IDE integrated with Meister

OpenMake

www.openmakesoftware.com

Introduction

This document presents a discussion of how Meister integrates into the Eclipse Project's Eclipse IDE for Java and extends the build process.

This document has two broad sections. The first set of sections discusses problems with builds in Eclipse, the Meister solution, and the Meister Eclipse plugin. The second set of sections details how developers use the Meister Eclipse plugin.

Managers, SCM Leads and Development Team Leads may be interested in the first four sections. Developers and those familiar with the Meister concepts may want to jump to the section "Using Meister with Eclipse Development".

The Meister Eclipse plugin integrates with the development process through the Eclipse IDE and is based on technology from the Eclipse Project (http://www.eclipse.org). The discussion that follows below is in terms of Eclipse 2.x, but it is applicable to Eclipse 1.x.

Overview

In the first section, we discuss challenges faced in building Java applications outside of the Eclipse IDE. We provide a general background of the build process within the Eclipse IDE and general solutions for outside builds.

In the next two sections, we present an overview of Meister where general terminology, features, and process are detailed.

In the fourth section, the Meister build process for Java applications is introduced. We discuss the setup and actual build steps used by Meister.

The final section provides details on the Meister Plugin for Eclipse, the key integration point between Meister and Eclipse. Typical example applications provide case studies for configuration and usage. We discuss how Meister takes the build outside of the Eclipse IDE and how the build process can be extended to provide additional functionality.

Challenges Building Java Applications without Meister using Eclipse

The Eclipse IDE provides "point and click" build support in two steps, first, the creation of .class files and second, the creation of deployable archive files (.jar, .war, and .ear). Builds are organized based on Workspaces allowing the entire Workspace to be built or allowing for the build of a single project within the Workspace.

For the creation of .class files, the user selects "Build" from the IDE menu to call the built-in Java Development Tooling (JDT) plugin. The JDT simply compiles .java files into .class files, incrementally and in an efficient manner. In order to create the deployable archive files (typically a .jar file), the developer points and clicks through the use of the built-in Export functionality.

Because the JDT and the Export functionality are managed through the IDE, the developers are shielded from the build steps. Information used to control the build such as project interdependencies and classpath may be “closed” and not readily accessible outside of the IDE.

Development efforts are managed within Eclipse based on Workspaces. A Workspace is a directory that contains objects associated with Eclipse projects. The Workspace also has associated metadata. This metadata includes developer preferences, and build information about how the objects in the workspace get built and packaged. Each developer can define the metadata uniquely. This uniqueness means that the metadata associated with the workspace is specific to each developers workstation.

In a team setting, each developer has their own version of the workspace, and each developer’s metadata is the main difference between developers’ workspaces. Each developer’s workspace contains a full source code tree. Developers coordinate the sharing of code by importing new versions of the code into their workspace as they see necessary based on team verbal communications or through the use of a version management tool.

When builds occur, each developer builds their version of the workspace. Because each developer’s workspace can contain different source code and different workspace metadata, developers cannot confirm that a build of the workspace, regardless of developer, will produce the same results.

Build challenges are encountered as the Eclipse Java application moves through the development lifecycle from development into production. The following challenges must be overcome in order to ensure a repeatable, consistent build process at the developer workspace level and at a final production build level:

- Each developer is responsible for maintaining the correct level of code in their workspace. When a production build is required, inconsistencies between developer workspaces may cause the build to break.
- Developers may branch the source code managed in their workspace without realizing that there is a potential integration issue between shared code. These issues may not be realized until the final production build, often too late in the lifecycle to provide a simple fix.
- A single build expert must be used to manage the build and coordinate the team’s changes.
- In order to do the production build, the build requires a machine with the Eclipse IDE installed.

- The build expert must determine the correct version of the source code and manually confirm the correct settings of the workspace metadata to produce a stable version of the deployable objects.
- All builds are dependent upon the Eclipse IDE in a point and click fashion. The point and click method requires multiple build steps.
- The workspace metadata including the buildpath information, the build machine and the build expert is critical in reproducing a single build.
- The ability to build parallel development efforts or support builds across a lifecycle (production, testing or emergency) becomes limited due to the dependency on a specific machine with a specific workspace configuration as well as the dependency on a particular build expert to perform a manual point and click process.

To avoid the point and click interface to the build process, most development teams will resort to creating Ant Build scripts to compile their Java application. This process will have one of two possible downsides:

1. Manual maintenance: Developers must maintain Ant build.xml scripts. These scripts are then passed along to the Team Lead or “Build Master” to be integrated into a full build. This leads to passing the build “back and forth” between the Build Master and the Developers in order to get it right.
2. Developers write custom Java classes to interact with the Eclipse IDE in “headless” mode. As suggested by IBM for Websphere Application Developer, and applicable to Eclipse, an an “outside” build using Eclipse can be accomplished by using the IDE in “headless” mode. Following is an excerpt from the WSDD Redbooks:

    > … [I]t would be best if the outside build could simply ask an Application Developer project to build itself using the existing project buildpath information. Such a build does not require the GUI to be running. You can have Application Developer launch itself “headless” and run a specified task, typically an Ant build. You need to create a wrapper HeadlessAntRunner that extends AntRunner and attaches a HeadlessAntListener as part of its run method. …
    >
    > *“Using Ant with WebSphere Studio Application Developer -- Part 1 of 3”* WSDD Library Support Downloads Redbooks Newsgroups All of IBM; Barry Searle

    Performing builds in headless mode only replaces the requirement of having a developer point and click through the process. All other challenges remain the same, *i.e.*, team coordination of the Eclipse metadata, management of a specific build machine and heavy dependency on a particular build expert. This solution also adds the new challenge of maintaining build specific Java classes in order for the Eclipse IDE to integrate with Ant and the creation of a build.xml Ant/XML script. Integration between Ant and Eclipse is not simple.

The Meister Solution

Meister addresses the challenges that developers face when building their Eclipse workspaces in a team setting from development to production. Meister is an enterprise based build management product designed to standardize builds from development through production. It is a unique tool in that it provides a standardized method to rebuild any executable module based on a platform type without being tied to a particular development machine. Following is an explanation of how Meister resolves the critical challenges facing WSAD developers when building their applications.

Challenge:

Each developer is responsible for maintaining the correct level of code in their workspace. When a production build is required, inconsistencies between developer workspaces may cause the build to break.

Meister Solution:

Meister resolves this problem because it does not rely upon the workspace metadata. Instead, Meister manages Target Definition files (TGTs) that report the dependency information needed for the build. These TGTs can be shared, centralized and automatically updated depending on the development team requirements.

Challenge:

Developers may branch the source code managed in their workspace without realizing that there is a potential integration issue between shared code. These issues may not be realized until the final production build, often too late in the lifecycle to provide a simple fix.

Meister Solution:

Meister uses a Dependency Directory to allow developers to perform Unit builds against an approved build. This allows developers to build the code in their workspace against the last “approved” build. Because builds can be scheduled nightly, this more closely supports extreme programming where the continuous integration of source code is pursued.

Challenge:

A single build expert must be used to manage the build and coordinate the team’s changes.

Meister Solution:

The build knowledge base replaces the build expert. All build information is stored and managed by the Meister Knowledge Base and in the Meister Target Definition files (TGTs).

Challenge:

In order to do the production build, the build requires a machine with the Eclipse IDE installed.

Meister Solution:

Meister does not require the Eclipse IDE installed. It does not use the Eclipse IDE in “headless” mode. Meister comes with extensible Perl scripts that generate, dynamic, Ant/XML files. The generated Ant/XML files then call the appropriate Java tasks.

Challenge:

The build expert must determine the correct version of the source code and manually confirm the correct settings of the workspace metadata to produce a stable version of the deployable objects.

Meister Solution:

Meister integrates with version management tools to determine the correct version of source code to be included in the build. In addition, the workspace metadata is replaced by the Meister TGT files and Knowledge Base.

Challenge:

All builds are dependent upon the Eclipse IDE in a point and click method. The point and click method requires multiple build steps.

Meister Solution:

Meister’s command line features allows you to automate the execution of the Eclipse Java build. No additional Java coding or Ant scripting is required.

Challenge:

The workspace metadata including the buildpath information, the build machine and the build expert is critical in reproducing a single build.

Meister Solution:

Meister does not rely on workspace metadata as all target information is derived from the workspace and saved in a Target Definition file (TGT). The combination of the TGT and the Knowledge Base allows for a repeatable build process across any workstation and executed by any person.

Challenge:

The ability to build parallel development efforts or support builds across a lifecycle (production, testing or emergency) becomes limited due to the dependency on a specific machine with a specific workspace configuration as well as the dependency on a particular build expert to perform a manual point and click process.

Meister Solution:

Meister can dynamically generate a build for any level of the application based on an Meister Project and Dependency Directory. A developer can build any level of any Eclipse Java Project from the command line or from the Meister Eclipse/Eclipse/WSAD plugin.

Meister Technical Overview

Meister is an enterprise build management solution to standardize builds from development through production. It is a unique tool in that it provides a standardized method to rebuild any executable module based on platform type without the dependency of a particular development machine. This is critical for development teams during day-to-day compilations, as well as the change and configuration management teams during the production turnover process.

Since Meister automates the entire build process and tracks dependencies between application components, the production turnover process can easily incorporate the rebuilding of all source modules turned over by the development teams. Development teams can easily implement nightly production builds without the need for a designated employee (the build master) who is dedicated to managing the application system Ant build.xml file.

This ease of implementation is of direct benefit to the development teams. As we will see below, the Meister Eclipse plugin transparently manages the build metadata within the Eclipse IDE. This metadata can then be used external of the IDE to build the application. This means that the development build follows the exact same process as the build that will eventually be deployed in production, thus reducing the confusion and "back-and-forth" between the various teams responsible for getting an application to production. If the Meister build works when invoked by the developer from the IDE, the developer knows that it will work for the other teams in the software lifecycle.

Meister Projects

An Meister project defines a set of build targets and the directory locations in which all source files for all the project's build targets can be found. Meister projects typically correspond in a one-to-one relationship with the Project Workspaces.

The Meister Knowledge Base

Meister separates common build information from critical project specific information. Common build information is managed in the Meister Knowledge Base, while application specific information, that is likely to change over time, is managed in Target Definition Files (e.g. target-dependency relationships).

The Meister Knowledgebase contains information on:

Build Types, Tasks and Rules

Build Types, Tasks and Rules contain all of the compiler rules and compile flags necessary to create a target from a dependencies of a particular file type.

For Java, Build Types correspond to a final archive file type (`.war, .jar, .ear`). Build Tasks correspond to an Ant Task used to execute the given step, such as:

- Ant Javac: used to compile `.java` → `.class`
- Ant Jar: used to jar the `.class` files and resources together

Project Dependency Directories

Source directories defined to a particular configuration of an application including all source code, libraries and Meister target definition files.

Build Machines

A machine that is used to perform local or remote builds. Build Machine meta data contains information on remote build machines that may be used to build different components of a project.

Groups and Users

Objects that store information on the users and their corresponding groups used to organize the availability and presentation of the Knowledge Base

Target Definition Files

Target Definition Files, identified by a .tgt file extension, are defined by developers and indicate build targets and high-level dependency information for those targets. These Target Definition Files are used to generate the Meister Build Control files. Unlike other build methodologies, a target definition file does not require scripting of any kind by developers. It is simply a target name, Build Type and high-level dependency list.

The Meister Eclipse plugin automatically creates the Target Definition Files for Eclipse Java Projects. The plugin parses the Eclipse metadata and converts it to the Meister TGT format. With these autogenerated TGT files, the build can be invoked internally within the Eclipse IDE, but more importantly, externally on a remote machine, with little developer interaction.

Using Meister with Eclipse Development

Meister provides full support of the use of Ant tasks when building Java-based applications. When using Meister with Eclipse, Ant tasks are derived based on the Eclipse project file. Developers work within the Eclipse IDE with Meister monitoring the changes being made to the project dependency through the Meister Eclipse plugin. By monitoring the developer's work activities, Meister can maintain accurate Target Definition files. These TGT files are then used outside of the Eclipse IDE to perform project builds at any release level or state of a development lifecycle.

This section provides an overview of how to set up and execute builds using Meister both within and outside the Eclipse IDE. The key integration point is the Meister Plugin, which:

- directly gathers information on the contents of the Eclipse Java Projects;
- helps the developer set up a build using Meister.

Plugin Architecture

The Meister Eclipse plugin introduces a new type of Eclipse Project to the Workspace: The Meister Build Project. This Build Project contains the Meister-specific components of the build. The Build project has two subdirectories by default:

- `build/` The location of the Meister build output.
- `tgt/` The location where the Meister Target Definition files are stored

Build metadata is stored in the Meister Build Project. This metadata includes

- References to Eclipse Java projects to be built through this Build Project
- The Meister Project, Dependency Directory and Build Job (stored on the Meister KB Server) corresponding to this Build Project.
- Options for generating TGT files within the Build Project
- Options passed to the `bldmake` and `om` programs when starting an Meister Build.

These features are discussed in more detail in the following sections.

The plugin will monitor developer's modifications to Eclipse Java projects, updating the relevant Meister TGT file within the relevant Meister Build Project. When invoked, the plugin will execute an Meister Build for the targets in the Build project.

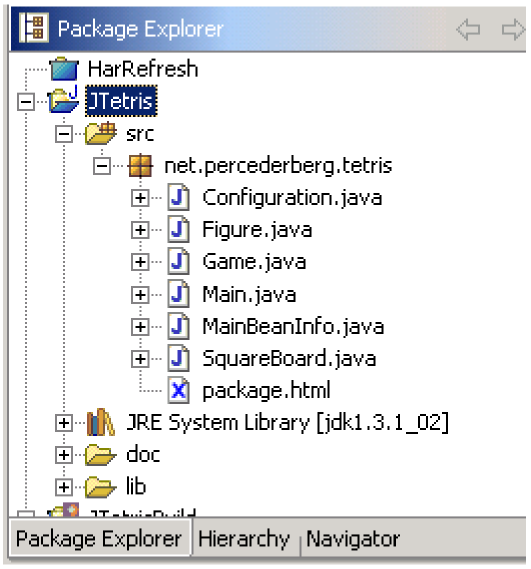
Eclipse IDE Assumptions

The developer should already be familiar with Eclipse development processes. The example below uses Java code for a Tetris game from Per Cederberg:

http://www.percederberg.net/home/java/tetris/tetris.html

The code is available under the GPL.

An Eclipse Project called "JTetris" and imported the source code has been created for this example. The source folder has been set on the build path to "JTetris/src". After creating the project, the filesystem looks as follows:



```
Package Explorer
  HarRefresh
  JTetris
    src
      net.percederberg.tetris
        Configuration.java
        Figure.java
        Game.java
        Main.java
        MainBeanInfo.java
        SquareBoard.java
        package.html
    JRE System Library [jdk1.3.1_02]
    doc
    lib
```

Figure 1

Meister Installation Assumptions

The Meister Knowledge Base Server is installed and running properly on a centralized server to which the developer's workstation has network access. The Meister Command Line Client is installed on the developer's local workstation.

For the JTetris project, we created an Meister Project on the Meister KB Server. This Meister Project is called "JTETRIS" and has two Dependency Directories: "DEVELOPMENT" and "INTEGRATION."

The Dependency Directories for the “DEVELOPMENT” Dependency Directory are shown in Figure 3. The various environment variable settings will be discussed later.

Meister Knowledge Base Setup

In order to build an Eclipse Project using Meister, it must be associated with an Meister Project. Go to the Manage Projects menu option from the main menu on the Meister Web Client to associate the Project.

Meister Dependency Directories must be defined for each Meister Project. The suggested Dependency Directory configurations are:

For the “development” Dependency Directory that will be used to build in the WSAD Workspace:

```
.
$(TARGET_DEFINITION)
$(WORKSPACE_ROOT)
[Third-Party Library Directory]
[Java Home]/lib
[Java Home]/jre/lib
```

The environment variables `$(TARGET_DEFINITION)` and `$(WORKSPACE_ROOT)` are dynamically set by the Meister Eclipse plugin when a build is executed in Eclipse. These variables are set to location of the TGT files for the Meister Build Project (`typically <Workspace root>/<Meister Build Project>/tgt` and <`Workspace Root>`, respectively).

This searchpath can be used to do builds external to the Eclipse IDE by setting the environment variables `$(TARGET_DEFINITION)` and `$(WORKSPACE_ROOT)`.

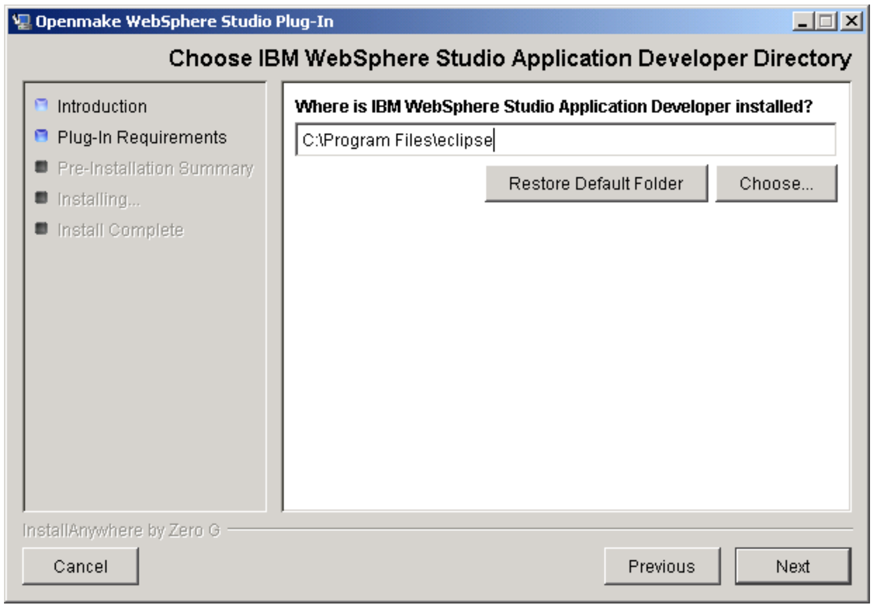
Outside of Eclipse, an “integration” level build should refer to an SCM-managed set of directories:

```
.
[SCM-Managed Directory of workspace files]
[SCM-Managed Directory of workspace files]/<Meister Build
Project>/tgt
[Third-Party Library Directory (SCM-Managed)]
[Java Home]/lib
[Java Home]/jre/lib
```

Installing the Meister Plugin for Websphere Developers

The Meister Plugin for Eclipse can be found in the WSAD directory on the Meister installation CD. Execute install.exe to initiate the install process.

The installer will attempt to automatically detect which version of WSAD is installed (4.x or 5.0) and the installation directory. For Eclipse, these directories obviously do not exist. Instead, simply select the root level directory of the Eclipse installation as shown in Figure 4.



Where is IBM WebSphere Studio Application Developer installed?

C:\Program Files\eclipse

Figure 4

After installation is complete, a new subdirectory will be created in the Eclipse plugins directory:

```
    [Eclipse]\plugins\com.openmake.eclipse_1.0.x
```

where *x* is the current revision number of the Meister Plugin.

To verify that the Meister Plugin is recognized, start up Eclipse and go to Help → About…. The Plugin Details button will list all installed Plugins. The Meister Plugin will have an entry:

Catalyst Systems Corporation	Meister Plug-In	1.0.*x*

Using Meister Context Sensitive Menus

The Meister Eclipse plugin is now ready for use. All of the Meister actions are invoked either through Context-Sensitive Menus. There are two broad categories of menus: Project Properties and Meister Build Projects

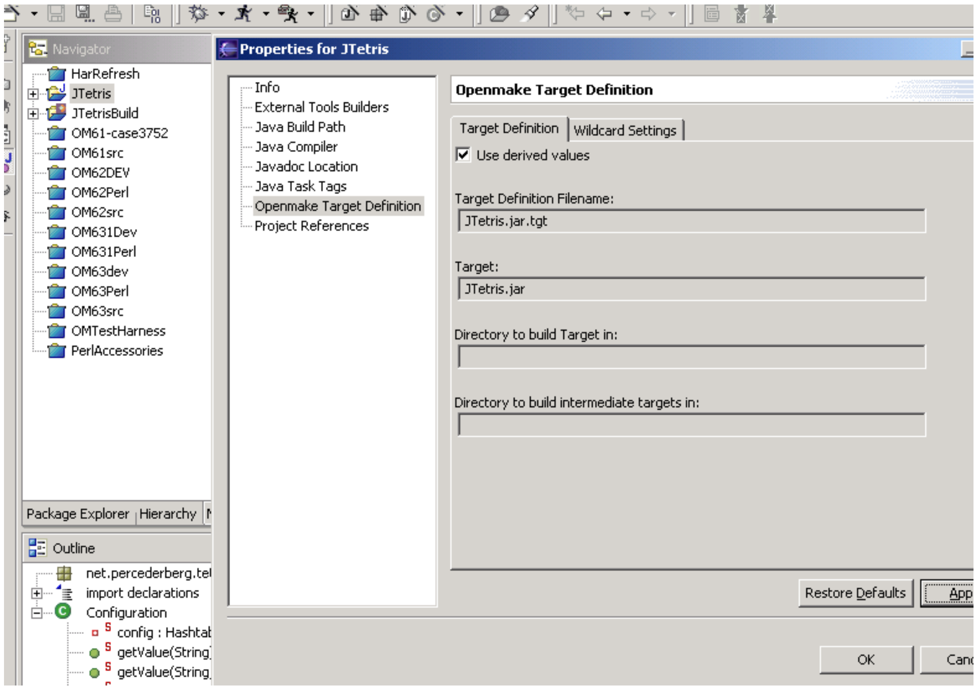
Project Properties

All existing projects, when viewed from the Navigator View, will have an Meister Target Definition option in their “Properties” section. This set of values determines how Eclipse metadata will be converted to an Meister Target Definition file. The default is to use values derived from the Eclipse metadata, although this can be customized from this screen.

The options available on the “Target Definition Tab” are:

- Target Definition Filename: The name of the file that will store the Meister metadata. This file will be saved in the Meister Build Project that is associated with this WSAD project (see the next section).
- Target: The name of the final archive file to create (typically a `.jar`).
- Directory to build Target in: One can specify a sub-directory to in which to place the final archive file (e.g. “`bin`” will create “`bin/<file>.jar`”).
- Directory to build intermediate targets in: This is the directory in which intermediate compiled classes will be placed.

The defaults can be overridden by unchecking "Use derived values" as shown in Figure 5.



Openmake Target Definition

Target Definition | Wildcard Settings

☑ Use derived values

Target Definition Filename: JTetris.jar.tgt

Target: JTetris.jar

Directory to build Target in:

Directory to build intermediate targets in:

Figure 5

The "Wildcard Settings" Tab details how the dependencies are listed within the Meister tgt file. Consider some code:

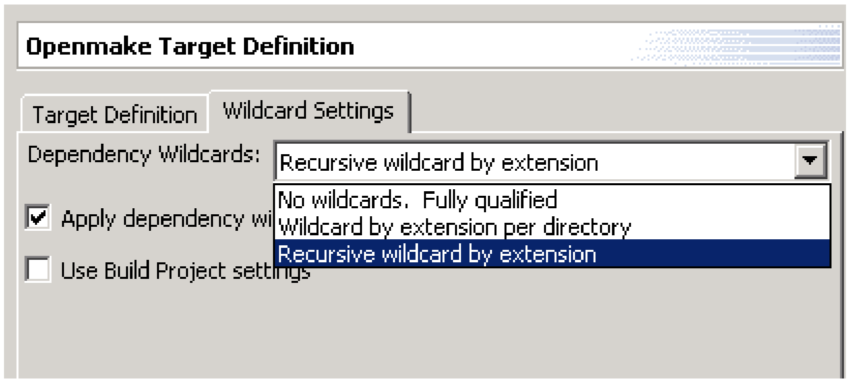
```
com/abc/package1/foo.java
com/abc/package1/bar.java
com/abc/package1/subpackage/sub-foo.java
```

The three wildcard options are (See Figure 6):

1. No Wildcards. Fully qualified:
   Here, the dependencies are listed exactly as shown above.
2. Wildcards by extension per directory:
   Here, there would be two dependencies

   ```
   com/abc/package1/*.java
   com/abc/package1/subpackage/*.java
   ```
3. Recurseive wildcard by extension:
   Here there would be one dependency

```
com/**/*.java
```


Openmake Target Definition

Target Definition | Wildcard Settings

Dependency Wildcards: Recursive wildcard by extension

No wildcards. Fully qualified
Wildcard by extension per directory
Recursive wildcard by extension

[✓] Apply dependency wi

[ ] Use Build Project settings

Figure 6

There are two available checkboxes:

- "Apply dependency wildcard choice to source": This option will apply the wildcard option to the .java source files, as shown in the above example. Unchecking it will apply the wildcarding option only to resource files such as `.html, .jsp,` etc
- "Use Build Project settings": This option will force the TGTs to use the wildcard settings that are defined globally in the Meister Build Project (see below).

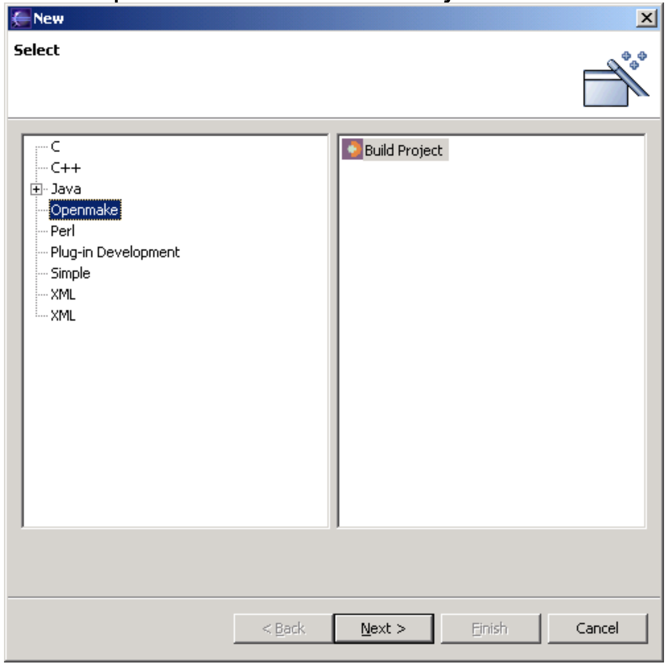
Creating an Meister Build Project

The Meister Plugin uses "Meister Build Projects" as a container to store the Meister Target Definition files and the output of the Meister builds. In this manner, the Meister builds are kept separate from the internal JDT class compiles. An Meister Build Project must be configured and associated with Eclipse Java projects before an Meister build can be invoked.

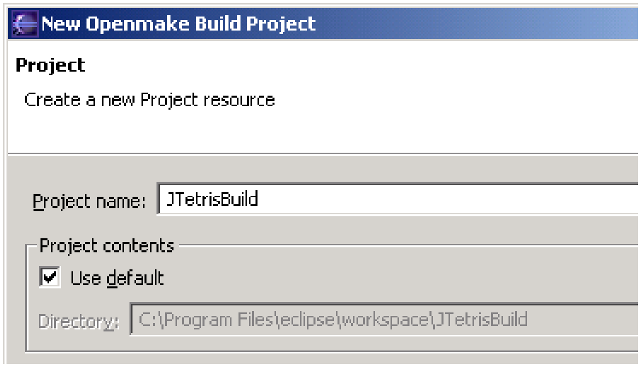
To create a new Meister Build Project within WSAD:

1. Select New → Project …

2. Select Openmake and Build Project. Click "Next"



3. Name the Build Project. Click "Next"



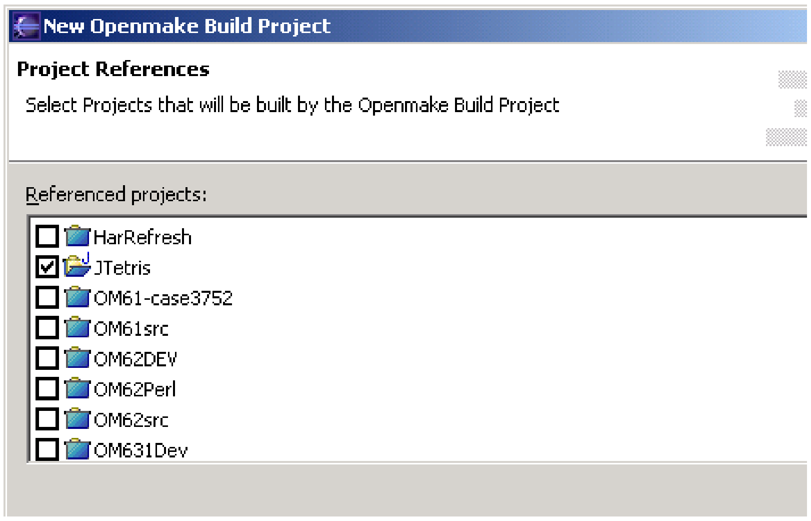
Project name: JTetrisBuild

Project contents

☑ Use default

Directory: C:\Program Files\eclipse\workspace\JTetrisBuild

4. Select the Eclipse Java projects that will be built through this Meister Build Project. Click “Next”.


New Openmake Build Project

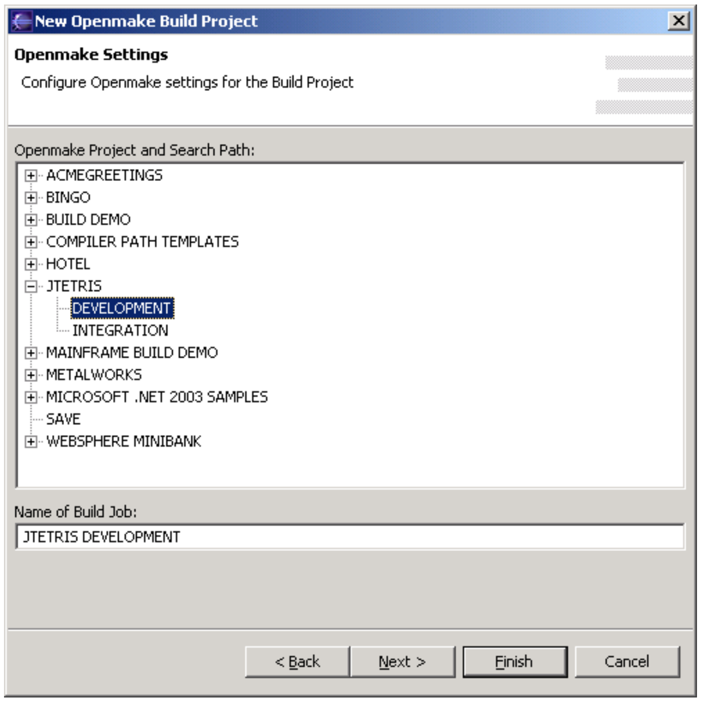
Project References

Select Projects that will be built by the Openmake Build Project

Referenced projects:

- [ ] HarRefresh
- [x] JTetris
- [ ] OM61-case3752
- [ ] OM61src
- [ ] OM62DEV
- [ ] OM62Perl
- [ ] OM62src
- [ ] OM631Dev

5. Select the Meister Project and Dependency Directory (residing on the Meister KB Server) under which this build will occur. Also give a “Build Job Name” to the project.


New Openmake Build Project

Openmake Settings

Configure Openmake settings for the Build Project

Openmake Project and Search Path:

```
+ ACMEGREETINGS
+ BINGO
+ BUILD DEMO
+ COMPILER PATH TEMPLATES
+ HOTEL
- JTETRIS
    DEVELOPMENT
    INTEGRATION
+ MAINFRAME BUILD DEMO
+ METALWORKS
+ MICROSOFT .NET 2003 SAMPLES
  SAVE
+ WEBSPHERE MINIBANK
```

Name of Build Job: JTETRIS DEVELOPMENT

At this point, one can "Finish" defining the Meister Build Project. "Next" will take you to further customizations of:

a. Build Directories
b. bldmake and om flags
c. Pre and post commands
d. Project-wide TGT wildcard settings
e. Nature Mappings

These settings can be later customized by selecting the "Properties" of the Meister Build Project.

At this point, the Meister Build Project will be created. By default, it will have two directories, `tgt/` and `build/` as shown in Figure 7. The "`tgt/`" directory will contain the Target Definition files for all Eclipse Java projects referenced by this Build Project. In this example, we see the TGT file for "`JTetris.jar`". The environment variable `$(TARGET_DEFINITION)` gets set to this location when a build is invoked. The "`build/`" directory will contain the output (`.class` and `.jar`) files from the Meister build.

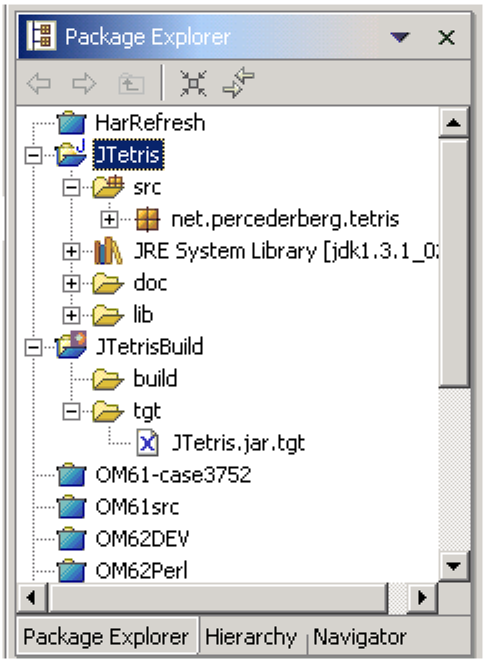


Figure 7

If the developer works on more than one Java application in the same workspace, the developer just needs to configure multiple Meister Build Projects, one for each application. Consider the following example, where the developer is working on the "JTetris" application, and another game called "Space Game" in the same workspace.

Here, the developer would create a new Meister Build Project called "SpaceGameBuild", following the steps outlined above. However, this Build Project would only be associated with the "SpaceGame" application Eclipse Java projects at step 4 of creating a new Build Project:

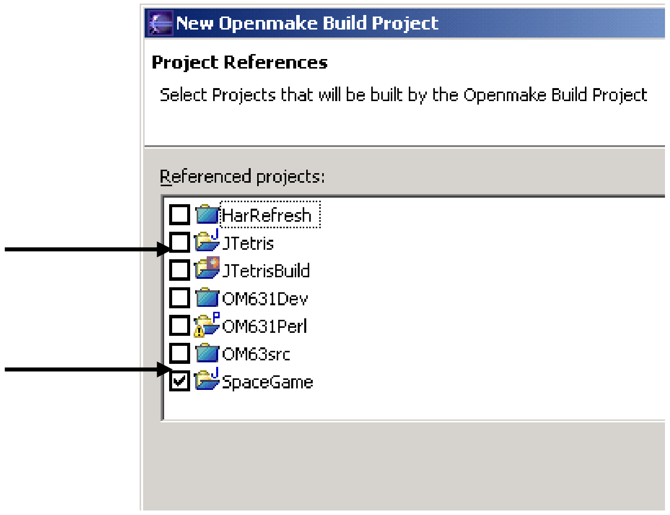


Figure 8

The "SpaceGameBuild" project will now have the TGT file for "SpaceGame.jar". The JTetris build components are isolated in their respective build project.

Executing Meister Tasks Against an Meister Build Project

With the Meister Build Project defined, one uses the context-sensitive menu to invoke Meister actions as shown in Figure 9.

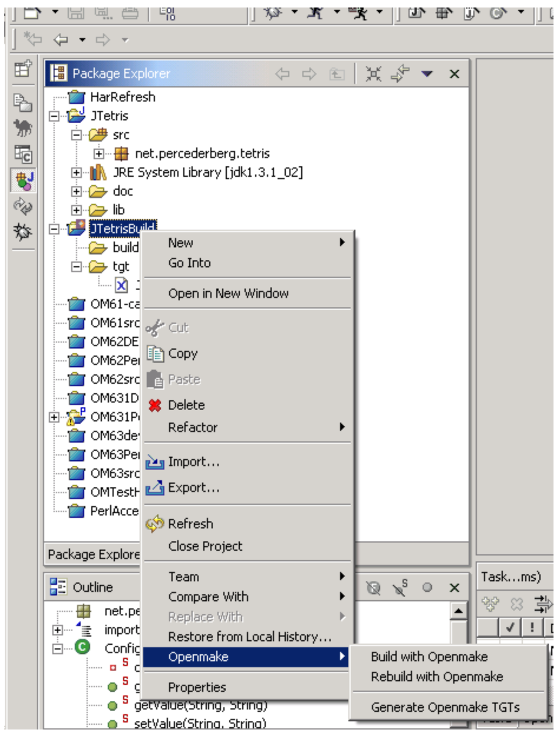
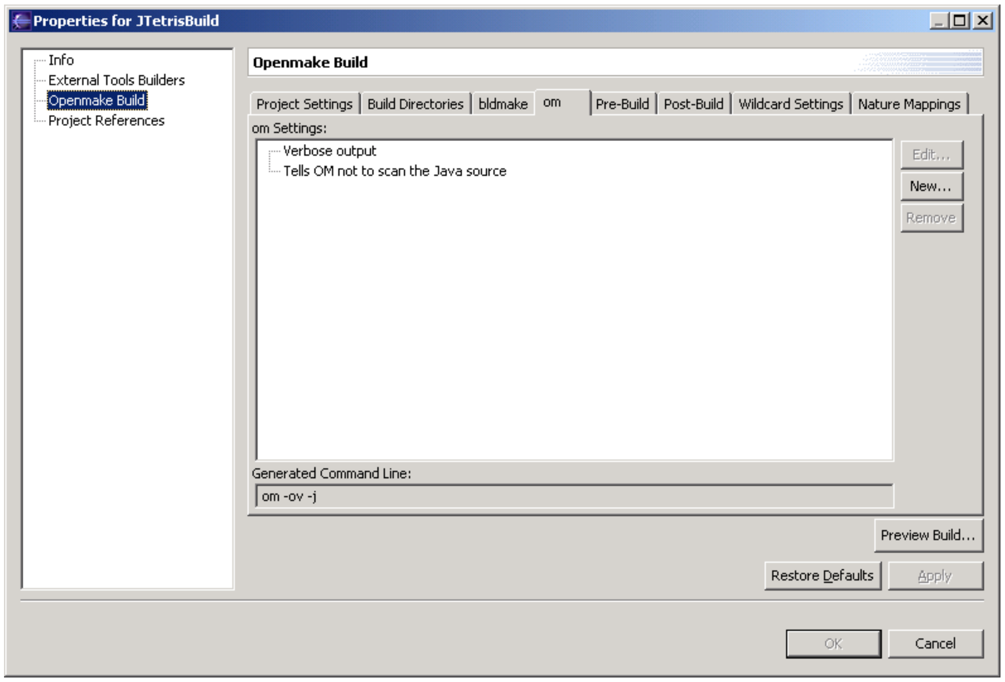


Figure 9

The possible actions are:

1. Generate Meister TGTs: This action will recreate the Target Definition files. By default, Target Definition files are updated:
    a. The Project is configured through the Plugin: Project Properties page
    b. Eclipse's Java Build Path is adjusted
    c. A file is added or removed from the Project
    d. Before a Project Rebuild
2. Build with Meister: This will call `bldmake` and `om` to start the Meister Build process. The build will occur in the "`build/`" directory of the Meister Build Project. `bldmake` and `om` will be called with the options specified in the Build Project. The typical set of options to om will cause om to run in the incremental build mode.
3. Rebuild with Meister: This is the same as "Build with Meister", except that the "clean" option is passed to `om`. This will remove the target files from the build area, and will invoke a full build.

The build options passed to `bldmake` and `om` are configured in the Meister Build Project Properties as shown in Figure 10.


Properties for JTetrisBuild

Openmake Build

om Settings:
- Verbose output
- Tells OM not to scan the Java source

Generated Command Line:

```
om -ov -j
```

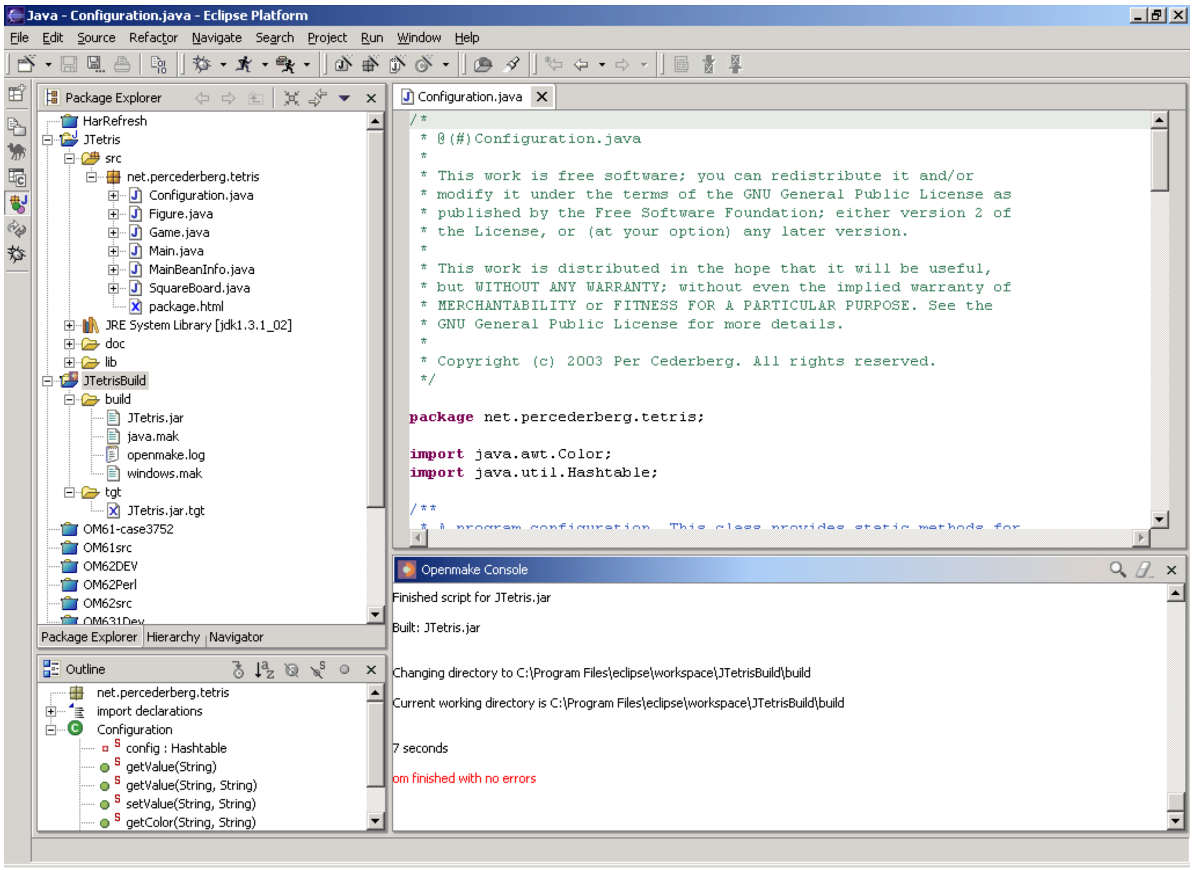
Figure 10

By default, the options passed are:

- bldmake:
    - Case Sensitive: `-s`
- om:
    - Verbose output: `-ov`
    - Tells OM not to scan the Java source: `-j`

The equivalent command line call to `bldmake` or `om` is shown below the options.

When an Meister Build is executed, an external process is called that executes the Meister build commands. This build is independent of the Eclipse IDE (although it is being invoked from within the IDE). The output of the build is shown in the "Meister Console" window (see figure 11).



```
/*
 * @(#)Configuration.java
 *
 * This work is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This work is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * Copyright (c) 2003 Per Cederberg. All rights reserved.
 */

package net.percederberg.tetris;

import java.awt.Color;
import java.util.Hashtable;

/**
 * A program configuration. This class provides static methods for
```

```
Finished script for JTetris.jar

Built: JTetris.jar

Changing directory to C:\Program Files\eclipse\workspace\JTetrisBuild\build

Current working directory is C:\Program Files\eclipse\workspace\JTetrisBuild\build

7 seconds

om finished with no errors
```

Figure 11

The fact that the build occurs external to the IDE is the key for the Meister build process. This insures that the build that the developers are doing will be repeatable on a machine that does not have the Eclipse IDE. The use of the Meister TGT files will allow a seamless transition between the Development teams and other teams responsible for other parts of the lifecycle. This greatly reduces the amount of “back and forth” and confusion between multiple teams involved in taking an enterprise application to production.

Meister Plugin Preferences

Developers can set global preferences for the Meister Plugin. These preferences will be used when creating new Meister Build Projects. (See Figures 12 and 13)

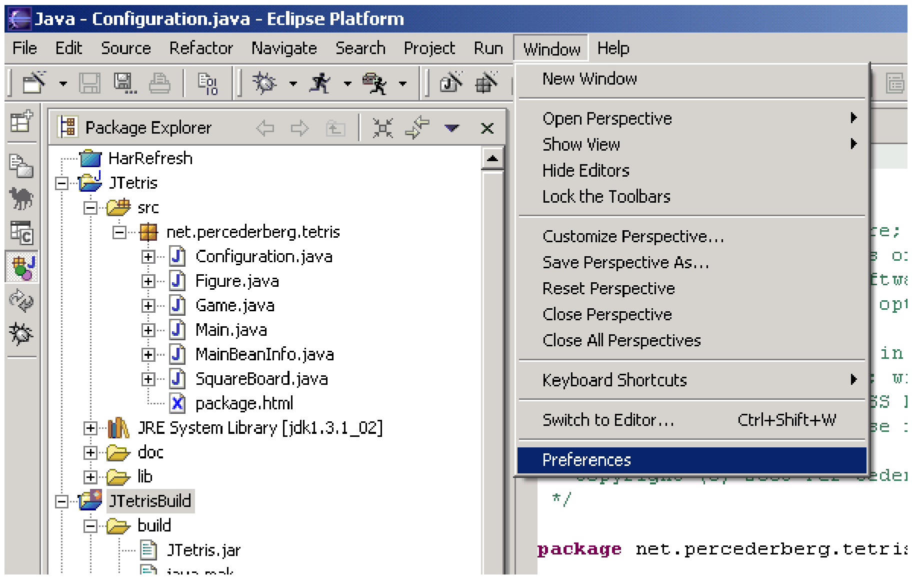
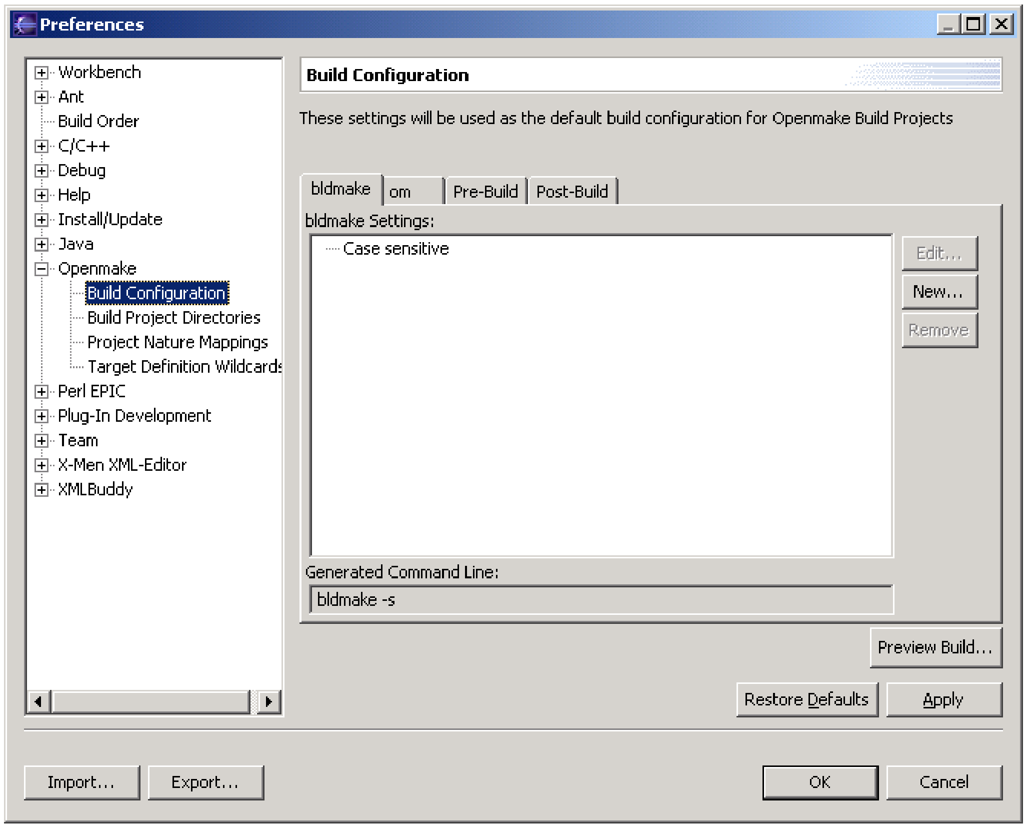


Figure 12

The preferences allow the developer to set the defaults as follows:



Build Configuration

These settings will be used as the default build configuration for Openmake Build Projects

bldmake Settings:

Case sensitive

Generated Command Line:

bldmake -s

Figure 13

Conclusion

This document has demonstrate both the usefulness of the the Meister Build methodology, as well as the configuration of the Meister Eclipse plugin within the Eclipse IDE. It has outlined the benefits of using Meister to build Eclipse based development projects with a focus on Websphere Studio Application Development.

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.