The Eclipse Builder

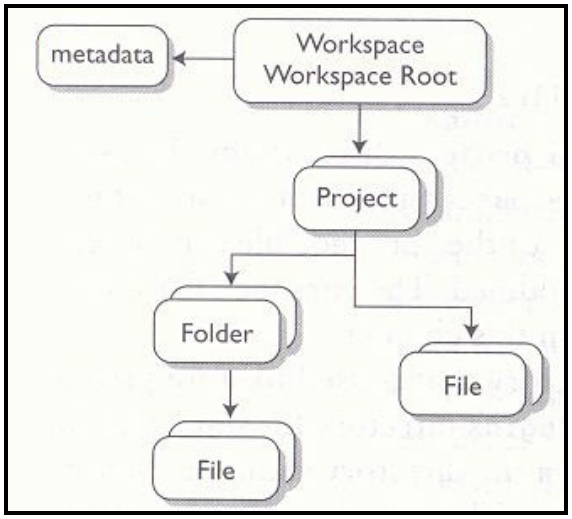
Patty Jablonski
EE564 – Enterprise Java
Mid-term Project Paper
March 26, 2007

Introduction

This paper explores the design of the builder in Eclipse version 3.2.2, including the generic builder's association with resources, the incremental builder infrastructure, and the Java-specific builder and its relationship with the compiler. In addition to background investigation and exploration, this paper also looks at the evolution of the builder over several versions of the Eclipse IDE with each of these designs evaluated. First, to understand the builder, it is necessary to understand what workspaces, resources, and projects are.

Workspace Resource Model

The workspace resource model represents the content that is accessible in the workspace. Since only one workspace is active at one time when using Eclipse, it acts as the root in this model. A workspace (see o.e.core.resources.IWorkspace[*] and o.e.core.internal.resources.Workspace) is where the user works and is represented as a directory in the file system. The workspace contains resources (see o.e.core.resources.IResource and o.e.core.internal.resources.Resource), which are projects (see o.e.core.resources.IProject and o.e.core.internal.resources.Project), files (see o.e.core.resources.IFile and o.e.core.internal.resources.File), and folders (see o.e.core.resources.IFolder and o.e.core.internal.resources.Folder). Together the workspace root (see o.e.core.resources.IWorkspaceRoot and o.e.core.internal.resources.WorkspaceRoot) and its resources form a resource tree. The workspace root, projects, and folders are containers, meaning that they can have children (see o.e.core.resources.IContainer and o.e.core.internal.resources.Container). A file cannot have children. Picture 1 shows a resource model that contains a workspace root with one project that contains both a file and a folder that contains a file.

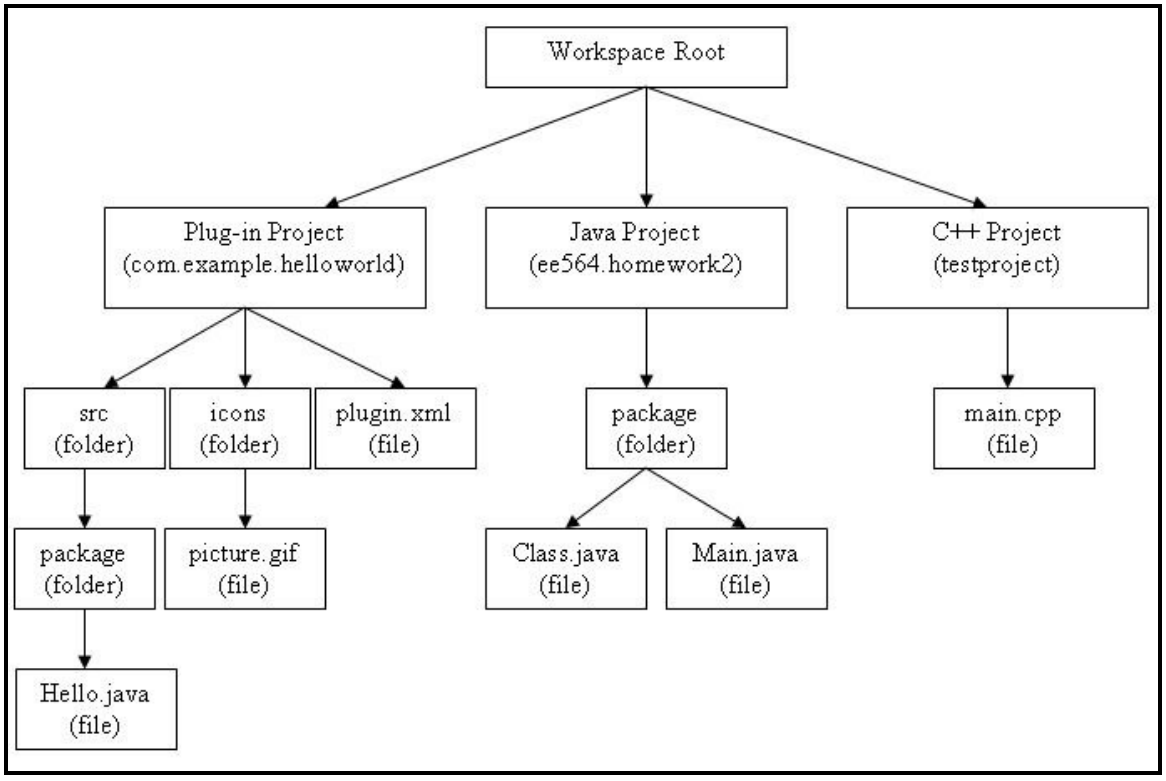


Picture 1: A resource model that contains a workspace root with one project that contains both a file and a folder that contains a file.

[*] Written throughout this paper, the shorthand notation o.e. = org.eclipse.

A workspace root must have projects as its direct children (a project's parent is always the workspace root). The project, which is represented as a directory in the file system, can contain files and folders, and the folders can then contain more files. The entire resource tree models the directories and files on the file system, however the workspace view in Eclipse and the actual file system contents may differ. The view may not show hidden files or unrelated files that are on the file system. To synchronize the Eclipse workspace view and the file system, there is an option to refresh the view if content is added directly to the file system instead of via the IDE.

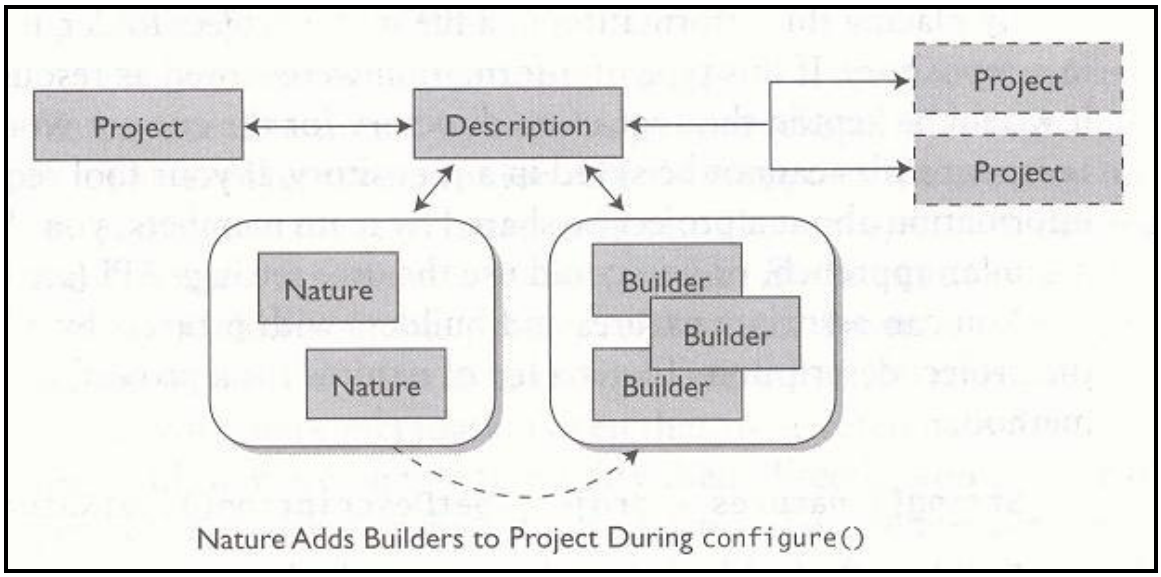
Picture 2 shows an example workspace with multiple projects in it. In this example, each of the projects shown is a different type of project – a plug-in project, a Java project, and a C++ project. This setup is quite possible in the Eclipse IDE, since it can support projects of multiple types and programming languages. The first project, the plug-in project, contains two folders (src and icons) and one file directly within the project (plugin.xml). The icons folder then contains picture.gif, which is a file. The src folder contains a folder (package) which then contains a source code file. The example Java project directly contains a package folder which contains two source code files, Class.java and Main.java. Finally, the C++ project simply contains a single main.cpp file.



Picture 2: An example with multiple projects in the workspace.

Projects – Natures and Builders

The resource of interest when discussing builders is a project. Each project has a project description, which defines the nature(s) and builder(s) of the project and any projects that this project references (as shown in Picture 3).



Picture 3: Each project has a project description, which defines the nature(s) and builder(s) of the project and any projects that this project references.

A nature tells the type a project is (i.e. plug-in, Java, C++, etc) and associates behavior and function with the project. The nature often tells which builder to use with a particular project and can even be used to determine the look-and-feel of the navigator for a project, for example, which icon to display for the project in the view (for example, a J on top of the folder icon for a Java project), and it may also be used to determine the perspectives that can be applied to a project. A single project can have more than one nature.

A builder is a mechanism that allows tool-specific logic to process changed files at specific times and it transforms resources from one form to another. For the Eclipse builder, the tool-specific logic is incremental compilation. Specifically, the Java builder uses the compiler to transform .java source code files to binary .class files for its resource transformation. Also, if there are problems when compiling, the problems are added as problem markers to the affected .java files. These new resources that are created by the builder are called derived resources, which if deleted, can easily be recreated from the source files. Like natures, a single project can have more than one builder. Also, a single nature can be mapped to one or more builders.

A builder that is associated with a nature is specified in the configure method of that nature. First, a nature is added to a project in the project description (specifically with the setNatureIds method listed in o.e.core.resources.IProjectDescription and implemented in

o.e.core.internal.resources.ProjectDescription, as seen in Fragments[*] 1 and 2, respectively) and then the nature's configure method can add the appropriate builders to the project. Similarly, when a nature is removed from a project, the deconfigure method can remove builders that were associated with that project. The configure and deconfigure methods begin and end the life of a specific builder that is associated with a particular project. They are part of the o.e.core.resources.IProjectNature interface, as seen in Fragment 3, and are called in the o.e.core.internal.resources.NatureManager class' configureNature and deconfigureNature methods, in Fragments 4 and 5, respectively. For example, a Java project (o.e.jdt.internal.core.JavaProject) adds the Java builder (as defined in o.e.jdt.core.JavaCore) to its build spec in its implementation of the configure method (and the deconfigure method removes the builder), as seen in Fragments 6 and 7. Also in o.e.core.internal.resources.NatureManager, the mapping between natures and builders is represented as a java.util.Map as seen in Fragment 8 and more specifically as a java.util.HashMap in the findNatureForBuilder method as seen in Fragment 9.

The Project Description

The project description is maintained in a file in the workspace named .project, which is shown for each project in Eclipse's Resource Perspective. The details of the project description can be found in the o.e.core.resources.IProjectDescription interface and the o.e.core.internal.resources.ProjectDescription class. The declaration of the .project file is located in o.e.core.resources.IProjectDescription as seen in Fragment 10. Furthermore, the hasPublicChanges method in o.e.core.internal.resources.ProjectDescription deals with the project's public attributes, which are the attributes that are displayed in the .project file – the project's name, comments, project references, natures, and builders (as defined in the build spec). The hasPublicChanges method, seen in Fragment 11, compares the known project description (passed in as the parameter "description" of type ProjectDescription) with this project's description as stored in the variables "comment", "buildSpec", "staticRefs", and "natures". This method also compares the linked resources, which are references to files or folders that are physically outside the project, that the project may have.

Another way to view a project's project description is by viewing the contents of the .project file. Picture 4 shows the .project file for the Helloworld Plug-in. This is a good example because this is a single project that has multiple natures and builders associated with it. This simple plug-in project has two natures (a plug-in nature and a Java nature) and three builders (a Java builder, a manifest builder, and a schema builder).

The first line of the .project file tells that the file is written in XML version 1.0 with UTF-8 encoding. This is then followed by the contents of the project description itself (defined within the "ProjectDescription" tags). The "name" tag tells the name of this project (com.example.helloworld) and the "comment" tag displays any optional comments about the project (it is empty in the Helloworld Plug-in example). The next tag, "projects", lists the simple project references, in other words, the projects that this project references, if

[*] All code fragments are in the Appendix at the end of this paper due to the amount of space they take up.

any (the Helloworld Plug-in does not have any project references). Next, the "buildSpec" tag represents the build spec, which specifies the set of builders to invoke for this project and the ordering of those builders. The build spec contains build commands (seen in the .project file as "BuildCommand" tags), where each build command tells the name of the builder extension to run ("name" tag) and an optional table of builder arguments ("arguments" tag). Finally, the "natures" tag in the .project file lists the natures of this project, each nature within it specified with a "nature" tag. As mentioned previously, the Helloworld Plug-in's project description consists of three builders – o.e.jdt.core.javabuilder, o.e.pde.ManifestBuilder, and o.e.pde.SchemaBuilder – and two natures – o.e.pde.PluginNature and o.e.jdt.core.javanature.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
    <name>com.example.helloworld</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.pde.ManifestBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.pde.SchemaBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.pde.PluginNature</nature>
        <nature>org.eclipse.jdt.core.javanature</nature>
    </natures>
</projectDescription>
```

Picture 4: The Helloworld Plug-in's .project file.

Each of the project description attributes can be set by the various "set" methods seen in o.e.core.resources.IProjectDescription.

Public Attributes: (displayed in the .project file)

Project Name: setName
Comment: setComment
Simple Project References: setReferencedProjects
Build Spec: setBuildSpec
Natures: setNatureIds

Private Attributes: (not displayed in the .project file)

Project's Location (in the file system): setLocation and setLocationURI
Dynamic Project References: setDynamicReferences

The project description is then set for a particular project via the setDescription method, which is listed in o.e.core.resources.IProject and implemented in o.e.core.internal.resources.Project.

The building of a single project involves invoking each builder in the build spec, one at a time, in the order specified in the build spec. The build spec is represented as an ordered array of ICommand objects, as seen in the setBuildSpec and getBuildSpec methods. The getBuildSpec method's comment in o.e.core.resources.IProjectDescription, shown in Fragment 12, states that "the commands are listed in the order in which they are to be run". In o.e.core.internal.resources.ProjectDescription, the implementation of the setBuildSpec method shows that this array is created in a certain order, seen in Fragment 13. The name of the incremental builder and the optional arguments are defined in the o.e.core.internal.events.BuildCommand class, which implements the interface o.e.core.resources.ICommand. More about the building process and the actual builder invocation will be discussed in the following section.

The Build Process

The first step of the build process is to determine the order of projects to build. The workspace build order, which is an ordered list of projects to build, is computed (or retrieved, if it is set by the user on the preference page, although this manual setting of the project build order is not recommended). If the order of projects to build is not set manually, then the workspace computes a default build order based on the project references, specifically with a breadth-first traversal of the project reference graph as seen in the computeProjectOrder (also listed in o.e.core.resources.IWorkspace) and computeFullProjectOrder methods of o.e.core.internal.resources.Workspace in Fragments 14, 15, and 16, respectively. The workspace build order (the ordered list of projects to build) is returned by o.e.core.internal.resources.Workspace's getBuildOrder method and is based on the workspace's workspace description. The getBuildOrder method that is implemented in o.e.core.internal.resources.Workspace and listed in o.e.core.resources.IWorkspaceDescription is shown in Fragments 17 and 18, respectively. Each opened project in the build order is built, followed by each remaining opened project in the workspace, in no particular order. Closed and non-existent projects are ignored by the build process.

Then, for a single project, the order of builders to invoke must be determined. The order of builders is specified in the build spec, as mentioned in the previous section. The o.e.core.resources.IWorkspace interface lists a build method and it is implemented in o.e.core.internal.resources.Workspace. This build method builds all projects in this workspace. It calls the first build method listed in o.e.core.internal.events.BuildManager, which then calls basicBuildLoop (also in o.e.core.internal.events.BuildManager). The basicBuildLoop method calls the last basicBuild method also listed in this class, which then calls the second basicBuild method in the list, which finally calls the first basicBuild method in the list. Similarly, the o.e.core.resources.IProject interface lists a build method and it is implemented in o.e.core.internal.resources.Project. This build method builds this particular project. The two build methods of o.e.core.internal.resources.Project call the corresponding build methods of o.e.core.internal.events.BuildManager. The second basicBuild method in o.e.core.internal.events.BuildManager, shown in Fragment 19, called with the parameters “project”, “trigger”, “commands”, “status”, and “monitor”, shows that the builders are, in fact, invoked in a specific order. The for loop invokes the builder that is associated with the first build command listed in the build spec, then continues to the next one until it has gone through all build commands in the build spec.

Finally, the builder is invoked only if necessary. For incremental and auto-build, the builder can be skipped if no resources have changed in this project or in any of the projects that the builder is interested in. The o.e.core.internal.events.BuildManager class has methods createBuildersPersistentInfo and getBuildersPersistentInfo, seen in Fragments 20 and 21, respectively, that return a list of BuilderPersistentInfo that includes all builders for the project that have a last built state. The class o.e.core.internal.events.BuilderPersistentInfo contains setInterestingProjects and getInterestingProjects methods, seen in Fragments 22 and 23, respectively, that actually “set” and “get” this array of IProjects that the builder is interested in. Projects that have a last built state are the “interesting projects” to the builder since they have had changes to them before (projects that have never had changes to them are not interesting until they have had changes made to them). The “interesting projects” are those projects returned by the incremental project builder’s build method. Having the builder skip projects that have not had any changes to them is the key concept behind incremental building, which is discussed in the next section.

The Incremental Project Builder and Resource Deltas

There are four kinds of builds: auto-build, clean build, full build, and incremental build. Each of these “build kinds” (also called “triggers”) indicate a particular type of build request and are defined in the class o.e.core.resources.IncrementalProjectBuilder (which is a subclass of o.e.core.internal.events.InternalBuilder) as integer constants AUTO_ BUILD = 9, CLEAN_BUILD = 15, FULL_BUILD = 6, and INCREMENTAL_BUILD = 10, as seen in Fragment 24. The different types of builds are similar in function to the Unix “make” utility, which has options of “make all” (similar to full build) and “make clean” (similar to clean build). Auto-build, full build, and incremental build are all implemented in the build method that is listed in o.e.core.resources.IncrementalProjectBuilder, seen in Fragment 25, and implemented in

its subclasses (such as o.e.jdt.internal.core.builder.JavaBuilder, discussed in a later section). The clean build has its own clean method that is listed in o.e.core.resources.IncrementalProjectBuilder, seen in Fragment 26, and implemented in its subclasses, because the clean build is new to Eclipse 3.0.

Auto-build

An auto-build is not triggered by an explicit build request, but instead it is done automatically when there are resource changes in the workspace. With auto-build turned on, all installed builders will be invoked every time resources are added, removed, or modified in the workspace. The endOperation method in o.e.core.internal.resources.Workspace, seen in Fragment 27, is called in various resources' (such as Projects') build, close, copy, create, delete, move, open, and touch methods. The endOperation method is called at the end of each one of these resource change operations, where it then notifies interested parties that resource changes have happened and notifies all registered resource change listeners (see o.e.core.resources.IResourceChangeListener) via a Notification Manager (see o.e.core.internal.events.NotificationManager). (o.e.core.resources.IWorkspace and o.e.core.internal.resources.Workspace contain addResourceChangeListener methods which add the resource change listener to this workspace). Specifically, the endOperation calls the broadcastPostChange method (also in o.e.core.internal.resources.Workspace), which calls the broadcastChanges method in o.e.core.internal.events.NotificationManager as seen in Fragments 28 and 29, respectively. After a method modifies resources in the workspace, registered listeners receive after-the-fact notification of what happened in the form of a resource change event (defined in o.e.core.resources.IResourceChangeEvent and o.e.core.internal.events.ResourceChangeEvent). If auto-building is enabled, a build is run – the endTopLevel method of o.e.core.internal.events.BuildManager, shown in Fragment 30, is called at the end of the endOperation method. The endTopLevel method then calls the build method of o.e.core.internal.events.AutoBuildJob, which is the job for performing workspace auto-builds and is run whenever the workspace changes regardless of whether auto-build is on or off. The build occurs in a background thread and runs whenever the workspace is not being modified (until the build of the workspace is done). The auto-build thread can be canceled and interrupted by other threads that are modifying the workspace.

Clean build

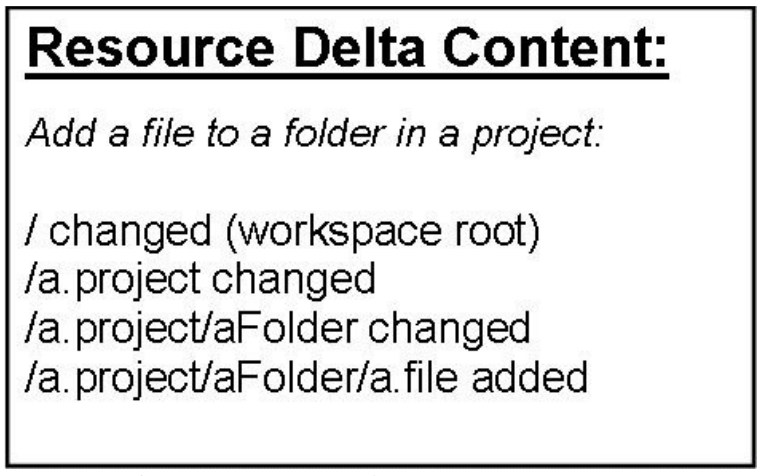
A clean is done by deleting all output files produced by the build and removing and problem markers associated with the builder (similar to a "make clean" in Unix). With all build problems and built state discarded from the clean, a full build (discussed next) will then be done, which will rebuild everything from scratch. A clean build request can be invoked by the user directly from the menu in Eclipse on all projects or selected projects. The clean feature was added to Eclipse since version 3.0 to help the building process to start over from scratch in case a build doesn't complete successfully. An example of cleaning done in an implementation of the clean method (and other methods that it calls) can be seen in o.e.jdt.internal.core.builder.JavaBuilder, shown in Fragment 31.

Full build

A full build is done by discarding all existing build state and starting over from scratch. Unlike a clean build, a full build cannot be invoked by the user directly. A full build is the first build done after a clean build. Since it rebuilds everything from scratch with no existing build state to work with, a full build works with a complete resource tree.

Incremental build

Unlike a full build, an incremental build works with a resource delta tree, which only contains the resources with changes in it. The building in Eclipse is incremental such that the after the first build, subsequent builds should only rebuild based on what has changed since the last build. The reason for this is because it would be inefficient to rebuild from scratch every time the builder is invoked. The changes in the state of a resource tree between two discrete points in time are represented as a resource delta (sometimes called just "delta" for short). For builders, the resource delta is based on a particular project. Picture 5 shows an example of a resource delta's content when the resource change is adding a file (named "a.file") to a folder (named "aFolder") in a project (named "a.project"). The tree of changes shows that / (the workspace root) was changed, /a.project (the project) was changed, /a.project/aFolder (the folder in the project) was changed, and /a.project/aFolder/a.file (the file in the folder in the project) was added. The kinds of resource deltas such as ADDED, REMOVED, and CHANGED are represented as constants in o.e.core.resources.IResourceDelta. The resource delta in the Eclipse source code is represented as an o.e.core.resources.IResourceDelta and o.e.core.internal.events.ResourceDelta. The resource delta tree is represented as type o.e.core.internal.watson.ElementTree with the changes (deltas) as type o.e.core.internal.dtree.DeltaDataTree. The resource delta is processed by a visitor (o.e.core.resources.IResourceDeltaVisitor). The visit process (with the visit method) continues until the complete resource delta tree has been traveled (with the accept method in o.e.core.internal.events.ResourceDelta).



Picture 5: An example of a resource delta's content when the resource change is adding a file (named "a.file") to a folder (named "aFolder") in a project (named "a.project").

Projects have a resource delta that identifies all of the changes that have occurred since the last build and a builder can get this resource delta to guide the incremental building process. (During a full build request, the delta is not available and is null). The incremental project builder can get this project's resource delta with the getDelta method. The getDelta method in o.e.core.resources.IncrementalProjectBuilder (shown in Fragment 32) returns a call to o.e.core.internal.events.InternalBuilder's getDelta method (shown in Fragment 33), which then finally returns a call to the getDelta method in o.e.core.internal.events.BuildManager (shown in Fragment 34).

The Java Model

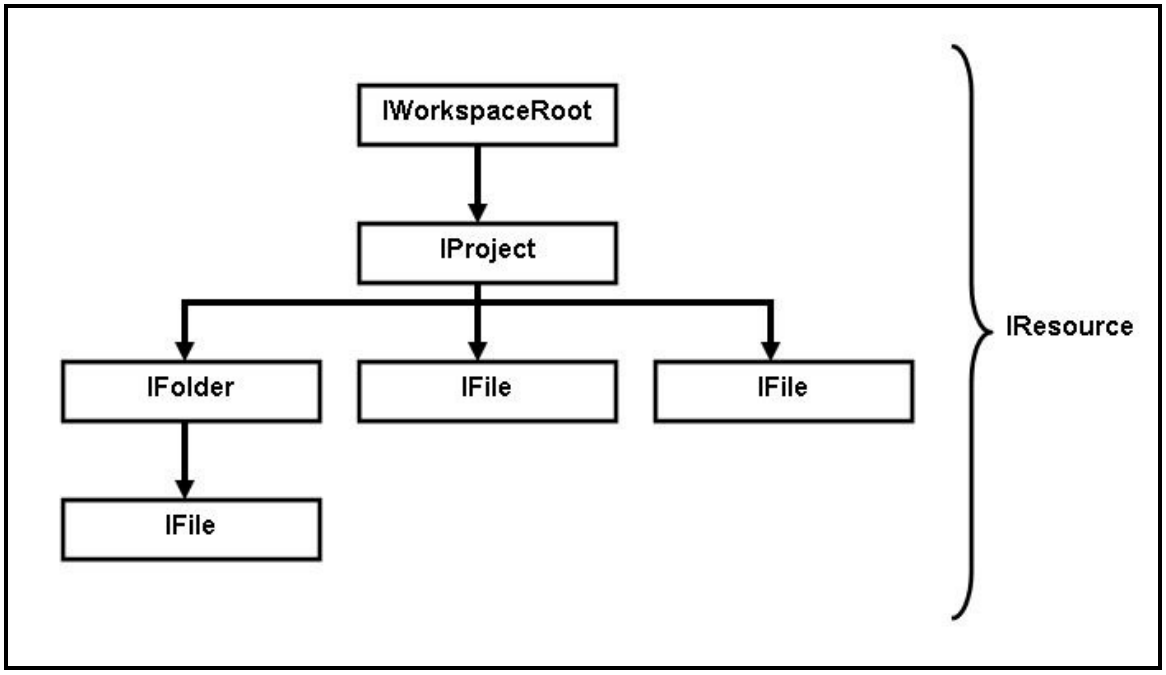
So far, the discussion has focused primarily on the generic builder in Eclipse and its association with resources in the workspace, particularly IProjects and Projects. Most of the classes looked at so far are from the o.e.core.internal packages and their corresponding interfaces are from the o.e.core.resources package. Now the focus will change to provide a brief overview of the Java-specific resources in Eclipse and the Java builder. The Java-related classes are part of the Java Development Tool (JDT), so the classes primarily reside in the o.e.jdt.internal.core packages and their corresponding interfaces in the o.e.jdt.core package.

First, the core support for Java projects is provided by o.e.jdt.core.JavaCore. The Java model, compared to the general workspace resource model that was introduced at the beginning of this paper, contains Java elements (similar to resources in the resource model), which are of type o.e.jdt.core.IJavaElement and o.e.jdt.internal.core.JavaElement. The root Java element corresponding to the workspace is called the Java model element and is of type o.e.jdt.core.IJavaModel (there is a Java Model Manager, o.e.jdt.internal.core.JavaModelManager, that helps manage instances of IJavaModel) and its children are of type o.e.jdt.core.IJavaProject and o.e.jdt.internal.core.JavaProject. The Java projects can have files and folders (folders are of type o.e.jdt.core.IPackageFragmentRoot and o.e.jdt.internal.core.PackageFragmentRoot with children of type o.e.jdt.core.IPackageFragment and o.e.jdt.internal.core.PackageFragment). Files can also be Java source code files of type ICompilationUnit (see o.e.jdt.core.ICompilationUnit and o.e.jdt.internal.core.CompilationUnit) or binary class files of type IClassFile (see o.e.jdt.core.IClassFile and o.e.jdt.internal.core.ClassFile), for example. Picture 6 shows the workspace resource model and Picture 7 shows the Java model, which resembles it.

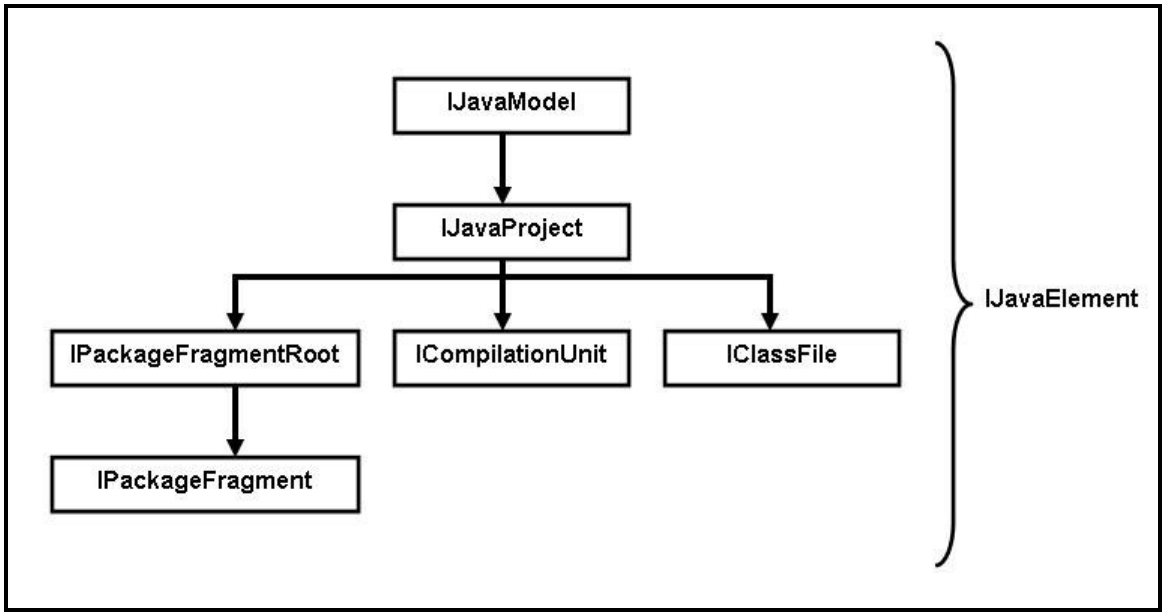
The o.e.jdt.core.JavaCore class has addElementChangedListener methods to add listeners of type o.e.jdt.core.IElementChangedListener to listen to changes to Java elements. The change is described in an o.e.jdt.core.ElementChangedEvent. The changes in Java elements between two discrete points in time are represented as Java element deltas of type o.e.jdt.core.IJavaElementDelta and o.e.jdt.internal.core.JavaElementDelta.

As seen earlier, the configure method in o.e.jdt.internal.core.JavaProject adds the Java builder, which has its name defined in o.e.jdt.core.JavaCore, to its build spec, in Fragments 6 and 7. With a Java nature added to a project, it indicates that the Java

Development Tool (JDT) plug-ins are aware of that project and have configured a classpath (this information is a part of it being a JavaProject) and a Java builder to work on that project.



Picture 6: The workspace resource model contains a workspace root (IWorkspaceRoot) at its root with projects (IProjects) as children, which can contain folders (IFolders) and files (IFiles).



Picture 7: The Java model resembles the workspace resource model. The Java model contains a Java model element (IJavaModel) at its root with Java projects (IJavaProjects) as children, which can contain folders (IPackageFragmentRoots) and files (IPackageFragments or ICompilationUnits and IClassFiles, for example).

The Java Builder and Compiler

The build method of o.e.jdt.internal.core.builder.JavaBuilder explains specifically what the Java builder does for a build, seen in Fragment 35. The o.e.jdt.internal.core.builder.JavaBuilder implements the build (and clean) method, which is an abstract method (unimplemented) in o.e.core.internal.events.InternalBuilder and o.e.core.resources.IncrementalProjectBuilder, which it inherits from. This build method is called by the Build Manager (o.e.core.internal.events.BuildManager).

The Java builder maintains a built state, which includes a list of all classes and interfaces that are referenced by each other in the workspace. This information is returned by the compiler each time a source file is compiled (the state is computed from scratch and updated incrementally). The o.e.jdt.internal.core.builder.JavaBuilder's build method calls the buildAll method, seen in Fragment 36, when it needs to build from scratch and it calls the buildDeltas method, seen in Fragment 37, when it has some built state to retrieve so that it can do an incremental build. BuildAll of o.e.jdt.internal.core.builder.JavaBuilder then calls the build method of o.e.jdt.internal.core.builder.BatchImageBuilder, seen in Fragment 38. Similarly, buildDeltas of o.e.jdt.internal.core.builder.JavaBuilder calls the build method of o.e.jdt.internal.core.builder.IncrementalImageBuilder, seen in Fragment 39. Then, both of these build methods call the compile method of o.e.jdt.internal.core.builder.AbstractImageBuilder, seen in Fragment 40. This compile method calls a second compile method in o.e.jdt.internal.core.builder.AbstractImageBuilder, seen in Fragment 41, which finally calls the compile method in o.e.jdt.internal.compiler.Compiler (the Java compiler), seen in Fragment 42.

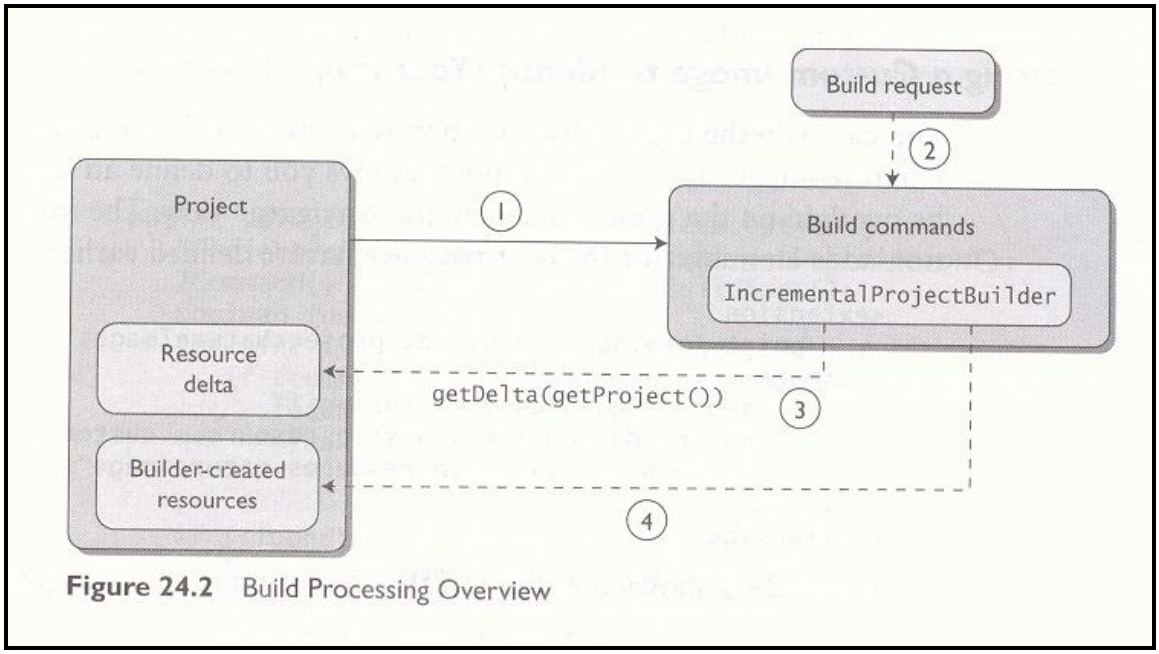
Whenever files are modified, the Java builder receives a resource delta that describes which files were added, removed, or changed. For deleted Java source files, the corresponding class files are deleted. Added and changed source files are added to a queue of files that need to be compiled.

The Java builder then processes this queue of files. It removes a file from the queue and compiles it. It then compares the resulting type to the old class file and sees if the type has structural changes, which are changes that can affect the compilation of a referencing type. Examples of structural changes are added or removed methods, fields, or types, and changed method signatures. If it has structural changes, the Java builder then finds all types that reference the changed type and adds them to the queue. If the type has changed at all, it is written to disk in the builder's output folder. The built state is then updated with the new reference information for the compiled type and this process is repeated until the queue is empty.

Finally, the builder then generates problem markers for each compiled type that had compilation problems.

Despite this long process, the Java builder would only need to process a long queue of files if they had structural changes, which might not occur too often in practice.

Build Summary



Picture 8: A summary of the build process.

This concludes the discussion on the exploration of the builder in Eclipse. To summarize the build process (as shown in Picture 8):

(1) Builders are associated with projects as build commands (recall the project description and .project example), and each project can have any number of build commands.

(2) Each build command is invoked when build processing is triggered. The build request can be invoked directly from the user when a build (see o.e.ui.actions.BuildAction) or clean (see o.e.ui.internal.ide.actions.BuildCleanAction) is selected from the menu in Eclipse. The build request can also be a result of a change in resources, if auto-build is on.

(3) Projects have a resource delta that identifies all of the changes that have occurred since the last build. A builder can get this resource delta to guide the incremental build process by using the getDelta method.

(4) Finally, the builder logic identifies what changed resources must be built and creates additional resources as necessary. For example, the Java builder uses the compiler to transform .java source code files to binary .class files and can also generate problem markers for the source code files if there were problems with them from compilation.

Evolution of the Eclipse Builder and Related Classes

Eclipse has gone through several major and minor revisions. During the exploration phase I noticed a few "@since" Javadoc comments that label when a feature was added (features that were a part of Eclipse from the beginning do not generally have this label). I also looked at the source code of previous versions of Eclipse in the CVS Perspective and downloaded the previous versions from the Eclipse website.

However, to get a more accurate and detailed description of the changes between Eclipse releases, I went through the change logs of Eclipse for every major and minor revision and collected the build notes that are related to builders and some of the other topics discussed in this paper. Table 1 shows a selection of builder and resource-related changes from the Platform/Core Release Notes and Table 2, which follows it, shows a selection of related changes from the JDT/Core Release Notes. For each set of build notes, I only looked at the content directly under the "What's new in this drop" headings, and did not include any of the problem/bug reports and fixes listed there, since I was only interested in the new and changed features in each release of Eclipse (although some new and changed features in Eclipse are a result of the problem/bug reports).

The date and version number of the releases of Eclipse are:

Date:	Version:
11/07/01	1.0
06/27/02	2.0
08/29/02	2.0.1
11/07/02	2.0.2
03/27/03	2.1
06/27/03	2.1.1
11/03/03	2.1.2
03/10/04	2.1.3
06/25/04	3.0
09/16/04	3.0.1
03/11/05	3.0.2
06/27/05	3.1
09/29/05	3.1.1
01/18/06	3.1.2
06/29/06	3.2
09/21/06	3.2.1
02/12/07	3.2.2

Comparing the first releases of Eclipse to its latest releases, many of the classes and interfaces that were discussed in this paper were included in Eclipse from the beginning. This makes sense because many of these classes and interfaces are part of the Eclipse core and define the basic concepts of workspaces, resources, projects, natures, and builders. Many of the new classes were added, and new methods were added to (and old ones were removed from) existing builder and resource-related classes, primarily for

improvement reasons (i.e. the basic functionality was there from the beginning, but the main focus has been on improving it in later releases). For example, the option for a clean build was added to the existing IncrementalProjectBuilder class (and its related classes) since version 3.0 to improve the building process by providing a recovery mechanism from an unsuccessful build via deleting the builder-generated files and starting again from scratch. Also, on 09/23/03, according to the Platform/Core Release Notes, making the auto-build as a thread running in the background would help all workspace changing APIs become more responsive. The JDT/Core Release Notes confirm (on 09/22/03) that as a result of this change to auto-build, the JavaModel will not broadcast deltas during PRE_AUTO_BUILD event notification anymore, which will improve its overall performance. In an earlier release of Eclipse, on 02/05/03, new logic was added to deal with building projects with cyclic dependencies by setting a limit on the number of build iterations. Furthermore, since the beginning (12/04/01), work was being done on improving the existing delta processing, merging deltas together when possible (12/11/01), and batching operations so that only one Java element changed event is reported at the end of the batch (09/24/02). On 11/12/02, the Java builder was improved to only consider resources in the resource tree and not all other files on the file system. More recently, for the Eclipse drop on 12/15/03, it was noted that that the performance of the incremental builder was improved even further.

The biggest design changes between each major Eclipse releases (1.0, 2.0, and 3.0) are:

Between version 1.0 and 2.0:

- By the release of version 2.0, the original Java builder implementation was removed (02/12/02) and the new incremental builder implementation became enabled by default (12/04/01).

Between version 2.0 and 3.0:

- By version 3.0, auto-build was changed to be run in a background thread (09/22/03 and 09/23/03).

Table 1: Select builder and resource-related changes from the Platform/Core Release Notes.

Date	Change
11/01/01	▪ API IncrementalProjectBuilder#build should mention how to handle cancel. ▪ Added a line in the javadoc recommending how cancelation should be handled.
03/12/02	▪ Consolidated many IResource and IWorkspace APIs (delete, copy, move, etc) to use integers specifying flags, rather than having multiple booleans as parameters. See IResource and IWorkspace for details. ▪ Added more support for natures. See IProjectNatureDescriptor and nature related methods on IWorkspace.
03/18/02	▪ The project description file has been moved from the project metadata

	area to the project content area into a file called ".project". Current limitations: ➢ If #setContents is called on the .project file the changes are reflected in the project description in memory during the next resource change notification. ➢ If new content is discovered for the .project file from a #refreshLocal the changes are reflected in the project description in memory during the next resource change notification.
11/18/02	▪ Build Order Computation - The algorithm to calculate the default project build order in the workspace has changed. The new API for this method is: IWorkspace.computeProjectOrder. The old method (IWorkspace.computePrerequisiteOrder) has been deprecated.
01/13/03	▪ New API for more efficient visiting of resource trees. This new visitor mechanism allows clients to traverse resource trees very quickly, by avoiding the creation of unnecessary objects during the traversal. See the new types org.eclipse.core.resources.IResourceProxyVisitor and org.eclipse.core.resources.IResourceProxy, and the corresponding new accept methods on org.eclipse.core.resources.IResource.
02/05/03	▪ Added IncrementalProjectBuilder.needRebuild() and IncrementalProjectBuilder.hasBeenBuilt(IProject) to help accommodate recent changes with building projects with cyclic dependencies. ▪ Added API to IWorkspaceDescription for setting/getting the values to use for the number of iterations for building projects with cyclic dependencies.
09/23/03	▪ Autobuild in the background!! When automatic build is enabled, it now runs in a background thread. This makes all workspace changing APIs more responsive.
05/20/04	▪ Users are now able to build working sets rather than having to build the entire workspace.

Table 2: Select builder and resource-related changes from the JDT/Core Release Notes.

Date	**Change**
12/04/01	▪ New incremental builder implementation enabled by default (can re-enable the old implementation by changing the builder extension in the plugin.xml). ▪ Delta processing improvement: ➢ No longer creates unnecessary Java elements when traversing the resource delta. ➢ Handles changes in binary folder libraries. ➢ Projects that share libraries are notified individually. ➢ Doesn't notify empty deltas any longer.
12/11/01	▪ Java element deltas are batched. If the java model operation modifies a

	resource, then the java element deltas are merged and fired during the resource delta processing. If the java model operation doesn't modify any resource (e.g. IWorkingCopy.reconcile()), then the java element delta is fired right away.
02/12/02	▪ Old Java builder implementation got removed.
04/09/02	▪ Adding a new empty source folder no longer causes a full build. Only an incremental build is needed now.
04/16/02	▪ ElementChangedEvent got added notion of type (similar to IResourceChangeEvent), so as to better allow clients to react to JavaModel changes. ▪ Also added a corresponding API on JavaCore so as to allow registering a listener for a given type of event.
04/23/02	▪ JavaModelOperations now guarantee the JavaModel is up to date when notifying the Java model change listeners. In particular, a builder running after the Java builder will be able to query the Java model with respect to the changes introduced through Java model operations (except for index queries). This was never guaranteed in 1.0, but indirectly occurred due to the fact that the previous Java builder implementation did force to refresh the Java model while building.
05/07/02	▪ JavaBuilder no longer builds projects for which prerequisite projects aborted the build process. This considerably reduces the number of secondary errors when dealing with workspace setup problems. ▪ JavaCore option added, to allow build to abort in presence of invalid classpath (default is ignore).
05/08/02	▪ Java builder is logging its internal errors.
05/15/02	▪ By default, the Java builder is now aborting build process on projects with classpath problems. This option can be disabled through the Java preferences: Window>Preferences>Java>Builder>
09/24/02	▪ Added JavaCore.run(IWorkspaceRunnable, IProgressMonitor) that allows batching of java model operations. Only one Java element changed event is reported at the end of the batch.
11/12/02	▪ The Java builder now iterates over the resource tree, allowing to take advantage of forthcoming workspace structure enhancements (in particular: linked folders). As a consequence, the Java builder will only consider the resources officially reflected in the resource tree (as opposed to existing underlying files not yet reflected when the resource tree is out of sync). Note that the build state format has changed to reflect this evolution, as a consequence, if reusing an existing workspace, the first build action will have to be a rebuild-all projects, since incrementally it will not be able to re-read old build states associated with prerequisite projects (and an incremental build cannot tell the build manager a full rebuild is necessary). ▪ An option allows to control whether the Java builder should clean the output folder(s). Since options can be specified on a per project basis, each individual project can be toggled for cleaning the output folder or not (default is to clean). Also, “scrubbing” output folder got renamed

	into “cleaning” output folder.
09/22/03	▪ As a consequence of migrating to background autobuild, the JavaModel will no longer broadcast deltas during PRE_AUTO_BUILD event notification. These were somewhat inconsistent in so far as the model wasn't totally up to date anyway. Now the model will only fire deltas during POST_CHANGE, or working copy reconcile operations.
12/15/03	▪ Improved incremental builder performance.
08/25/04	▪ All resource change listeners/builder now react to new encoding change notification.
10/19/04	▪ Changed build state format to record access restrictions. As a consequence, a full rebuild will be required when reusing existing workspaces.
03/22/05	▪ The internal build state format has changed and a full build is expected when restarting an existing workspace with this version of JDT Core.
06/09/05	▪ The build state version number has changed. A full build of all projects in the workspace will be triggered upon startup if autobuild is on, or on the next build if autobuild is off.
02/07/06	▪ Java projects can now depend on other Java projects that have replaced the default builder with their own builder, such as an Ant builder. We will now trust that the Ant build was successful and propagate any changes to the affected class files. ▪ Note: When projects are associated with the Java builder, it is able to track structural changes to classfiles (signatures etc...) and only recompile dependents of structurally changed classfiles. In the absence of a Java builder on a prereq project, all modified classfiles will be considered as (potentially) structurally changed; and thus recompilation will be less optimal.
02/21/06	▪ Build states for very large projects should now save in a fraction of the time.
03/29/06	▪ Added new option to JavaCore. Indicate whether the JavaBuilder should check for any changes to .class files in the output folders while performing incremental build operations.

Design Evaluation

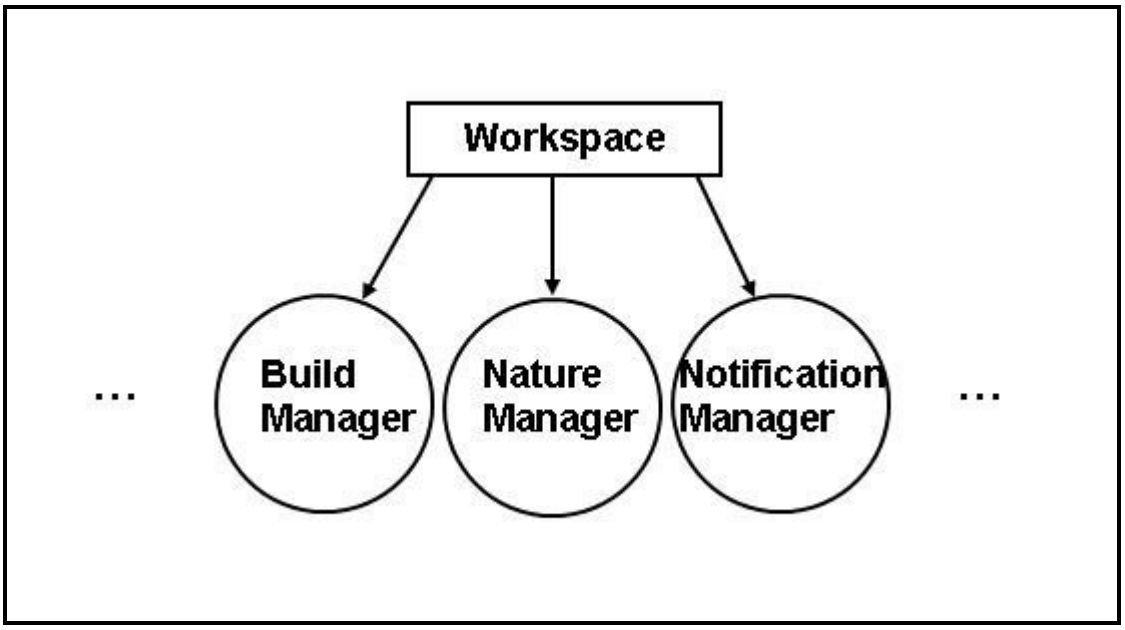
Design Patterns

There are various examples of design patterns in the Eclipse source code, specifically in the builder and resource-related classes. I will discuss a few of these design patterns next.

Singleton Pattern

The workspace contains various types of managers that handle tasks in their specialized areas. The creation of managers most likely is so that the workspace isn’t too large and so that it doesn’t have to do all of these tasks by itself. Instead, the workspace delegates

some tasks to the managers, which continue to work closely with the workspace, seen in Picture 9. Each manager is a singleton, meaning that each has only one instance. This makes sense because each manager should have just a single instance in the workspace. The o.e.core.internal.resources.Workspace's startup method, shown in Fragment 43, calls each manager's constructor. Later on, when o.e.core.internal.events.InternalBuilder wants a BuildManager, it does not call the BuildManager's constructor, but instead gets the Workspace's BuildManager, seen in Fragment 44.



Picture 9: The various managers in Eclipse are examples of singletons.

Command Pattern

The o.e.core.internal.events.BuildManager contains an implementation of the o.e.core.runtime.ISafeRunnable interface as an anonymous class in its third basicBuild method, shown in Fragment 45. The ISafeRunnable interface is an example of a command pattern that has a run method that deals with threads running a procedure, where there is no expectation about the behavior of the procedure.

Composite Pattern and Visitor Pattern

As said earlier, the resource delta in the Eclipse source code is represented as an o.e.core.resources.IResourceDelta and o.e.core.internal.events.ResourceDelta. The resource delta tree is represented as type o.e.core.internal.watson.ElementTree with the changes (deltas) as type o.e.core.internal.dtree.DeltaDataTree. The resource delta is processed by a visitor (o.e.core.resources.IResourceDeltaVisitor). The visit process (with the visit method) continues until the complete resource delta tree has been traveled (with the accept method in o.e.core.internal.events.ResourceDelta). This is an example of the composite pattern (the use of trees to store information) with the visitor pattern (the resource delta nodes provide accept methods with an IResourceDeltaVisitor argument and IResourceDeltaVisitor lists the visit method that is used to visit the resource delta).

Exceptions

The most common exception used in the builder and resource-related classes is CoreException (in the package o.e.core.runtime). The o.e.core.runtime.CoreException is a checked exception (its supertype is Exception). This exception is used as we learned in class. Its name follows the "throws" clause in a method header and is caught in the code that calls this method (or this method can also have a "throws" clause, etc., but the exception is eventually handled). For example, createBuildersPersistentInfo in o.e.core.internal.events.BuildManager throws a CoreException, as seen in its header, in Fragment 46. The createBuildersPersistentInfo is called by the collapseTrees method (and other methods) in o.e.core.internal.resources.SaveManager, shown in Fragment 47. However, the collapseTrees method does not catch this exception, but also has the CoreException listed in the "throws" clause of its header. The collapseTrees method is then called by the save method also in o.e.core.internal.resources.SaveManager. The save method has a "throws" clause also, but it has the call to collapseTrees in a try statement and the CoreException is caught in a following catch statement, shown in Fragment 48.

Comments

The developers of Eclipse use Javadoc comments and regular comments, however not all classes and methods are commented, and the ones that are commented do not have the same components. For example, the comment may include an "@since" tag if the class or method was added in later releases, but would not necessarily have this tag if it is an older class or method that has been in Eclipse all along. Also, some methods have tags like "@param" and "@return", while many others do not. I would suggest deciding on a standard set of tags and being consistent with this amount of comment documentation for all classes and methods.

Also, I would suggest that classes are located in packages that they are expected to be in based on where the rest of the similar ones are located. Many of the interfaces that I looked at were in o.e.core.resources and their corresponding class implementations were in o.e.core.internal.resources. So, why is IResourceDelta in o.e.core.resources, but ResourceDelta is in o.e.core.internal.events? According to the location where most of the others are, ResourceDelta should be in o.e.core.internal.resources to be consistent. However, the names of packages may be arbitrary anyway, as seen by the naming of o.e.core.internal.watson.ElementTree's package, explained in Picture 10.

```
Finally, why are ElementTrees in a package called "watson"?
 - "It's ElementTree my dear Watson, ElementTree."
```

Picture 10: The developers explain why they chose the package o.e.core.internal.watson for the ElementTree class.

As a part of this famous quote, the reason for this package name seems to be due to the humor that "ElementTree" sounds like "elementary".

Finally, I would also suggest that the code be cleaned up a bit more. I saw a DoNotUse() method and some other code that was left in for (temporary) testing purposes. There are specific tests that Eclipse goes through and these extra print statements and other temporary code should be removed when the code is officially released.

However, I still think that Eclipse is some of the best-documented and well-written source code that I've seen. The class, interface, and method names were all simple and made the code easier to understand as a whole. The documentation helped explain a class, interface, or method's purpose, usage, or functionality. Of course, the comments did not include the specification that we learned in class – with an abstract function and rep invariant explicitly written out, and a REQUIRES, MODIFIES, and EFFECTS clause to describe the methods – but I still think that the documentation provided was sufficient.

The developers did create some new types of comments standard to Eclipse. They have "XXX" and "FIXME" tags put in places that still need to be fixed. The Eclipse source code also has "$NON-NLS-#$" comments in it, where # is a number that specifies the part of the preceding print statement in quotes that will remain unchanged. For example, a print statement can contain static components concatenated with a variable in the middle. Each of the static parts would be in quotes. If "$NON-NLS-1$" is seen as a comment after this example print statement, then the developers guarantee that the first part of the print statement will remain unchanged. This comment is useful for other developers who rely on the format of the print statement output for their code.

List of Classes, Interfaces, and Methods

Here I list the classes and interfaces with packages and the methods of each that I looked at during the design evaluation phase of this project. (A more detailed list of the classes and interfaces with packages and the methods of each that I looked at during the code exploration phase of this project is included in a future section).

Table 3: List of classes, interfaces, and methods discussed in the design evaluation section of this paper.

Class or Interface with Methods of Interest
o.e.core.internal.dtree.DeltaDataTree
o.e.core.internal.events.BuildManager - basicBuild - createBuildersPersistentInfo
o.e.core.internal.events.InternalBuilder
o.e.core.internal.events.ResourceDelta - accept
o.e.core.internal.resources.SaveManager - collapseTrees - save
o.e.core.internal.resources.Workspace - startup

o.e.core.internal.watson.ElementTree
o.e.core.resources.IResourceDelta - accept
o.e.core.resources.IResourceDeltaVisitor - visit
o.e.core.runtime.CoreException
o.e.core.runtime.ISafeRunnable - run

Lessons Learned

The first step of this project was choosing a topic. At first I wanted to focus on the compiler, but I found out after one week that it is too big of a topic (as a whole) to do, so I then decided to narrow the topic to just the Java builder. I figured that the Java builder is a much smaller part of Eclipse to focus on. However, I soon realized that I had to also know about workspaces, resources, projects, and natures before I could fully understand builders and I had to know about the generic builder in Eclipse before I could understand the Java-specific builder. So, while the project still grew much larger than I had first anticipated, in the end, I feel like I learned the most important concepts about builders and have kept the project contained within this scope. At first I had wanted to make sure that I mentioned every possible builder-related class in Eclipse including any UI (menu) components. I did this by searching for all classes and interfaces that contain the word "build" or "builder" in them. This was not a good approach to this project, since many of these classes and interfaces are irrelevant to understanding the builder and the building process itself. (I include a list of packages that have builder-related classes in them that I saw during this phase of the project, but that were not relevant enough to be included in this paper, in the next section).

At this point, I had to start over again. I found the BuildManager class from the previous search, which seemed to be very important, so it was recommended that I start the code exploration here. However, this still did not work well for me. Instead, I had to learn about the Eclipse builder first by reading about it before looking at the source code. The best source of information about the internals of Eclipse is from the developers themselves, so I bought a book written by Eclipse developers for Eclipse developers. The information from this book was extremely helpful to me and helped me get up to speed with the background material and then determine the correct place to begin – with the workspace. The book's information helped me to then find and isolate the more important parts of the source code – especially to determine the relevant methods. While I had an idea of the important and relevant classes and interfaces, I then had to sort through their methods and determine which ones matter the most. It turns out that many of the classes have methods of the same name (the build method, for example) that link each class together (a build method in one class calls the build method of another class). What didn't work well for me was trying to figure out what the Eclipse builder does by looking at the source code only. Instead, what worked well for me was gathering the background information so that I knew what to look for in the code. Then, it was a great feeling when I actually found the pieces of code that did the functionality that I was looking for.

I was also new to using Eclipse for code exploration purposes. Although I've used Eclipse to create new projects and do programming assignments, I never used it for exploring the contents of existing source code before this project. Now I regularly use the exploration features of Eclipse and they were truly helpful to me during the exploration phase of this project. The features that I used the most were in the Navigate menu: Open Type, Open Type Hierarchy, and Open Call Hierarchy and some of the features available in a right-click on the source code: Open Declaration, References in Workspace, and Declarations in Workspace. The Outline view was also helpful in seeing all of the fields and methods in a class or interface in alphabetical order and allowing me to jump to certain methods of interest. Before I knew enough to use these features, I always wondered where everything originated from (if I started looking at a class or interface that was used as a type in another class or if I started looking at a method implementation that was called by other methods beforehand, this was important to know). These features helped me see how everything fits together and helped me to get a better understanding of the entire system.

Finally, even though it was a little premature, having had the presentation early during the project helped me get the basic information about builders figured out early on. Also, with existing presentation slides and notes, the paper was eventually easier to write than it would have been without. And, of course, having extra time to finish the project was the most helpful of all (due to starting late, having to re-start and learn the basics, etc). The completion of this final paper would not have been possible otherwise.

List of Classes, Interfaces, and Methods

Here I list the classes and interfaces with packages and the methods of each that I looked at during the code exploration phase of this project. I also include a rough estimate of the lines of code (LOC) for these classes and interfaces, which actually includes comments and blank lines, so I label it as number of lines (NOL) to be more accurate. I got this number by right-clicking the vertical ruler to the left of the editor in Eclipse and selecting "Show Line Numbers" and then reading the last line of each file. I also include the page numbers where each of the classes and interfaces is mentioned in this paper.

This list includes all classes, interfaces, and methods that I believe a newcomer should look at, since I believe that they are most relevant to understanding how the Eclipse builder works. I believe that once a newcomer understands the necessary background information (before looking at the source code), he/she should explore the source code starting with the IWorkspace interface and the Workspace class, exactly the way that this paper presented it. The workspace should be explored first since it introduces the important concepts of resources, projects, project descriptions, natures, builders, and managers.

Table 4: List of classes, interfaces, and methods explored in this paper, with number of lines statistics and the page numbers where each class and interface is mentioned.

Class or Interface with Methods of Interest	Number of Lines (NOL)	Page Numbers
java.util.HashMap	1076	5
java.util.Map	451	5
o.e.core.internal.dtree.DeltaDataTree	961	10
o.e.core.internal.events.AutoBuildJob - build	250	9
o.e.core.internal.events.BuildCommand	220	7
o.e.core.internal.events.BuilderPersistentInfo - getInterestingProjects - setInterestingProjects	61	8
o.e.core.internal.events.BuildManager - basicBuild (3 of them) - basicBuildLoop - build (3 of them) - createBuildersPersistentInfo - endTopLevel - getBuildersPersistentInfo - getDelta	938	8,9,11,13
o.e.core.internal.events.InternalBuilder - build - clean - getDelta	180	8,11,13
o.e.core.internal.events.NotificationManager - broadcastChanges	321	9
o.e.core.internal.events.ResourceChangeEvent	103	9
o.e.core.internal.events.ResourceDelta - accept	543	10
o.e.core.internal.resources.Container	299	2
o.e.core.internal.resources.File	475	2
o.e.core.internal.resources.Folder	185	2
o.e.core.internal.resources.NatureManager - configureNature - deconfigureNature - findNatureForBuilder	650	5
o.e.core.internal.resources.Project - build (2 of them) - close - copy - create - delete - move - open	1100	2,7,8,9

- setDescription - touch		
o.e.core.internal.resources.ProjectDescription - hasPublicChanges - setBuildSpec - setNatureIds	379	5,7
o.e.core.internal.resources.Resource	1695	2
o.e.core.internal.resources.Workspace - addResourceChangeListener - broadcastPostChange - build - computeFullProjectOrder - computeProjectOrder - endOperation - getBuildOrder	2048	2,7,8,9
o.e.core.internal.resources.WorkspaceRoot	289	2
o.e.core.internal.watson.ElementTree	729	10
o.e.core.resources.ICommand	131	7
o.e.core.resources.IContainer	456	2
o.e.core.resources.IFile	1086	2
o.e.core.resources.IFolder	429	2
o.e.core.resources.IncrementalProjectBuilder - build - clean - getDelta	337	8,9,11,13
o.e.core.resources.IProject - build (2 of them) - setDescription	778	2,7,8
o.e.core.resources.IProjectDescription - getBuildSpec - setBuildSpec - setComment - setDynamicReferences - setLocation - setLocationURI - setName - setNatureIds - setReferencedProjects	290	4,5,6,7
o.e.core.resources.IProjectNature - configure - deconfigure	82	5
o.e.core.resources.IResource	2361	2
o.e.core.resources.IResourceChangeEvent	246	9
o.e.core.resources.IResourceChangeListener	48	9
o.e.core.resources.IResourceDelta - accept	553	10

o.e.core.resources.IResourceDeltaVisitor - visit	56	10
o.e.core.resources.IWorkspace - addResourceChangeListener - broadcastPostChange - build - computeProjectOrder	1569	2,7,8,9
o.e.core.resources.IWorkspaceDescription - getBuildOrder	222	7
o.e.core.resources.IWorkspaceRoot	234	2
o.e.jdt.core.ElementChangedEvent	127	11
o.e.jdt.core.IClassFile	174	11
o.e.jdt.core.ICompilationUnit	624	11
o.e.jdt.core.IElementChangedListener	30	11
o.e.jdt.core.IJavaElement	368	11
o.e.jdt.core.IJavaElementDelta	385	11
o.e.jdt.core.IJavaModel	259	11
o.e.jdt.core.IJavaProject	1034	11
o.e.jdt.core.IPackageFragment	198	11
o.e.jdt.core.IPackageFragmentRoot	433	11
o.e.jdt.core.JavaCore - addElementChangedListener	4382	5,11
o.e.jdt.internal.compiler.Compiler - compile	703	13
o.e.jdt.internal.core.builder.AbstractImageBuilder - compile (2 of them)	743	13
o.e.jdt.internal.core.builder.BatchImageBuilder - build	289	13
o.e.jdt.internal.core.builder.IncrementalImageBuilder - build	868	13
o.e.jdt.internal.core.builder.JavaBuilder - build - buildAll - buildDeltas - clean	712	9,13
o.e.jdt.internal.core.ClassFile	735	11
o.e.jdt.internal.core.CompilationUnit	1206	11
o.e.jdt.internal.core.JavaElement	806	11
o.e.jdt.internal.core.JavaElementDelta	733	11
o.e.jdt.internal.core.JavaModelManager	4106	11
o.e.jdt.internal.core.JavaProject - configure - deconfigure	3217	5,11
o.e.jdt.internal.core.PackageFragment	487	11
o.e.jdt.internal.core.PackageFragmentRoot	854	11

o.e.ui.actions.BuildAction	302	14
o.e.ui.internal.ide.actions.BuildCleanAction	54	14

And finally, I would like to briefly mention that there are many other builder-related classes and interfaces that I did not focus on in this paper. Here I list the main categories and the packages to explore for more information.

Ant – This includes builder classes for Apache Ant.
o.apache.tools.ant, o.e.ant.internal

APT – This includes builder classes to deal with files with annotations in them; part of the annotation processing tool (APT).
o.e.jdt.apt.core.internal.env

CVS – This includes builder classes related to the CVS version control system.
o.e.team.internal

External Tools – This includes builder support for external tools.
o.e.ui.externaltools.internal

JDT – This includes additional Java-specific builder classes related to the Java Development Tool (JDT).
o.e.jdt.internal.core.builder

JDT UI – This includes Java-specific builder classes related to the JDT user interface (UI).
o.e.jdt.internal.ui, o.e.jdt.ui.actions

JUnit – This includes builder classes for JUnit.
o.e.jdt.internal.junit.buildpath

PDE – This includes builder classes related to the Plug-in Developer Environment (PDE).
o.e.pde.internal

UI – This includes builder classes related to the general user interface (UI).
o.e.ui.internal.ide.actions

Wizards – This includes builder classes related to wizards.
o.e.jdt.internal.ui.wizards.buildpaths

References

J. Arthorne. *Project Builders and Natures*. Eclipse Corner Article. 2003, 2004. http://www.eclipse.org/articles/Article-Builders/builders.html

J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, P. McCarthy. The Java Developer's Guide to ECLIPSE, Second Edition. 2005.

Incremental Project Builders. Eclipse Platform Plug-in Developer Guide. Part of the Eclipse Help documentation. http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/resAdv_builders.htm

Eclipse Source Code, versions 3.2.2, and previous releases. http://www.eclipse.org/

Appendix of Code Fragments

```
/**
 * Sets the list of natures associated with the described project.
 * A project created with this description will have these natures
 * added to it in the given order.
 * <p>
 * Users must call {@link IProject#setDescription(IProjectDescription, int, IProgressMonitor)}
 * before changes made to this description take effect.
 * </p>
 *
 * @param natures the list of natures
 * @see IProject#setDescription(IProjectDescription, int, IProgressMonitor)
 * @see #getNatureIds()
 */
public void setNatureIds(String[] natures);
```

Fragment 1: The setNatureIds method listed in o.e.core.resources.IProjectDescription.

```
/* (non-Javadoc)
 * @see IProjectDescription#setNatureIds(String[])
 */
public void setNatureIds(String[] value) {
    natures = (String[]) value.clone();
}
```

Fragment 2: The setNatureIds method implemented in o.e.core.internal.resources.ProjectDescription.

```
public interface IProjectNature {
    /**
     * Configures this nature for its project. This is called by the workspace
     * when natures are added to the project using <code>IProject.setDescription</code>
     * and should not be called directly by clients.  The nature extension
     * id is added to the list of natures before this method is called,
     * and need not be added here.
     *
     * Exceptions thrown by this method will be propagated back to the caller
     * of <code>IProject.setDescription</code>, but the nature will remain in
     * the project description.
     *
     * @exception CoreException if this method fails.
     */
    public void configure() throws CoreException;

    /**
     * De-configures this nature for its project.  This is called by the workspace
     * when natures are removed from the project using
     * <code>IProject.setDescription</code> and should not be called directly by
     * clients.  The nature extension id is removed from the list of natures before
     * this method is called, and need not be removed here.
     *
     * Exceptions thrown by this method will be propagated back to the caller
     * of <code>IProject.setDescription</code>, but the nature will still be
     * removed from the project description.
     * *
     * @exception CoreException if this method fails.
     */
    public void deconfigure() throws CoreException;
```

Fragment 3: The configure and deconfigure methods listed in o.e.core.resources.IProjectNature.

```
/**
 * Configures the nature with the given ID for the given project.
 */
protected void configureNature(final Project project, final String natureID, final MultiStatus errors) {
    ISafeRunnable code = new ISafeRunnable() {
        public void run() throws Exception {
            IProjectNature nature = createNature(project, natureID);
            nature.configure();
            ProjectInfo info = (ProjectInfo) project.getResourceInfo(false, true);
            info.setNature(natureID, nature);
        }

        public void handleException(Throwable exception) {
            if (exception instanceof CoreException)
                errors.add(((CoreException) exception).getStatus());
            else
                errors.add(new ResourceStatus(IResourceStatus.INTERNAL_ERROR, project.getFullPath(), NLS.
        }
    };
    if (Policy.DEBUG_NATURES) {
        System.out.println("Configuring nature: " + natureID + " on project: " + project.getName()); //$
    }
    SafeRunner.run(code);
}
```

Fragment 4: The configureNature method of o.e.core.internal.resources.NatureManager, which calls the configure method.

```
/**
 * Deconfigures the nature with the given ID for the given project.
 */
protected void deconfigureNature(final Project project, final String natureID, final MultiStatus status) {
    final ProjectInfo info = (ProjectInfo) project.getResourceInfo(false, true);
    IProjectNature existingNature = info.getNature(natureID);
    if (existingNature == null) {
        // if there isn't a nature then create one so we can deconfig it.
        try {
            existingNature = createNature(project, natureID);
        } catch (CoreException e) {
            // have to swallow the exception because it must be possible
            //to remove a nature that no longer exists in the install
            Policy.log(e.getStatus());
            return;
        }
    }
    final IProjectNature nature = existingNature;
    ISafeRunnable code = new ISafeRunnable() {
        public void run() throws Exception {
            nature.deconfigure();
            info.setNature(natureID, null);
        }

        public void handleException(Throwable exception) {
            if (exception instanceof CoreException)
                status.add(((CoreException) exception).getStatus());
            else
                status.add(new ResourceStatus(IResourceStatus.INTERNAL_ERROR, project.getFullPath(), NLS.b
        }
    };
    if (Policy.DEBUG_NATURES) {
        System.out.println("Deconfiguring nature: " + natureID + " on project: " + project.getName()); //$
    }
    SafeRunner.run(code);
}
```

Fragment 5: The deconfigureNature method of o.e.core.internal.resources.NatureManager, which calls the deconfigure method.

```
/**
 * Configure the project with Java nature.
 */
public void configure() throws CoreException {

    // register Java builder
    addToBuildSpec(JavaCore.BUILDER_ID);
}
/**
 * Removes the Java nature from the project.
 */
public void deconfigure() throws CoreException {

    // deregister Java builder
    removeFromBuildSpec(JavaCore.BUILDER_ID);
}
```

Fragment 6: The configure (deconfigure) method in o.e.jdt.internal.core.JavaProject adds (removes) the Java builder to (from) its build spec.

```
/**
 * The identifier for the Java builder
 * (value <code>"org.eclipse.jdt.core.javabuilder"</code>).
 */
public static final String BUILDER_ID = PLUGIN_ID + ".javabuilder" ; //$NON-NLS-1$
```

Fragment 7: The name of the Java builder is defined in o.e.jdt.core.JavaCore.

```
//maps String (builder ID) -> String (nature ID)
protected Map buildersToNatures = null;
```

Fragment 8: The mapping between natures and builders is represented as a java.util.Map.

```
/**
 * Returns the ID of the project nature that claims ownership of the
 * builder with the given ID.  Returns null if no nature owns that builder.
 */
public String findNatureForBuilder(String builderID) {
    if (buildersToNatures == null) {
        buildersToNatures = new HashMap(10);
        IProjectNatureDescriptor[] descs = getNatureDescriptors();
        for (int i = 0; i < descs.length; i++) {
            String natureId = descs[i].getNatureId();
            String[] builders = ((ProjectNatureDescriptor) descs[i]).getBuilderIds();
            for (int j = 0; j < builders.length; j++) {
                //FIXME: how to handle multiple natures specifying same builder
                buildersToNatures.put(builders[j], natureId);
            }
        }
    }
    return (String) buildersToNatures.get(builderID);
}
```

Fragment 9: The mapping between natures and builders is represented more specifically as a java.util.HashMap in the findNatureForBuilder method.

```
/**
 * Constant that denotes the name of the project description file (value
 * <code>".project"</code>).
 * The handle of a project's description file is
 * <code>project.getFile(DESCRIPTION_FILE_NAME)</code>.
 * The project description file is located in the root of the project's content area.
 *
 * @since 2.0
 */
public static final String DESCRIPTION_FILE_NAME = ".project"; //$NON-NLS-1$
```

Fragment 10: The declaration of the .project file is located in o.e.core.resources.IProjectDescription.

```
/**
 * Returns true if any public attributes of the description have changed.
 * Public attributes are those that are stored in the project description
 * file (.project).
 */
public boolean hasPublicChanges(ProjectDescription description) {
    if (!getName().equals(description.getName()))
        return true;
    if (!comment.equals(description.getComment()))
        return true;
    //don't bother optimizing if the order has changed
    if (!Arrays.equals(buildSpec, description.getBuildSpec(false)))
        return true;
    if (!Arrays.equals(staticRefs, description.getReferencedProjects(false)))
        return true;
    if (!Arrays.equals(natures, description.getNatureIds(false)))
        return true;
    HashMap otherLinks = description.getLinks();
    if (linkDescriptions == null)
        return otherLinks != null;
    return !linkDescriptions.equals(otherLinks);
}
```

Fragment 11: The hasPublicChanges method in o.e.core.internal.resources.ProjectDescription deals with the project's public attributes – the project's name, comments, project references, natures, and builders (as defined in the build spec).

```
/**
 * Returns the list of build commands to run when building the described project.
 * The commands are listed in the order in which they are to be run.
 *
 * @return the list of build commands for the described project
 */
public ICommand[] getBuildSpec();
```

Fragment 12: The build spec is represented as an ordered array of ICommand objects, as seen in the getBuildSpec method (see the comment).

```
/* (non-Javadoc)
 * @see IProjectDescription#setBuildSpec(ICommand[])
 */
public void setBuildSpec(ICommand[] value) {
    Assert.isLegal(value != null);
    //perform a deep copy in case clients perform further changes to the command
    ICommand[] result = new ICommand[value.length];
    for (int i = 0; i < result.length; i++) {
        result[i] = (ICommand) ((BuildCommand) value[i]).clone();
        //copy the reference to any builder instance from the old build spec
        //to preserve builder states if possible.
        for (int j = 0; j < buildSpec.length; j++) {
            if (result[i].equals(buildSpec[j])) {
                ((BuildCommand) result[i]).setBuilder(((BuildCommand) buildSpec[j]).getBuilder());
                break;
            }
        }
    }
    buildSpec = result;
}
```

Fragment 13: The setBuildSpec method, implemented in o.e.core.internal.resources.ProjectDescription, shows that the array of build commands is created in a certain order.

```
/**
 * Computes a total ordering of the given projects based on both static and
 * dynamic project references. If an existing and open project P references
 * another existing and open project Q also included in the list, then Q
 * should come before P in the resulting ordering. Closed and non-existent
 * projects are ignored, and will not appear in the result. References to
 * non-existent or closed projects are also ignored, as are any
 * self-references. The total ordering is always consistent with the global
 * total ordering of all open projects in the workspace.
 * <p>
 * When there are choices, the choice is made in a reasonably stable way.
 * For example, given an arbitrary choice between two projects, the one with
 * the lower collating project name is usually selected.
 * </p>
 * <p>
 * When the project reference graph contains cyclic references, it is
 * impossible to honor all of the relationships. In this case, the result
 * ignores as few relationships as possible. For example, if P2 references
 * P1, P4 references P3, and P2 and P3 reference each other, then exactly
 * one of the relationships between P2 and P3 will have to be ignored. The
 * outcome will be either [P1, P2, P3, P4] or [P1, P3, P2, P4]. The result
 * also contains complete details of any cycles present.
 * </p>
 * <p>
 * This method is time-consuming and should not be called unnecessarily.
 * There are a very limited set of changes to a workspace that could affect
 * the outcome: creating, renaming, or deleting a project; opening or
 * closing a project; adding or removing a project reference.
 * </p>
 *
 * @param projects the projects to order
 * @return result describing the project order
 * @since 2.1
 */
public ProjectOrder computeProjectOrder(IProject[] projects);
```

Fragment 14: ComputeProjectOrder listed in o.e.core.resources.IWorkspace.

```
/* (non-Javadoc)
 * @see IWorkspace#computeProjectOrder(IProject[])
 * @since 2.1
 */
public ProjectOrder computeProjectOrder(IProject[] projects) {

    // compute the full project order for all accessible projects
    ProjectOrder fullProjectOrder = computeFullProjectOrder();

    // "fullProjectOrder.projects" contains no inaccessible projects
    // but might contain accessible projects omitted from "projects"
    // optimize common case where "projects" includes everything
    int accessibleCount = 0;
    for (int i = 0; i < projects.length; i++) {
        if (projects[i].isAccessible()) {
            accessibleCount++;
        }
    }
    // no filtering required if the subset accounts for the full list
    if (accessibleCount == fullProjectOrder.projects.length) {
        return fullProjectOrder;
    }

    // otherwise we need to eliminate mention of other projects...
    // ... from "fullProjectOrder.projects"...
    // Set<IProject> keepers
    Set keepers = new HashSet(Arrays.asList(projects));
    // List<IProject> projects
    List reducedProjects = new ArrayList(fullProjectOrder.projects.length);
    for (int i = 0; i < fullProjectOrder.projects.length; i++) {
        IProject project = fullProjectOrder.projects[i];
        if (keepers.contains(project)) {
            // remove projects not in the initial subset
            reducedProjects.add(project);
        }
    }
    IProject[] p1 = new IProject[reducedProjects.size()];
    reducedProjects.toArray(p1);

    // ... and from "fullProjectOrder.knots"
    // List<IProject[]> knots
    List reducedKnots = new ArrayList(fullProjectOrder.knots.length);
    for (int i = 0; i < fullProjectOrder.knots.length; i++) {
        IProject[] knot = fullProjectOrder.knots[i];
        List x = new ArrayList(knot.length);
        for (int j = 0; j < knot.length; j++) {
            IProject project = knot[j];
            if (keepers.contains(project)) {
                x.add(project);
            }
        }
        // keep knots containing 2 or more projects in the specified subset
        if (x.size() > 1) {
            reducedKnots.add(x.toArray(new IProject[x.size()]));
        }
    }
    IProject[][] k1 = new IProject[reducedKnots.size()][];
    // okay to use toArray here because reducedKnots elements are IProject[]
    reducedKnots.toArray(k1);
    return new ProjectOrder(p1, (k1.length > 0), k1);
}
```

Fragment 15: The workspace computes a default build order with a breadth-first traversal of the project reference graph in the computeProjectOrder method of o.e.core.internal.resources.Workspace.

```
/**
 * Computes the global total ordering of all open projects in the
 * workspace based on project references. If an existing and open project P
 * references another existing and open project Q also included in the list,
 * then Q should come before P in the resulting ordering. Closed and non-
 * existent projects are ignored, and will not appear in the result. References
 * to non-existent or closed projects are also ignored, as are any self-
 * references.
 * <p>
 * When there are choices, the choice is made in a reasonably stable way. For
 * example, given an arbitrary choice between two projects, the one with the
 * lower collating project name is usually selected.
 * </p>
 * <p>
 * When the project reference graph contains cyclic references, it is
 * impossible to honor all of the relationships. In this case, the result
 * ignores as few relationships as possible.  For example, if P2 references P1,
 * P4 references P3, and P2 and P3 reference each other, then exactly one of the
 * relationships between P2 and P3 will have to be ignored. The outcome will be
 * either [P1, P2, P3, P4] or [P1, P3, P2, P4]. The result also contains
 * complete details of any cycles present.
 * </p>
 *
 * @return result describing the global project order
 * @since 2.1
 */
private ProjectOrder computeFullProjectOrder() {

    // determine the full set of accessible projects in the workspace
    // order the set in descending alphabetical order of project name
    SortedSet allAccessibleProjects = new TreeSet(new Comparator() {
        public int compare(Object x, Object y) {
            IProject px = (IProject) x;
            IProject py = (IProject) y;
            return py.getName().compareTo(px.getName());
        }
    });
    IProject[] allProjects = getRoot().getProjects();
    // List<IProject[]> edges
    List edges = new ArrayList(allProjects.length);
    for (int i = 0; i < allProjects.length; i++) {
        Project project = (Project) allProjects[i];
        // ignore projects that are not accessible
        if (!project.isAccessible())
            continue;
        ProjectDescription desc = project.internalGetDescription();
        if (desc == null)
            continue;
        //obtain both static and dynamic project references
        IProject[] refs = desc.getAllReferences(false);
        allAccessibleProjects.add(project);
        for (int j = 0; j < refs.length; j++) {
            IProject ref = refs[j];
            // ignore self references and references to projects that are not accessible
            if (ref.isAccessible() && !ref.equals(project))
                edges.add(new IProject[] {project, ref});
        }
    }

    ProjectOrder fullProjectOrder = ComputeProjectOrder.computeProjectOrder(allAccessibleProjects, edges);
    return fullProjectOrder;
}
```

Fragment 16: The workspace computes a default build order with a breadth-first traversal of the project reference graph in the computeFullProjectOrder method of o.e.core.internal.resources.Workspace.

```
/**
 * Returns the order in which open projects in this workspace will be built.
 * <p>
 * The project build order is based on information specified in the workspace
 * description. The projects are built in the order specified by
 * <code>IWorkspaceDescription.getBuildOrder</code>; closed or non-existent
 * projects are ignored and not included in the result. If
 * <code>IWorkspaceDescription.getBuildOrder</code> is non-null, the default
 * build order is used; again, only open projects are included in the result.
 * </p>
 * <p>
 * The returned value is cached in the <code>buildOrder</code> field.
 * </p>
 *
 * @return the list of currently open projects in the workspace in the order in
 * which they would be built by <code>IWorkspace.build</code>.
 * @see IWorkspace#build(int, IProgressMonitor)
 * @see IWorkspaceDescription#getBuildOrder()
 * @since 2.1
 */
public IProject[] getBuildOrder() {
    if (buildOrder != null) {
        // return previously-computed and cached project build order
        return buildOrder;
    }
    // see if a particular build order is specified
    String[] order = description.getBuildOrder(false);
    if (order != null) {
        // convert from project names to project handles
        // and eliminate non-existent and closed projects
        List projectList = new ArrayList(order.length);
        for (int i = 0; i < order.length; i++) {
            IProject project = getRoot().getProject(order[i]);
            if (project.isAccessible()) {
                projectList.add(project);
            }
        }
        buildOrder = new IProject[projectList.size()];
        projectList.toArray(buildOrder);
    } else {
        // use default project build order
        // computed for all accessible projects in workspace
        buildOrder = computeFullProjectOrder().projects;
    }
    return buildOrder;
}
```

Fragment 17: The workspace build order (the ordered list of projects to build) is returned by o.e.core.internal.resources.Workspace's getBuildOrder method.

```
/**
 * Returns the order in which projects in the workspace should be built.
 * The returned value is <code>null</code> if the workspace's default build
 * order is being used.
 *
 * @return the names of projects in the order they will be built,
 *   or <code>null</code> if the default build order should be used
 * @see #setBuildOrder(String[])
 * @see ResourcesPlugin#PREF_BUILD_ORDER
 */
public String[] getBuildOrder();
```

Fragment 18: The getBuildOrder method that is listed in o.e.core.resources.IWorkspaceDescription.

```
protected void basicBuild(IProject project, int trigger, ICommand[] commands, MultiStatus
    monitor = Policy.monitorFor(monitor);
    try {
        String message = NLS.bind(Messages.events_building_1, project.getFullPath());
        monitor.beginTask(message, Math.max(1, commands.length));
        for (int i = 0; i < commands.length; i++) {
            checkCanceled(trigger, monitor);
            BuildCommand command = (BuildCommand) commands[i];
            IProgressMonitor sub = Policy.subMonitorFor(monitor, 1);
            IncrementalProjectBuilder builder = getBuilder(project, command, i, status);
            if (builder != null)
                basicBuild(trigger, builder, command.getArguments(false), status, sub);
        }
    } catch (CoreException e) {
        status.add(e.getStatus());
    } finally {
        monitor.done();
    }
}
```

Fragment 19: The second of three basicBuild methods defined in o.e.core.internal.events.BuildManager, shows that the builders are, in fact, invoked in a specific order.

```
/**
 * Creates and returns an ArrayList of BuilderPersistentInfo.
 * The list includes entries for all builders that are
 * in the builder spec, and that have a last built state, even if they
 * have not been instantiated this session.
 */
public ArrayList createBuildersPersistentInfo(IProject project) throws CoreException {
    /* get the old builders (those not yet instantiated) */
    ArrayList oldInfos = getBuildersPersistentInfo(project);

    ICommand[] commands = ((Project) project).internalGetDescription().getBuildSpec(false);
    if (commands.length == 0)
        return null;

    /* build the new list */
    ArrayList newInfos = new ArrayList(commands.length);
    for (int i = 0; i < commands.length; i++) {
        String builderName = commands[i].getBuilderName();
        BuilderPersistentInfo info = null;
        IncrementalProjectBuilder builder = ((BuildCommand) commands[i]).getBuilder();
        if (builder == null) {
            // if the builder was not instantiated, use the old info if any.
            if (oldInfos != null)
                info = getBuilderInfo(oldInfos, builderName, i);
        } else if (!(builder instanceof MissingBuilder)) {
            ElementTree oldTree = ((InternalBuilder) builder).getLastBuiltTree();
            //don't persist build state for builders that have no last built state
            if (oldTree != null) {
                // if the builder was instantiated, construct a memento with the important info
                info = new BuilderPersistentInfo(project.getName(), builderName, i);
                info.setLastBuildTree(oldTree);
                info.setInterestingProjects(((InternalBuilder) builder).getInterestingProjects());
            }
        }
        if (info != null)
            newInfos.add(info);
    }
    return newInfos;
}
```

Fragment 20: The o.e.core.internal.events.BuildManager class has a method createBuildersPersistentInfo that returns a list of BuilderPersistentInfo that includes all builders for the project that have a last built state.

```
/**
 * Returns a list of BuilderPersistentInfo.
 * The list includes entries for all builders that are in the builder spec,
 * and that have a last built state but have not been instantiated this session.
 */
public ArrayList getBuildersPersistentInfo(IProject project) throws CoreException {
    return (ArrayList) project.getSessionProperty(K_BUILD_LIST);
}
```

Fragment 21: The o.e.core.internal.events.BuildManager class has a method getBuildersPersistentInfo that returns a list of BuilderPersistentInfo that includes all builders for the project that have a last built state.

```
public void setInterestingProjects(IProject[] projects) {
    interestingProjects = projects;
}
```

Fragment 22: The class o.e.core.internal.events.BuilderPersistentInfo contains the setInterestingProjects method that actually "sets" the array of Iprojects that the builder is interested in.

```
public IProject[] getInterestingProjects() {
    return interestingProjects;
}
```

Fragment 23: The class o.e.core.internal.events.BuilderPersistentInfo contains the getInterestingProjects method that actually "gets" the array of Iprojects that the builder is interested in.

```
/*
 * ======================================================================
 * Constants related to build requests
 * ======================================================================
 */
/**
 * Build kind constant (value 10) indicating an incremental build request.
 *
 * @see IProject#build(int, IProgressMonitor)
 * @see IProject#build(int, String, Map, IProgressMonitor)
 * @see IWorkspace#build(int, IProgressMonitor)
 */
public static final int INCREMENTAL_BUILD = 10;
/**
 * Build kind constant (value 6) indicating a full build request.
 *
 * @see IProject#build(int, IProgressMonitor)
 * @see IProject#build(int, String, Map, IProgressMonitor)
 * @see IWorkspace#build(int, IProgressMonitor)
 */
public static final int FULL_BUILD = 6;
/**
 * Build kind constant (value 9) indicating an automatic build request.
 *
 */
public static final int AUTO_BUILD = 9;
/**
 * Build kind constant (value 15) indicating a build clean request
 *
 * @see IProject#build(int, IProgressMonitor)
 * @see IProject#build(int, String, Map, IProgressMonitor)
 * @see IWorkspace#build(int, IProgressMonitor)
 * @since 3.0
 */
public static final int CLEAN_BUILD = 15;
```

Fragment 24: The four kinds of builds listed in o.e.core.resources.IncrementalProjectBuilder.

```
/**
 * Runs this builder in the specified manner. Subclasses should implement
 * this method to do the processing they require.
 * <p>
 * If the build kind is <code>INCREMENTAL_BUILD</code> or
 * <code>AUTO_BUILD</code>, the <code>getDelta</code> method can be
 * used during the invocation of this method to obtain information about
 * what changes have occurred since the last invocation of this method. Any
 * resource delta acquired is valid only for the duration of the invocation
 * of this method.
 * </p>
 * <p>
 * After completing a build, this builder may return a list of projects for
 * which it requires a resource delta the next time it is run. This
 * builder's project is implicitly included and need not be specified. The
 * build mechanism will attempt to maintain and compute deltas relative to
 * the identified projects when asked the next time this builder is run.
 * Builders must re-specify the list of interesting projects every time they
 * are run as this is not carried forward beyond the next build. Projects
 * mentioned in return value but which do not exist will be ignored and no
 * delta will be made available for them.
 * </p>
 * <p>
 * This method is long-running; progress and cancellation are provided by
 * the given progress monitor. All builders should report their progress and
 * honor cancel requests in a timely manner. Cancelation requests should be
 * propagated to the caller by throwing
 * <code>OperationCanceledException</code>.
 * </p>
 * <p>
 * All builders should try to be robust in the face of trouble. In
 * situations where failing the build by throwing <code>CoreException</code>
 * is the only option, a builder has a choice of how best to communicate the
 * problem back to the caller. One option is to use the
 * <code>BUILD_FAILED</code> status code along with a suitable message;
 * another is to use a multi-status containing finer-grained problem
 * diagnoses.
 * </p>
 *
 * @param kind the kind of build being requested. Valid values are
 * <ul>
 * <li><code>FULL_BUILD</code>- indicates a full build.</li>
 * <li><code>INCREMENTAL_BUILD</code>- indicates an incremental build.
 * </li>
 * <li><code>AUTO_BUILD</code>- indicates an automatically triggered
 * incremental build (autobuilding on).</li>
 * </ul>
 * @param args a table of builder-specific arguments keyed by argument name
 * (key type: <code>String</code>, value type: <code>String</code>);
 * <code>null</code> is equivalent to an empty map
 * @param monitor a progress monitor, or <code>null</code> if progress
 * reporting and cancellation are not desired
 * @return the list of projects for which this builder would like deltas the
 * next time it is run or <code>null</code> if none
 * @exception CoreException if this build fails.
 * @see IProject#build(int, String, Map, IProgressMonitor)
 */
protected abstract IProject[] build(int kind, Map args, IProgressMonitor monitor) throws CoreExc
```

Fragment 25: Auto-build, full build, and incremental build are all implemented in the build method that is listed in o.e.core.resources.IncrementalProjectBuilder and implemented in its subclasses.

```
/**
 * Clean is an opportunity for a builder to discard any additional state that has
 * been computed as a result of previous builds. It is recommended that builders
 * override this method to delete all derived resources created by previous builds,
 * and to remove all markers of type <code>IMarker.PROBLEM</code> that
 * were created by previous invocations of the builder. The platform will
 * take care of discarding the builder's last built state (there is no need
 * to call <code>forgetLastBuiltState</code>).
 * </p>
 * <p>
 * This method is called as a result of invocations of
 * <code>IWorkspace.build</code> or <code>IProject.build</code> where
 * the build kind is <code>CLEAN_BUILD</code>.
 * <p>
 * This default implementation does nothing. Subclasses may override.
 * <p>
 * This method is long-running; progress and cancellation are provided by
 * the given progress monitor. All builders should report their progress and
 * honor cancel requests in a timely manner. Cancelation requests should be
 * propagated to the caller by throwing
 * <code>OperationCanceledException</code>.
 * </p>
 *
 * @param monitor a progress monitor, or <code>null</code> if progress
 * reporting and cancellation are not desired
 * @exception CoreException if this build fails.
 * @see IWorkspace#build(int, IProgressMonitor)
 * @see #CLEAN_BUILD
 * @since 3.0
 */
protected void clean(IProgressMonitor monitor) throws CoreException {
    //default implementation does nothing
    if (false)
        throw new CoreException(Status.OK_STATUS);//thwart compiler warning
}
```

Fragment 26: The clean build has its own clean method that is listed in o.e.core.resources.IncrementalProjectBuilder and implemented in its subclasses, because the clean build is new to Eclipse 3.0.

```
/**
 * End an operation (group of resource changes).
 * Notify interested parties that resource changes have taken place.  All
 * registered resource change listeners are notified.  If autobuilding is
 * enabled, a build is run.
 */
public void endOperation(ISchedulingRule rule, boolean build, IProgressMonitor monitor) throws Core
    WorkManager workManager = getWorkManager();
    //don't do any end operation work if we failed to check in
    if (workManager.checkInFailed(rule))
        return;
    // This is done in a try finally to ensure that we always decrement the operation count
    // and release the workspace lock.  This must be done at the end because snapshot
    // and "hasChanges" comparison have to happen without interference from other threads.
    boolean hasTreeChanges = false;
    boolean depthOne = false;
    try {
        workManager.setBuild(build);
        // if we are not exiting a top level operation then just decrement the count and return
        depthOne = workManager.getPreparedOperationDepth() == 1;
        if (!(notificationManager.shouldNotify() || depthOne)) {
            notificationManager.requestNotify();
            return;
        }
        // do the following in a try/finally to ensure that the operation tree is nulled at the end
        // as we are completing a top level operation.
        try {
            notificationManager.beginNotify();
            // check for a programming error on using beginOperation/endOperation
            Assert.isTrue(workManager.getPreparedOperationDepth() > 0, "Mismatched begin/endOperati

            // At this time we need to re-balance the nested operations. It is necessary because
            // build() and snapshot() should not fail if they are called.
            workManager.rebalanceNestedOperations();

            //find out if any operation has potentially modified the tree
            hasTreeChanges = workManager.shouldBuild();
            //double check if the tree has actually changed
            if (hasTreeChanges)
                hasTreeChanges = operationTree != null && ElementTree.hasChanges(tree, operationTre
            broadcastPostChange();
            // Request a snapshot if we are sufficiently out of date.
            saveManager.snapshotIfNeeded(hasTreeChanges);
        } finally {
            // make sure the tree is immutable if we are ending a top-level operation.
            if (depthOne) {
                tree.immutable();
                operationTree = null;
            } else
                newWorkingTree();
        }
    } finally {
        workManager.checkOut(rule);
    }
    if (depthOne)
        buildManager.endTopLevel(hasTreeChanges);
}
```

Fragment 27: The endOperation method in o.e.core.internal.resources.Workspace is called at the end of one of the various resource change operations like build, close, copy, create, delete, move, open, and touch methods.

```
public void broadcastPostChange() {
    ResourceChangeEvent event = new ResourceChangeEvent(this, IResourceChangeEvent.POST_CHANGE, 0, null);
    notificationManager.broadcastChanges(tree, event, true);
}
```

Fragment 28: The endOperation calls the broadcastPostChange method (also in o.e.core.internal.resources.Workspace).

```
/**
 * The main broadcast point for notification deltas
 */
public void broadcastChanges(ElementTree lastState, ResourceChangeEvent event, boolean lockTree) {
    final int type = event.getType();
    try {
        // Do the notification if there are listeners for events of the given type.
        if (!listeners.hasListenerFor(type))
            return;
        isNotifying = true;
        ResourceDelta delta = getDelta(lastState, type);
        //don't broadcast POST_CHANGE or incremental build events if the delta is empty
        if (delta == null || delta.getKind() == 0) {
            int trigger = event.getBuildKind();
            if (trigger != IncrementalProjectBuilder.FULL_BUILD && trigger != IncrementalProjectBuilder.CLEAN_
                return;
        }
        event.setDelta(delta);
        long start = System.currentTimeMillis();
        notify(getListeners(), event, lockTree);
        lastNotifyDuration = System.currentTimeMillis() - start;
    } finally {
        // Update the state regardless of whether people are listening.
        isNotifying = false;
        cleanUp(lastState, type);
    }
}
```

Fragment 29: The broadcastPostChange method in o.e.core.internal.resources.Workspace calls the broadcastChanges method in o.e.core.internal.events.NotificationManager.

```
/**
 * The outermost workspace operation has finished.  Do an autobuild if necessary.
 */
public void endTopLevel(boolean needsBuild) {
    autoBuildJob.build(needsBuild);
}
```

Fragment 30: The endTopLevel method of o.e.core.internal.events.BuildManager is called at the end of the endOperation method of o.e.core.internal.resources.Workspace to perform an auto-build if it is enabled.

```
protected void clean(IProgressMonitor monitor) throws CoreException {
    this.currentProject = getProject();
    if (currentProject == null || !currentProject.isAccessible()) return;

    if (DEBUG)
        System.out.println("\nCleaning " + currentProject.getName() //$NON-NLS-1$
            + " @ " + new Date(System.currentTimeMillis())); //$NON-NLS-1$
    this.notifier = new BuildNotifier(monitor, currentProject);
    notifier.begin();
    try {
        notifier.checkCancel();

        initializeBuilder(CLEAN_BUILD, true);
        if (DEBUG)
            System.out.println("Clearing last state as part of clean : " + lastState); //$NON-NLS-1$
        clearLastState();
        removeProblemsAndTasksFor(currentProject);
        new BatchImageBuilder(this, false).cleanOutputFolders(false);
    } catch (CoreException e) {
        Util.log(e, "JavaBuilder handling CoreException while cleaning: " + currentProject.getName()
        IMarker marker = currentProject.createMarker(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER);
        marker.setAttribute(IMarker.MESSAGE, Messages.bind(Messages.build_inconsistentProject, e.get
        marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
    } finally {
        notifier.done();
        cleanup();
    }
    if (DEBUG)
        System.out.println("Finished cleaning " + currentProject.getName() //$NON-NLS-1$
            + " @ " + new Date(System.currentTimeMillis())); //$NON-NLS-1$
}

private void cleanup() {
    this.participants = null;
    this.nameEnvironment = null;
    this.binaryLocationsPerProject = null;
    this.lastState = null;
    this.notifier = null;
    this.extraResourceFileFilters = null;
    this.extraResourceFolderFilters = null;
}

private void clearLastState() {
    JavaModelManager.getJavaModelManager().setLastBuiltState(currentProject, null);
}
```

Fragment 31: An example of cleaning done in an implementation of the clean method (and other methods that it calls) can be seen in o.e.jdt.internal.core.builder.JavaBuilder.

```
/**
 * Returns the resource delta recording the changes in the given project
 * since the last time this builder was run. <code>null</code> is returned
 * if no such delta is available. An empty delta is returned if no changes
 * have occurred. If <code>null</code> is returned, clients should assume
 * that unspecified changes have occurred and take the appropriate action.
 * <p>
 * The system reserves the right to trim old state in an effort to conserve
 * space. As such, callers should be prepared to receive <code>null</code>
 * even if they previously requested a delta for a particular project by
 * returning that project from a <code>build</code> call.
 * </p>
 * <p>
 * A non- <code>null</code> delta will only be supplied for the given
 * project if either the result returned from the previous
 * <code>build</code> included the project or the project is the one
 * associated with this builder.
 * </p>
 * <p>
 * If the given project was mentioned in the previous <code>build</code>
 * and subsequently deleted, a non- <code>null</code> delta containing the
 * deletion will be returned. If the given project was mentioned in the
 * previous <code>build</code> and was subsequently created, the returned
 * value will be <code>null</code>.
 * </p>
 * <p>
 * A valid delta will be returned only when this method is called during a
 * build. The delta returned will be valid only for the duration of the
 * enclosing build execution.
 * </p>
 *
 * @return the resource delta for the project or <code>null</code>
 */
public final IResourceDelta getDelta(IProject project) {
    return super.getDelta(project);
}
```

Fragment 32: The getDelta method in o.e.core.resources.IncrementalProjectBuilder returns a call to o.e.core.internal.events.InternalBuilder's getDelta method.

```
/*
 * @see IncrementalProjectBuilder#forgetLastBuiltState
 */
protected IResourceDelta getDelta(IProject aProject) {
    return buildManager.getDelta(aProject);
}
```

Fragment 33: The o.e.core.internal.events.InternalBuilder's getDelta method returns a call to the getDelta method in o.e.core.internal.events.BuildManager.

```
IResourceDelta getDelta(IProject project) {
    try {
        lock.acquire();
        if (currentTree == null) {
            if (Policy.DEBUG_BUILD_FAILURE)
                Policy.debug("Build: no tree for delta " + debugBuilder() + " [" + debugProject()
            return null;
        }
        //check if this builder has indicated it cares about this project
        if (!isInterestingProject(project)) {
            if (Policy.DEBUG_BUILD_FAILURE)
                Policy.debug("Build: project not interesting for this builder " + debugBuilder()
            return null;
        }
        //check if this project has changed
        if (currentDelta != null && currentDelta.findNodeAt(project.getFullPath()) == null) {
            //if the project never existed (not in delta and not in current tree), return null
            if (!project.exists())
                return null;
            //just return an empty delta rooted at this project
            return ResourceDeltaFactory.newEmptyDelta(project);
        }
        //now check against the cache
        IResourceDelta result = (IResourceDelta) deltaCache.getDelta(project.getFullPath(), lastB
        if (result != null)
            return result;

        long startTime = 0L;
        if (Policy.DEBUG_BUILD_DELTA) {
            startTime = System.currentTimeMillis();
            Policy.debug("Computing delta for project: " + project.getName()); //$NON-NLS-1$
        }
        result = ResourceDeltaFactory.computeDelta(workspace, lastBuiltTree, currentTree, project
        deltaCache.cache(project.getFullPath(), lastBuiltTree, currentTree, result);
        if (Policy.DEBUG_BUILD_FAILURE && result == null)
            Policy.debug("Build: no delta " + debugBuilder() + " [" + debugProject() + "] " + pro
        if (Policy.DEBUG_BUILD_DELTA)
            Policy.debug("Finished computing delta, time: " + (System.currentTimeMillis() - start
        return result;
    } finally {
        lock.release();
    }
}
```

Fragment 34: The getDelta method in o.e.core.internal.events.BuildManager.

```
protected IProject[] build(int kind, Map ignored, IProgressMonitor monitor) throws CoreException {
    this.currentProject = getProject();
    if (currentProject == null || !currentProject.isAccessible()) return new IProject[0];

    if (DEBUG)
        System.out.println("\nStarting build of " + currentProject.getName() //$NON-NLS-1$
            + " @ " + new Date(System.currentTimeMillis())); //$NON-NLS-1$
    this.notifier = new BuildNotifier(monitor, currentProject);
    notifier.begin();
    boolean ok = false;
    try {
        notifier.checkCancel();
        kind = initializeBuilder(kind, true);

        if (isWorthBuilding()) {
            if (kind == FULL_BUILD) {
                buildAll();
            } else {
                if ((this.lastState = getLastState(currentProject)) == null) {
                    if (DEBUG)
                        System.out.println("Performing full build since last saved state was not found"); //$NO
                    buildAll();
                } else if (hasClasspathChanged()) {
                    // if the output location changes, do not delete the binary files from old location
                    // the user may be trying something
                    buildAll();
                } else if (nameEnvironment.sourceLocations.length > 0) {
                    // if there is no source to compile & no classpath changes then we are done
                    SimpleLookupTable deltas = findDeltas();
                    if (deltas == null)
                        buildAll();
                    else if (deltas.elementSize > 0)
                        buildDeltas(deltas);
                    else if (DEBUG)
                        System.out.println("Nothing to build since deltas were empty"); //$NON-NLS-1$
                } else {
                    if (hasStructuralDelta()) { // double check that a jar file didn't get replaced in a binary
                        buildAll();
                    } else {
                        if (DEBUG)
                            System.out.println("Nothing to build since there are no source folders and no delta
                        lastState.tagAsNoopBuild();
                    }
                }
            }
            ok = true;
        }
    } catch (CoreException e) {
        Util.log(e, "JavaBuilder handling CoreException while building: " + currentProject.getName()); //$NON-N
        IMarker marker = currentProject.createMarker(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER);
        marker.setAttribute(IMarker.MESSAGE, Messages.bind(Messages.build_inconsistentProject, e.getLocalizedMe
        marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
        marker.setAttribute(IJavaModelMarker.CATEGORY_ID, CategorizedProblem.CAT_BUILDPATH);
    } catch (ImageBuilderInternalException e) {
        Util.log(e.getThrowable(), "JavaBuilder handling ImageBuilderInternalException while building: " + curr
        IMarker marker = currentProject.createMarker(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER);
        marker.setAttribute(IMarker.MESSAGE, Messages.bind(Messages.build_inconsistentProject, e.getLocalizedMe
        marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
        marker.setAttribute(IJavaModelMarker.CATEGORY_ID, CategorizedProblem.CAT_BUILDPATH);
    } catch (MissingSourceFileException e) {
        // do not log this exception since its thrown to handle aborted compiles because of missing source file
        if (DEBUG)
            System.out.println(Messages.bind(Messages.build_missingSourceFile, e.missingSourceFile));
        removeProblemsAndTasksFor(currentProject); // make this the only problem for this project
        IMarker marker = currentProject.createMarker(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER);
        marker.setAttribute(IMarker.MESSAGE, Messages.bind(Messages.build_missingSourceFile, e.missingSourceFi
        marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
    } finally {
        if (!ok)
            // If the build failed, clear the previously built state, forcing a full build next time.
            clearLastState();
        notifier.done();
        cleanup();
    }
    IProject[] requiredProjects = getRequiredProjects(true);
    if (DEBUG)
        System.out.println("Finished build of " + currentProject.getName() //$NON-NLS-1$
            + " @ " + new Date(System.currentTimeMillis())); //$NON-NLS-1$
    return requiredProjects;
}
```

Fragment 35: The build method of o.e.jdt.internal.core.builder.JavaBuilder.

```
private void buildAll() {
    notifier.checkCancel();
    notifier.subTask(Messages.bind(Messages.build_preparingBuild, this.currentProject.getName()));
    if (DEBUG && lastState != null)
        System.out.println("Clearing last state : " + lastState); //$NON-NLS-1$
    clearLastState();
    BatchImageBuilder imageBuilder = new BatchImageBuilder(this, true);
    imageBuilder.build();
    recordNewState(imageBuilder.newState);
}
```

Fragment 36: The buildAll method of o.e.jdt.internal.core.builder.JavaBuilder.

```
private void buildDeltas(SimpleLookupTable deltas) {
    notifier.checkCancel();
    notifier.subTask(Messages.bind(Messages.build_preparingBuild, this.currentProject.getName()));
    if (DEBUG && lastState != null)
        System.out.println("Clearing last state : " + lastState); //$NON-NLS-1$
    clearLastState(); // clear the previously built state so if the build fails, a full build will
    IncrementalImageBuilder imageBuilder = new IncrementalImageBuilder(this);
    if (imageBuilder.build(deltas))
        recordNewState(imageBuilder.newState);
    else
        buildAll();
}
```

Fragment 37: The buildDeltas method of o.e.jdt.internal.core.builder.JavaBuilder.

```
public void build() {
    if (JavaBuilder.DEBUG)
        System.out.println("FULL build"); //$NON-NLS-1$

    try {
        notifier.subTask(Messages.bind(Messages.build_cleaningOutput, this.javaBuilder.
        JavaBuilder.removeProblemsAndTasksFor(javaBuilder.currentProject);
        cleanOutputFolders(true);
        notifier.updateProgressDelta(0.05f);

        notifier.subTask(Messages.build_analyzingSources);
        ArrayList sourceFiles = new ArrayList(33);
        addAllSourceFiles(sourceFiles);
        notifier.updateProgressDelta(0.10f);

        if (sourceFiles.size() > 0) {
            SourceFile[] allSourceFiles = new SourceFile[sourceFiles.size()];
            sourceFiles.toArray(allSourceFiles);

            notifier.setProgressPerCompilationUnit(0.75f / allSourceFiles.length);
            workQueue.addAll(allSourceFiles);
            compile(allSourceFiles);

            if (this.typeLocatorsWithUndefinedTypes != null)
                if (this.secondaryTypes != null && !this.secondaryTypes.isEmpty())
                    rebuildTypesAffectedBySecondaryTypes();
            if (this.incrementalBuilder != null)
                this.incrementalBuilder.buildAfterBatchBuild();
        }

        if (javaBuilder.javaProject.hasCycleMarker())
            javaBuilder.mustPropagateStructuralChanges();
    } catch (CoreException e) {
        throw internalException(e);
    } finally {
        cleanUp();
    }
}
```

Fragment 38: The build method of o.e.jdt.internal.core.builder.BatchImageBuilder.

```
public boolean build(SimpleLookupTable deltas) {
    // initialize builder
    // walk this project's deltas, find changed source files
    // walk prereq projects' deltas, find changed class files & add affected source files
    //   use the build state # to skip the deltas for certain prereq projects
    //   ignore changed zip/jar files since they caused a full build
    // compile the source files & acceptResult()
    // compare the produced class files against the existing ones on disk
    // recompile all dependent source files of any type with structural changes or new/removed secondary
    // keep a loop counter to abort & perform a full build

    if (JavaBuilder.DEBUG)
        System.out.println("INCREMENTAL build"); //$NON-NLS-1$

    try {
        resetCollections();

        notifier.subTask(Messages.build_analyzingDeltas);
        if (javaBuilder.hasBuildpathErrors()) {
            // if a missing class file was detected in the last build, a build state was saved since its
            // but we need to rebuild every source file since problems were not recorded
            // AND to avoid the infinite build scenario if this project is involved in a cycle, see bug
            // we need to avoid unnecessary deltas caused by doing a full build in this case
            javaBuilder.currentProject.deleteMarkers(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER, false,
            addAllSourceFiles(sourceFiles);
            notifier.updateProgressDelta(0.25f);
        } else {
            IResourceDelta sourceDelta = (IResourceDelta) deltas.get(javaBuilder.currentProject);
            if (sourceDelta != null)
                if (!findSourceFiles(sourceDelta)) return false;
            notifier.updateProgressDelta(0.10f);

            Object[] keyTable = deltas.keyTable;
            Object[] valueTable = deltas.valueTable;
            for (int i = 0, l = valueTable.length; i < l; i++) {
                IResourceDelta delta = (IResourceDelta) valueTable[i];
                if (delta != null) {
                    IProject p = (IProject) keyTable[i];
                    ClasspathLocation[] classFoldersAndJars = (ClasspathLocation[]) javaBuilder.binaryLoc
                    if (classFoldersAndJars != null)
                        if (!findAffectedSourceFiles(delta, classFoldersAndJars, p)) return false;
                }
            }
            notifier.updateProgressDelta(0.10f);

            notifier.subTask(Messages.build_analyzingSources);
            addAffectedSourceFiles();
            notifier.updateProgressDelta(0.05f);
        }

        this.compileLoop = 0;
        float increment = 0.40f;
        while (sourceFiles.size() > 0) { // added to in acceptResult
            if (++this.compileLoop > MaxCompileLoop) {
                if (JavaBuilder.DEBUG)
                    System.out.println("ABORTING incremental build... exceeded loop count"); //$NON-NLS-1
                return false;
            }
            notifier.checkCancel();

            SourceFile[] allSourceFiles = new SourceFile[sourceFiles.size()];
            sourceFiles.toArray(allSourceFiles);
            resetCollections();

            workQueue.addAll(allSourceFiles);
            notifier.setProgressPerCompilationUnit(increment / allSourceFiles.length);
            increment = increment / 2;
            compile(allSourceFiles);
            removeSecondaryTypes();
            addAffectedSourceFiles();
        }
        if (this.hasStructuralChanges && javaBuilder.javaProject.hasCycleMarker())
            javaBuilder.mustPropagateStructuralChanges();
    } catch (AbortIncrementalBuildException e) {
        // abort the incremental build and let the batch builder handle the problem
        if (JavaBuilder.DEBUG)
            System.out.println("ABORTING incremental build... problem with " + e.qualifiedTypeName + //$N
                ". Likely renamed inside its existing source file."); //$NON-NLS-1$
        return false;
    } catch (CoreException e) {
        throw internalException(e);
    } finally {
        cleanUp();
    }
    return true;
}
```

Fragment 39: The build method of o.e.jdt.internal.core.builder.IncrementalImageBuilder.

```
/* Compile the given elements, adding more elements to the work queue
* if they are affected by the changes.
*/
protected void compile(SourceFile[] units) {
    if (this.filesWithAnnotations != null && this.filesWithAnnotations.elementSize > 0)
        // will add files that have annotations in acceptResult() & then processAnnotations() bef
        this.filesWithAnnotations.clear();

    // notify CompilationParticipants which source files are about to be compiled
    BuildContext[] participantResults = this.javaBuilder.participants == null ? null : notifyPart
    if (participantResults != null && participantResults.length > units.length) {
        units = new SourceFile[participantResults.length];
        for (int i = participantResults.length; --i >= 0;)
            units[i] = participantResults[i].sourceFile;
    }

    int unitsLength = units.length;
    this.compiledAllAtOnce = unitsLength <= MAX_AT_ONCE;
    if (this.compiledAllAtOnce) {
        // do them all now
        if (JavaBuilder.DEBUG)
            for (int i = 0; i < unitsLength; i++)
                System.out.println("About to compile " + units[i].typeLocator()); //$NON-NLS-1$
        compile(units, null, true);
    } else {
        SourceFile[] remainingUnits = new SourceFile[unitsLength]; // copy of units, removing uni
        System.arraycopy(units, 0, remainingUnits, 0, unitsLength);
        int doNow = unitsLength < MAX_AT_ONCE ? unitsLength : MAX_AT_ONCE;
        SourceFile[] toCompile = new SourceFile[doNow];
        int remainingIndex = 0;
        boolean compilingFirstGroup = true;
        while (remainingIndex < unitsLength) {
            int count = 0;
            while (remainingIndex < unitsLength && count < doNow) {
                // Although it needed compiling when this method was called, it may have
                // already been compiled when it was referenced by another unit.
                SourceFile unit = remainingUnits[remainingIndex];
                if (unit != null && (compilingFirstGroup || this.workQueue.isWaiting(unit))) {
                    if (JavaBuilder.DEBUG)
                        System.out.println("About to compile #" + remainingIndex + " : "+ unit.ty
                    toCompile[count++] = unit;
                }
                remainingUnits[remainingIndex++] = null;
            }
            if (count < doNow)
                System.arraycopy(toCompile, 0, toCompile = new SourceFile[count], 0, count);
            if (!compilingFirstGroup)
                for (int a = remainingIndex; a < unitsLength; a++)
                    if (remainingUnits[a] != null && this.workQueue.isCompiled(remainingUnits[a])
                        remainingUnits[a] = null; // use the class file for this source file sinc
            compile(toCompile, remainingUnits, compilingFirstGroup);
            compilingFirstGroup = false;
        }
    }

    if (participantResults != null) {
        for (int i = participantResults.length; --i >= 0;)
            if (participantResults[i] != null)
                recordParticipantResult(participantResults[i]);

        processAnnotations(participantResults);
    }
}
```

Fragment 40: The first compile method of o.e.jdt.internal.core.builder.AbstractImageBuilder.

```
protected void compile(SourceFile[] units, SourceFile[] additionalUnits, boolean compilingFirstGroup) {
    if (units.length == 0) return;
    notifier.aboutToCompile(units[0]); // just to change the message

    // extend additionalFilenames with all hierarchical problem types found during this entire build
    if (!problemSourceFiles.isEmpty()) {
        int toAdd = problemSourceFiles.size();
        int length = additionalUnits == null ? 0 : additionalUnits.length;
        if (length == 0)
            additionalUnits = new SourceFile[toAdd];
        else
            System.arraycopy(additionalUnits, 0, additionalUnits = new SourceFile[length + toAdd], 0, length);
        for (int i = 0; i < toAdd; i++)
            additionalUnits[length + i] = (SourceFile) problemSourceFiles.get(i);
    }
    String[] initialTypeNames = new String[units.length];
    for (int i = 0, l = units.length; i < l; i++)
        initialTypeNames[i] = units[i].initialTypeName;
    nameEnvironment.setNames(initialTypeNames, additionalUnits);
    notifier.checkCancel();
    try {
        inCompiler = true;
        compiler.compile(units);
    } catch (AbortCompilation ignored) {
        // ignore the AbortCompilcation coming from BuildNotifier.checkCancelWithinCompiler()
        // the Compiler failed after the user has chose to cancel... likely due to an OutOfMemory error
    } finally {
        inCompiler = false;
    }
    // Check for cancel immediately after a compile, because the compiler may
    // have been cancelled but without propagating the correct exception
    notifier.checkCancel();
}
```

Fragment 41: Fragment 40: The second compile method of o.e.jdt.internal.core.builder.AbstractImageBuilder.

```
/**
 * General API
 * -> compile each of supplied files
 * -> recompile any required types for which we have an incomplete principle structure
 */
public void compile(ICompilationUnit[] sourceUnits) {
    CompilationUnitDeclaration unit = null;
    int i = 0;
    try {
        // build and record parsed units

        beginToCompile(sourceUnits);

        // process all units (some more could be injected in the loop by the lookup environment)
        for (; i < this.totalUnits; i++) {
            unit = unitsToProcess[i];
            try {
                if (options.verbose)
                    this.out.println(
                        Messages.bind(Messages.compilation_process,
                        new String[] {
                            String.valueOf(i + 1),
                            String.valueOf(this.totalUnits),
                            new String(unitsToProcess[i].getFileName())
                        }));
                process(unit, i);
            } finally {
                // cleanup compilation unit result
                unit.cleanUp();
            }
            unitsToProcess[i] = null; // release reference to processed unit declaration
            requestor.acceptResult(unit.compilationResult.tagAsAccepted());
            if (options.verbose)
                this.out.println(
                    Messages.bind(Messages.compilation_done,
                    new String[] {
                        String.valueOf(i + 1),
                        String.valueOf(this.totalUnits),
                        new String(unit.getFileName())
                    }));
        }
    } catch (AbortCompilation e) {
        this.handleInternalException(e, unit);
    } catch (Error e) {
        this.handleInternalException(e, unit, null);
        throw e; // rethrow
    } catch (RuntimeException e) {
        this.handleInternalException(e, unit, null);
        throw e; // rethrow
    } finally {
        this.reset();
    }
    if (options.verbose) {
        if (this.totalUnits > 1) {
            this.out.println(
                Messages.bind(Messages.compilation_units, String.valueOf(this.totalUnits)));
        } else {
            this.out.println(
                Messages.bind(Messages.compilation_unit, String.valueOf(this.totalUnits)));
        }
    }
}
```

Fragment 42: The compile method of o.e.jdt.internal.compiler.Compiler.

```
protected void startup(IProgressMonitor monitor) throws CoreException {
    // ensure the tree is locked during the startup notification
    try {
        _workManager = new WorkManager(this);
        _workManager.startup(null);
        fileSystemManager = new FileSystemResourceManager(this);
        fileSystemManager.startup(monitor);
        pathVariableManager = new PathVariableManager();
        pathVariableManager.startup(null);
        natureManager = new NatureManager();
        natureManager.startup(null);
        buildManager = new BuildManager(this, getWorkManager().getLock());
        buildManager.startup(null);
        notificationManager = new NotificationManager(this);
        notificationManager.startup(null);
        markerManager = new MarkerManager(this);
        markerManager.startup(null);
        synchronizer = new Synchronizer(this);
        refreshManager = new RefreshManager(this);
        saveManager = new SaveManager(this);
        saveManager.startup(null);
        //must start after save manager, because (read) access to tree is needed
        refreshManager.startup(null);
        aliasManager = new AliasManager(this);
        aliasManager.startup(null);
        propertyManager = ResourcesCompatibilityHelper.createPropertyManager();
        propertyManager.startup(monitor);
        charsetManager = new CharsetManager(this);
        charsetManager.startup(null);
        contentDescriptionManager = new ContentDescriptionManager();
        contentDescriptionManager.startup(null);
    } finally {
        //unlock tree even in case of failure, otherwise shutdown will also fail
        treeLocked = null;
    }
}
```

Fragment 43: The o.e.core.internal.resources.Workspace's startup method calls each manager's constructor.

```
public abstract class InternalBuilder {
    private static BuildManager buildManager = ((Workspace) ResourcesPlugin.getWorkspace()).getBuildManager();
```

Fragment 44: When o.e.core.internal.events.InternalBuilder wants a BuildManager, it does not call the BuildManager's constructor, but instead gets the Workspace's BuildManager.

```
private void basicBuild(final IProject project, final int trigger, final MultiStatus status, final
    if (!project.isAccessible())
        return;
    final ICommand[] commands = ((Project) project).internalGetDescription().getBuildSpec(false);
    if (commands.length == 0)
        return;
    ISafeRunnable code = new ISafeRunnable() {
        public void handleException(Throwable e) {
            if (e instanceof OperationCanceledException)
                throw (OperationCanceledException) e;
            // don't log the exception....it is already being logged in Workspace#run
            // should never get here because the lower-level build code wrappers
            // builder exceptions in core exceptions if required.
            String message = e.getMessage();
            if (message == null)
                message = NLS.bind(Messages.events_unknown, e.getClass().getName(), project.getNam
            status.add(new Status(IStatus.WARNING, ResourcesPlugin.PI_RESOURCES, IResourceStatus.I
        }

        public void run() throws Exception {
            basicBuild(project, trigger, commands, status, monitor);
        }
    };
    SafeRunner.run(code);
}
```

Fragment 45: The o.e.core.internal.events.BuildManager contains an implementation of the o.e.core.runtime.ISafeRunnable interface as an anonymous class in its third basicBuild method.

```
/**
 * Creates and returns an ArrayList of BuilderPersistentInfo.
 * The list includes entries for all builders that are
 * in the builder spec, and that have a last built state, even if they
 * have not been instantiated this session.
 */
public ArrayList createBuildersPersistentInfo(IProject project) throws CoreException {
```

Fragment 46: The createBuildersPersistentInfo method in o.e.core.internal.events.BuildManager throws a CoreException, as seen in its header.

```
/**
 * Collects the set of ElementTrees we are still interested in,
 * and removes references to any other trees.
 */
protected void collapseTrees() throws CoreException {
    //collect trees we're interested in

    //trees for plugin saved states
    ArrayList trees = new ArrayList();
    for (Iterator i = savedStates.values().iterator(); i.hasNext();) {
        SavedState state = (SavedState) i.next();
        if (state.oldTree != null) {
            trees.add(state.oldTree);
        }
    }

    //trees for builders
    IProject[] projects = workspace.getRoot().getProjects();
    for (int i = 0; i < projects.length; i++) {
        IProject project = projects[i];
        if (project.isOpen()) {
            ArrayList builderInfos = workspace.getBuildManager().createBuildersPers
            if (builderInfos != null) {
                for (Iterator it = builderInfos.iterator(); it.hasNext();) {
                    BuilderPersistentInfo info = (BuilderPersistentInfo) it.next();
                    trees.add(info.getLastBuiltTree());
                }
            }
        }
    }

    //no need to collapse if there are no trees at this point
    if (trees.isEmpty())
        return;

    //the complete tree
    trees.add(workspace.getElementTree());

    //collapse the trees
    //sort trees in topological order, and set the parent of each
    //tree to its parent in the topological ordering.
    ElementTree[] treeArray = new ElementTree[trees.size()];
    trees.toArray(treeArray);
    ElementTree[] sorted = sortTrees(treeArray);
    // if there was a problem sorting the tree, bail on trying to collapse.
    // We will be able to GC the layers at a later time.
    if (sorted == null)
        return;
    for (int i = 1; i < sorted.length; i++)
        sorted[i].collapseTo(sorted[i - 1]);
}
```

Fragment 47: The createBuildersPersistentInfo is called by the collapseTrees method in o.e.core.internal.resources.SaveManager.

```
public IStatus save(int kind, Project project, IProgressMonitor monitor) throws CoreException {
    monitor = Policy.monitorFor(monitor);
    try {
        isSaving = true;
        String message = Messages.resources_saving_0;
        monitor.beginTask(message, 7);
        message = Messages.resources_saveWarnings;
        MultiStatus warnings = new MultiStatus(ResourcesPlugin.PI_RESOURCES, IStatus.WARNING, m
        ISchedulingRule rule = project != null ? (IResource) project : workspace.getRoot();
        try {
            workspace.prepareOperation(rule, monitor);
            workspace.beginOperation(false);
            hookStartSave(kind, project);
            long start = System.currentTimeMillis();
            Map contexts = computeSaveContexts(getSaveParticipantPlugins(), kind, project);
            broadcastLifecycle(PREPARE_TO_SAVE, contexts, warnings, Policy.subMonitorFor(monito
            try {
                broadcastLifecycle(SAVING, contexts, warnings, Policy.subMonitorFor(monitor, 1)
                switch (kind) {
                    case ISaveContext.FULL_SAVE :
```

•••

```
        //remove unused files
        removeUnusedSafeTables();
        removeUnusedTreeFiles();
        workspace.getFileSystemManager().getHistoryStore().clean(Policy.subMonitc
        // write out all metainfo (e.g., workspace/project descriptions)
        saveMetaInfo(warnings, Policy.subMonitorFor(monitor, 1));
        break;
    case ISaveContext.SNAPSHOT :
        snapTree(workspace.getElementTree(), Policy.subMonitorFor(monitor, 1));
        // snapshot the markers and sync info for the workspace
        persistMarkers = 0l;
        persistSyncInfo = 0l;
        visitAndSnap(workspace.getRoot());
        monitor.worked(1);
        if (Policy.DEBUG_SAVE) {
            Policy.debug("Total Snap Markers: " + persistMarkers + "ms"); //$NON-
            Policy.debug("Total Snap Sync Info: " + persistSyncInfo + "ms"); //$N
        }
        collapseTrees();
        clearSavedDelta();
        // write out all metainfo (e.g., workspace/project descriptions)
        saveMetaInfo(warnings, Policy.subMonitorFor(monitor, 1));
        break;
```

•••

```
                    break;
                }
                // save contexts
                commit(contexts);
                if (kind == ISaveContext.FULL_SAVE)
                    removeClearDeltaMarks();
                //this must be done after committing save contexts to update participant save nu
                saveMasterTable();
                broadcastLifecycle(DONE_SAVING, contexts, warnings, Policy.subMonitorFor(monitor
                hookEndSave(kind, project, start);
                return warnings;
            } catch (CoreException e) {
                broadcastLifecycle(ROLLBACK, contexts, warnings, Policy.subMonitorFor(monitor, 1
                // rollback ResourcesPlugin master table
                restoreMasterTable();
                throw e; // re-throw
            }
        } catch (OperationCanceledException e) {
            workspace.getWorkManager().operationCanceled();
            throw e;
        } finally {
            workspace.endOperation(rule, false, Policy.monitorFor(null));
        }
    } finally {
        isSaving = false;
        monitor.done();
    }
}
```

Fragment 48: The save method of o.e.core.internal.resources.SaveManager has a "throws" clause also, but it has the call to collapseTrees in a try statement and the CoreException is caught in a catch statement afterwards.