Component Dependency Extractor 1.0 Component Specification

1. Design

A TopCoder component may have many dependencies, including other TopCoder components or third-party software. This component's task is to extract the dependencies for one or more components. The current version of the component parses the build file of the component to extract the dependency information needed.

This design provides the entities that can be used to specify dependencies between components. They are Component, ComponentDependency and DependenciesEntry. ComponentLanguage enumeration is used to specify the component language (currently Java or .NET/C#). DependencyCategory enumeration provides values for compile and test dependencies. DependencyType is used for separating internal TopCoder components and external ones.

DependenciesEntryExtractor is the interface for dependency extractor implementations. BaseDependenciesEntryExtractor is extension of this interface that must be used by extractors that support dependency filtering by type and category.

This design contains implementations of DependenciesEntryExtractor for each component language: JavaDependenciesEntryExtractor and DotNetDependenciesEntryExtractor. Also MultipleFormatDependenciesEntryExtractor is provided. It uses both extractors specified above and can extract dependencies for both Java and .NET components.

BuildFileProvider interface allows extractor implementations to abstract away from build files source peculiarity. Two implementations of this interface are provided: LocalBuildFileProvider and ZipBuildFileProvider. Thus Java and .NET extractors can extract files from both build files on local file disk and from distribution archive files (JAR and ZIP accordingly).

This design also provides a standalone command line utility that can be used to extract dependencies from component build/distribution files and save them to file. ComponentDependencyExtractorUtility is the main class of this application. The user can provide parameters to it with configuration file and command line arguments.

NOTE: please see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6296446 and there is an indentation problem in transformer in JDK5. also see post : http://forums.topcoder.com/?module=Thread&threadID=615299&start=0&mc=1#984005

1.1 Design Patterns

Composite pattern – MultipleFormatDependenciesEntryExtractor extends BaseDependenciesEntryExtractor and aggregates other BaseDependenciesEntryExtractor instances to perform its functions.

Strategy pattern – ComponentDependencyExtractorUtility uses implementations of DependenciesEntryPersistence and DependenciesEntryExtractor; DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor use implementstions of BuildFileProvider; MultipleFormatDependenciesEntryExtractor uses the provided subclasses of BaseDependenciesEntryExtractor.

1.2 Industry Standards

XML, Ant, Nant, ZIP, JAR

1.3 Required Algorithms

1.3.1 Extraction of dependencies from build.xml (Java components)

This algorithm corresponds to JavaDependenciesEntryExtractor#extract(). This method can also process JAR distribution files, but ZipBuildFileProvider is used to perform the real work of extracting files from archive. Thus implementation of this algorithm don't have to worry about what build file source is used (file system or JAR archive).

This is the basic structure of build.xml-file (it includes only elements we are interested in):

```
<?xml version="1.0"?>
<project basedir="...">

  <property name="..." value="..." />

  <property file="..." />

  <import file="..." />

  <path id="..." location="..." path="...">
    <path refid="..." />
    <pathelement location="..." />
    <pathelement path="..." />
  </path>

  <macrodef name="...">
    <xxxxx>
      <javac>
        <classpath refid="..." />
        <classpath location="..." />
        <classpath path="..." />
        <classpath>
          <path refid="..." />
          <pathelement location="..." />
          <pathelement path="..." />
        </classpath>
      </javac>
      <junit>
        <classpath refid="..." />
        <classpath location="..." />
        <classpath path="..." />
        <classpath>
          <path refid="..." />
          <pathelement location="..." />
          <pathelement path="..." />
        </classpath>
      </junit>
    </xxxxx>
  </macrodef>

  <target name="...">
    <macrodef_name/>
    <javac>
      <classpath refid="..." />
      <classpath location="..." />
      <classpath path="..." />
      <classpath>
        <path refid="..." />
        <pathelement location="..." />
        <pathelement path="..." />
      </classpath>
    </javac>
    <junit>
      <classpath refid="..." />
      <classpath location="..." />
      <classpath path="..." />
      <classpath>
        <path refid="..." />
        <pathelement location="..." />
        <pathelement path="..." />
      </classpath>
    </junit>
  </target>

</project>
```

To understand the meaning of each element in the given file structure, please see Ant reference (Ant Tasks link in the contents list): http://ant.apache.org/manual/

The following elements must be processed by the parser in special way:

```
<property name="distfilename" value="..." />

<property name="component" value="..." />

<property name="component_version" value="..." />

<target name="compile">
  ...
</target>

<target name="test">
  ...
</target>
```

The following information must be extracted from build files:

- The target component name. It is extracted from <property> element with name="distfilename". If this property is not found, other <property> element with name="component" is used. (The value of latter one must be converter to lower case).
- The target component version. It is extracted from <property> element with name="component_version".
- The path of each dependency component. Paths are extracted from <path> and <classpath> elements.
- The type of dependency component. If component path starts with "../tcs/lib/tcs" or "lib/tcs", then this is internal TopCoder component, otherwise it is external component.
- The dependency category. If component path is mentioned in <javac> element in <target> with name="compile", then it's a compile dependency. If component path is mentioned in <junit> element in <target> with name="test", then it's a test dependency.
- The dependency component name and version. Currently the following path formats are supported:
  - "file_path/component_name-component_version.jar";
  - "file_path/component_version/component_name.jar";
  - "file_path/component_version/component_name.jar";
  - "file_path/component_name.jar".

    Here file_path – any path substring, component_name – any path substring that doesn't include path separator, component_version – not empty and contains only digits, dots and optional letter at the end.

Note that properties specified in format "${property_name}" must be processed accordingly in all strings retrieved from build file.

Unlike Nant-build files, Ant ones can contain links to additional property files in <property file="..."> elements. These files have the same format as Java property files. They must be loaded and properly parsed.

Also note that each build file can link other build files with use of <import> element. Such linked build files must be loaded and properly processed as if they were parts of the main build file.

Please see implementation notes of extract() and all processXXX() methods of JavaDependenciesEntryExtractor in TC UML Tool class diagram for recommended steps that can be used to implement this algorithm.

1.3.2 Extraction of dependencies from default.build (.NET components)

This algorithm corresponds to DotNetDependenciesEntryExtractor#extract(). This method can also process ZIP distribution files, but ZipBuildFileProvider is used to perform the real work of extracting

files from archive. Thus implementation of this algorithm don't have to worry about what build file source is used (file system or ZIP archive).

This algorithm is very similar to the one described in the section 1.3.1. But default.xml and default.build has some differences in their formats.

This is the basic structure of default.build-file (it includes only elements we are interested in):

```
<?xml version="1.0" encoding="utf-8"?>
<project basedir="...">

  <property name="..." value="..." />

  <include buildfile="..." />

  <fileset id="..." refid="...">
    <include name="..." />
    <includes name="..." />
  </fileset>

  <assemblyfileset id="..." refid="...">
    <include name="..." />
    <includes name="..." />
  </assemblyfileset>

  <target name="...">
    <csc>
      <references refid="..." />
      <references>
        <include name="..." />
        <includes name="..." />
      </references>
    </csc>
    <copy file="..." />
  </target>

</project>
```

To understand the meaning of each element in the given file structure, please see NAnt reference (Task Reference and Type Reference links): http://nant.sourceforge.net/release/latest/help/index.html

The following elements must be processed by the parser in special way:

```
  <property name="distfilename" value="..." />

  <property name="component" value="..." />

  <property name="component_version" value="..." />

  <target name="compile">
    ...
  </target>

  <target name="compile_tests">
    ...
  </target>

  <target name="copy_dependencies">
    ...
  </target>
```

The following information must be extracted from build files:

- The target component name. It is extracted from <property> element with name="distfilename". If this property is not found, other <property> element with name="component" is used. (The latter one must be converter to lower case).
- The target component version. It is extracted from <property> element with name="component_version".

- The path of each dependency component. Paths are extracted from <fileset>, <assemblyfileset> and <references> elements.
- The type of dependency component. If component path starts with "..\tcs\bin\tcs\", then this is internal TopCoder component, otherwise it is external component.
- The dependency category. If component path is mentioned in <csc> element in <target> with name="compile", then it's a compile dependency. If component path is mentioned in <csc> element in <target> with name="compile_tests" or name="copy_dependencies", then it's a test dependency.
- The dependency component name and version. Currently the following path formats are supported:
  - "file_path\component_name\component_version\dll_name.dll";
  - "file_path\component_name-component_version.dll";
  - "file_path\component_name\component_version\bin\dll_name.dll";
  - "file_path\component_name.dll".

    Here file_path – any path substring, component_name and dll_name – any path substring that doesn't include path separator, component_version – not empty and contains only digits, dots and optional letter at the end.

Note that properties specified in format "${property_name}" must be processed accordingly in all strings retrieved from build file.

Also note that each build file can link other build files with use of <include> element. Such linked build files must be loaded and properly processed as if they were parts of the main build file.

Please see implementation notes of extract() and all processXXX() methods of DotNetDependenciesEntryExtractor in TC UML Tool class diagram for recommended steps that can be used to implement this algorithm.

1.3.3 *Saving dependencies to XML file*

This algorithm corresponds to the method DefaultXmlDependenciesEntryPersistence#save(). It writes the list of dependencies to XML file.
The output XML content is generated with proper indentation that makes the file readable for the user.
The output file of this algorithm has the following format:

```
<?xml version="1.0"?>
<components>
  <component name=”component1_name” version=”component1_version”
language=”component1_language”>
    <dependency type=”c1_dep1_type” category=”c1_dep1_category” name=”c1_dep1_name”
version=”c1_dep1_version” path=”c1_dep1_path />
    <dependency type=”c1_dep2_type” category=”c1_dep2_category” name=”c1_dep2_name”
version=”c1_dep2_version” path=”c1_dep2_path />
  </component>
  <component name=”component2_name” version=”component2_version”
language=”component2_language”>
    <dependency type=”c2_dep1_type” category=”c2_dep1_category” name=”c2_dep1_name”
version=”c2_dep1_version” path=”c2_dep1_path />
  </component>
</components>
```

This is XML DTD for the provided format:

```
<!ELEMENT components (component*)>
<!ELEMENT component (dependency*)>
<!ELEMENT dependency EMPTY>

<!ATTLIST component name CDATA #REQUIRED>
<!ATTLIST component version CDATA #REQUIRED>
<!ATTLIST component language (java|dot_net) #REQUIRED>
```

```
<!ATTLIST dependency type (internal|external) #REQUIRED>
<!ATTLIST dependency category (compile|test) #REQUIRED>
<!ATTLIST dependency name CDATA #REQUIRED>
<!ATTLIST dependency version CDATA #REQUIRED>
<!ATTLIST dependency path CDATA #REQUIRED>
```

Please see the demo section for a sample of such file.

The following steps produce the required output (developers of this component may use another, more efficient algorithm):

1. Create document build factory:
   DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
2. Create document builder:
   DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
3. Create new document:
   Document doc = docBuilder.newDocument();
4. Create root XML element:
   Element root = doc.createElement("components");
5. Add element to the document:
   doc.appendChild(root);
6. For each entry from entries do:

6.1. Create new subelement:
   Element component = doc.createElement("component");
6.2. Set name attribute:
   component.setAttribute("name", entry.getTargetComponent().getName());
6.3. Set version attribute:
   component.setAttribute("version", entry.getTargetComponent().getVersion());
6.4. Set language attribute:
   component.setAttribute("language", (entry.getTargetComponent().getLanguage() == ComponentLanguage.JAVA) ? "java" : "dot_net");
6.5. For each dependency from entry.getDependencies() do:
6.5.1. Create new dependency element:
   dependencyElement = doc.createElement("dependency");
6.5.2. Set type attribute:
   dependencyElement.setAttribute("type", (dependency.getType() == DependencyType.INTERNAL) ? "internal" : "external");
6.5.3. Set category attribute:
   dependencyElement.setAttribute("category", (dependency.getCategory() == DependencyCategory.COMPILE) ? "compile" : "test");
6.5.4. Set name attribute:
   dependencyElement.setAttribute("name", dependency.getName());
6.5.5. Set version attribute:
   dependencyElement.setAttribute("version", dependency.getVersion());
6.5.6. Set path attribute:
   dependencyElement.setAttribute("path", dependency.getPath());
6.5.7. Add element to the document:
   component.appendChild(dependencyElement);
6.6. Add element to the document:
   root.appendChild(component);
7. Create transformer factory:
   TransformerFactory transfac = TransformerFactory.newInstance();
8. Create transformer:
   Transformer trans = transfac.newTransformer();
9. Switch on the indentation flag:
   trans.setOutputProperty(OutputKeys.INDENT, "yes");
10. Create string writer:
   StringWriter sw = new StringWriter();

11. Create stream result:
    StreamResult result = new StreamResult(sw);
12. Create DOM source:
    DOMSource source = new DOMSource(doc);
13. Transform the document:
    trans.transform(source, result);
14. Convert result stream to string:
    String xmlString = sw.toString();
15. Create a text writer:
    writer = new OutputStreamWriter(os);
16. Write text to the output stream:
    writer.write(xmlString, 0, xmlString.length());

1.3.4 Loading of dependencies from XML file

This algorithm corresponds to the method DefaultXmlDependenciesEntryPersistence#load(). Please see section 1.3.3 for the proper format of XML file.

The following steps produce the required output (developers of this component may use another, more efficient algorithm):

1. If file filePath doesn't exist then throw an exception.
2. Create document builder factory:
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
3. Load XML document:
   Document doc = factory.newDocumentBuilder().parse(filePath);
4. Get XML root document:
   Element root = doc.getDocumentElement();
5. Get the list of "component" elements:
   NodeList componentList = root.getElementsByTagName("component");
6. result = new ArrayList<DependenciesEntry>();
7. For each componentElement:Element from componentList do:
7.1. If componentElement.getTagName() != "component" then throw an exception.
7.2. Read component name:
   name = componentElement.getAttribute("name");
7.3. Read component version:
   version = componentElement.getAttribute("version");
7.4. Read component language:
   languageStr = componentElement.getAttribute("language");
7.5. If languageStr = "java" then language = ComponentLanguage.JAVA;
7.6. Else if languageStr = "dot_net" then language = ComponentLanguage.DOT_NET;
7.7. Else throw an exception.
7.8. Create component:
   component = new Component(name, version, language);
7.9. Create a list of component dependencies:
   dependencies = new ArrayList<ComponentDependency>();
7.10. Get dependency nodes:
   NodeList dependencyList = componentElement.getElementsByTagName("*");
7.11. For each dependencyElement:Element from dependencyList do:
7.11.1. If dependencyElement.getTagName() != "dependency" then throw an exception.
7.11.2. Get dependency type:
   typeStr = componentElement.getAttribute("type");
7.11.3. If typeStr = "internal" then type = DependencyType.INTERNAL;
7.11.4. Else if typeStr = "external" then type = DependencyType.EXTERNAL;
7.11.5. Else throw an exception.
7.11.6. Get dependency category:
   categoryStr = componentElement.getAttribute("category");
7.11.7. If categoryStr = "compile" then category = DependencyCategory.COMPILE;

7.11.8. Else if categoryStr = "test" then category = DependencyCategory.TEST;
7.11.9. Else throw an exception.
7.11.10. Get dependency component name:
name = dependencyElement.getAttribute("name");
7.11.11. Get dependency component version:
version = componentElement.getAttribute("version");
7.11.12. Get dependency component path:
path = componentElement.getAttribute("path");
7.11.13. Create dependency instance:
dependency = new ComponentDependency(name, version, language, category, type, path);
7.11.14. Add dependency to the list:
dependencies.add(dependency);
7.12. Create new dependency entry:
entry = new DependenciesEntry(component, dependencies);
7.13. Add entry to the result list:
result.add(entry);

1.3.5 Execution of standalone application

This algorithm corresponds to the static method ComponentDependencyExtractorUtility#main().
It read the configuration parameters from configuration file, extracts them from the command line arguments, locates the list of files to be process, extracts the list of dependency entries and writes it to the specified file.

Please see sections 3.2.2 and 3.2.3 of CS for details about configuration parameters, supported command line switches and arguments.

This method MUST NOT throw any exception. Instead it must print out the detailed error explanation to the standard output and terminate.

1. Create a command line utility:
CommandLineUtility commandLineUtility = new CommandLineUtility();
2. For each command line switch (see CS 3.2.3) do:
2.1. Create switch:
Switch switch = new Switch(...);
2.2. Add switch to the command line utility:
commandLineUtility.addSwitch(switch);
3. Parse command line arguments:
commandLineUtility.parse(args);
4. If error occurred while parsing the arguments, print it to System.out and return.
5. If user requests the help (with -h, -? or -help) then
5.1. Get command line usage string:
usageString = commandLineUtility.getUsageString();
5.2. Print string to standard output:
System.out.println(usageString);
5.3. Return.
6. Create a configuration file manager:
ConfigurationFileManager manager = new ConfigurationFileManager();
7. Get configuration file name (use default if not specified):
configFileName = commandLineUtility.getSwitch(...).getValue();
8. Load configuration file (print error message and return if file doesn't exist):

manager.loadFile("com.topcoder.util.dependency.extractor.utility.ComponentDependencyExtractorUtility", filePath);
9. Get configuration for this component:
config =
manager.getConfiguration("com.topcoder.util.dependency.extractor.utility.ComponentDependencyExtractorUtility");

10. If config is null then print error message and return.
11. Get persistence class name from config (use default if not specified):
    persistenceClassName = config.getPropertyValue(...);
    Update persistenceClassName if "-f" or "-pclass" switch is present:
    persistenceClassName = commandLineUtility.getSwitch(...).getValue();
12. Get persistence configuration from config (print error message and return if not provided):
    persistenceConfig = config.getChild(...);
13. If output file name is provided in the command line (-o switch) then
13.1. Get output file name:
    outputFileName = commandLineUtility.getSwitch(...).getValue();
13.2. Put outputFileName to the persistence configuration:
    persistenceConfig.setPropertyValue(...);
14. If log factory key is provided in config:
14.1. Read key from config:
    logFactoryKey = config.getPropertyValue(...);
14.2. Read OF config name from config (use default if not provided):
    objectFactoryConfigName = config.getPropertyValue(...);
14.3. Get OF config:
    objectFactoryConfig = config.getChild(...);
14.4. Create object factory specification:
    cosf = new ConfigurationObjectSpecificationFactory(config);
14.5. Create object factory:
    objFactory = new ObjectFactory(cosf);
14.6. Create log factory with OF:
    logFactory = objFactory.createObject(logFactoryKey);
14.7. Assign log factory to manager:
    LogManager.setLogFactory(logFactory);
15. Read logger name from config:
   loggerName = config.getPropertyValue(...);
16. If "-nolog" switch is not provided then create logger:
    logger = LogManager.getLog(loggerName);
    Else logger = null.
17. Get extractor class name from config:
    extractorClassName = config.getPropertyValue(...);
    Update extractorClassName if "-eclass" switch is present:
    extractorClassName = commandLineUtility.getSwitch(...).getValue();
18. Read dependency types from config (use default if not specified):
    dependencyTypesStr = config.getPropertyValue(...);
    Update dependencyTypesStr if "-dtype" switch is present:
    dependencyTypesStr = commandLineUtility.getSwitch(...).getValue();
19. Parse dependencyTypesStr to dependencyTypes:Set<DependencyType> (use comma as a separator).
20. Read dependency categories from config (use default if not specified):
    dependencyCategoriesStr = config.getPropertyValue(...);
    Update dependencyCategoriesStr if "-dcat" switch is present:
    dependencyCategoriesStr = commandLineUtility.getSwitch(...).getValue();
21. Parse dependencyCategoriesStr to dependencyCategories:Set<DependencyCategory> (use comma as a separator).
22. Create an extractor:DependenciesEntryExtractor via reflection. Use extractorClassName to locate the class. Use logger as the only constructor argument.
23. If extractor is of BaseDependenciesEntryExtractor type then:
23.1. Set dependency types to be extracted:
    extractor.setExtractedTypes(dependencyTypes);
23.2. Set dependency categories to be extracted:
    extractor.setExtractedCategories(dependencyCategories);

24. Create a persistence:DependenciesEntryPersistence via reflection. Use persistenceClassName to locate the class and persistenceConfig as the only constructor argument.
25. Get the string with masks of files to be processed from config:
    fileMasks = config.getPropertyValues(...);
    Update fileMasks if "-i" switch is provided.
    fileMasksStr = commandLineUtility.getSwitch(...).getValue();
    Parse fileMasksStr to fileMasks (use semicolon as a delimiter).
26. ignoreErrors = <"-ignore_errors" switch is provided>;
27. resultDependencies = new ArrayList<DependenciesEntry>();
28. For each fileMask from fileMasks do:
28.1. Get the list of available files by file mask:
    filesToProcess = findFiles(fileMask); // log the result if logging is required
28.2. For each fileToProcess from filesToProcess do:
28.2.1. If extractor.isSupportedFileType(fileToProcess) then
28.2.1.1. entry = extractor.extract(fileToProcess);
28.2.1.2. If exception is thrown, catch it, log and ignore. If not ignoreErrors then print out an error message and return.
28.2.1.3. resultDependencies.add(entry);
29. persistence.save(resultDependencies);
30. Print out info about extraction result.

1.3.6 Locating the input files for the standalone application

This algorithm corresponds to the method ComponentDependencyExtractorUtility #findFiles(). It uses the given file path mask to locate files on the local file disk.

Required steps for this algorithm:

1. Replace all '\' chars in filePathMask to '/'.
2. result = new ArrayList<String>();
3. If filePathMask doesn't contain '*' then
3.1. result.add(filePathMask);
3.2. Return.
4. Parse filePathMask as "part_1/part_2/part3/.../last_part" (only one part can be present). Save result to partNum:int and parts:List<String>. If last_part is empty then return.
5. Container for currently matched folders:
    pathsToCheck = new ArrayList<String>();
6. pathsToCheck.add("/") if filePathMask strats with “/” or pathsToCheck.add(“.”)
7. For i = 0..partNum-1 do:
7.1. newPaths = new ArrayList<String>();
7.2. curPart = parts,get(i);
7.3. If curPart doesn't contain "*" then
7.3.1. For each pathToCheck from pathsToCheck do:
7.3.1.1. file = new File(pathToCheck+"/"+curPart);
7.3.1.2. If file.exists() then newPaths.add(pathToCheck+"/"+curPart);
7.4. Else
7.4.1. Convert curPart to regExpPattern:String. For this replace "*" with ".*" and escape all other command characters of regular expressions.
7.4.2. Pattern pattern = Pattern.compile(regExpPattern);
7.4.3. For each pathToCheck from pathsToCheck do:
7.4.3.1. file = new File(pathToCheck);
7.4.3.2. If file.isDirectory() then
7.4.3.2.1. subfiles = file.list();
7.4.3.2.2. For each subfile from subfiles do:
7.4.3.2.2.1. Matcher m = p.matcher(subfile);
7.4.3.2.2.2. If m.matches() then newPaths.add(pathToCheck+"/"+subfile);
7.5. pathsToCheck = newPaths;
8. For each pathToCheck from pathsToCheck do:

8.1. file = new File(pathToCheck);
8.2. If file.isFile() then result.add(pathToCheck);
9. Return result.

1.3.7 Logging

The logging is used in ComponentDependencyExtractorUtility and all provided BaseDependenciesEntryExtractor subclasses. Logging is not required if the user specifies "-nolog" switch in command line of standalone application or doesn't provide Log instance to BaseDependenciesEntryExtractor subclasses.

The following events must be logged at the specified level in the mentioned method:

- INFO level
  - Start to process a component – is logged in extract() method of DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor.
  - Extracted a dependency – is logged in processDependencyPath() method of DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor.
  - End of processing a component – is logged in extract() method of DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor.
- WARN level
  - The extracted information is not well-formed – is logged in processXXX() and parseXXX() methods of DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor. E.g. when required element attributes are missing.
  - Cannot find expected build file from an archive file or other error that occurs while retrieving build file – is logged in processFile() method of DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor.
- ERROR level
  - Exception caught – any exception that is caught in ComponentDependencyExtractorUtility, DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor and is not ignored must be properly logged;
  - Other fatal error during extraction – can occur in DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor when the specified build/distribution file is not found or build files don't have target component name specified.
- DEBUG level
  - Any useful debug information at developer's discretion.

Logged records if possible must contain information about target component name, version, dependency component name and version.

1.4 Component Class Overview

Component

This class is a container for information that can identify some component. Currently all components are identified with name, version and programming language.

ComponentDependency

This class is a container for information about component dependency. It extends Component class to hold the dependency component information. Additionally it holds information about dependency component path, dependency type and category.

DependenciesEntry

This class represents the dependencies for a single component. It holds the information about the component and all its dependencies.

DependencyCategory [enum]

The enumeration for component dependency categories. Currently only two categories are provided: compile and test. It is used in ComponentDependency class.

DependencyType [enum]

The enumeration for component dependency types. Currently only two types are provided: internal and external. It is used in ComponentDependency class.

ComponentLanguage [enum]

The enumeration of component programming languages. Current only two languages are supported: Java and .NET(C#). It is used in Component class.

DependenciesEntryPersistence [interface]

This interface must be implemented by all classes that provide DependenciesEntry persistence functionality. It provides two methods that can be used to save the list of dependency entries to persistence and load the list of entries from it. Implementations of this interface must provide a constructor that accepts ConfigurationObject as the only argument to be compatible with ComponentDependencyExtractorUtility and Component Dependency Report Generator component. This constructor can throw ComponentDependencyConfigurationException.

DefaultXmlDependenciesEntryPersistence

This is a default implementation of DependenciesEntryPersistence. It loads (saves) dependency entries from (to) XML files. Indentation is used while formatting the XML file, so it can be easily read by the user.

BinaryFileDependenciesEntryPersistence

This implementation of DependenciesEntryPersistence loads (saves) dependency entries from (to) binary files. Unlike XML format, binary format has very small overhead, thus size of binary dependency files are 2-3 times less that size of XML files with the same amount of information. Also binary files can be loaded and saved much faster that XML files (generally, more than 10 times faster). Thus this type of persistence is preferred for large dependency lists.

DependenciesEntryExtractor [interface]

This interface must be implemented by all classes that can extract DependenciesEntry instance with information about the target component and all its dependencies from any source file. This interface also provides a method isSupportedFileType() that can be used for checking whether some extractor can process some file.

BaseDependenciesEntryExtractor [abstract]

This abstract class extends DependenciesEntryExtractor interface. This class must be extended by extractors that can extract dependencies of specific type and category. BaseDependenciesEntryExtractor holds sets of dependency types and categories that must be extracted. It also provided setters and getters for them. Getters return shallow copies of sets, thus subclasses should better access them as protected attributes.

DotNetDependenciesEntryExtractor

This subclass of BaseDependenciesEntryExtractor can extract dependencies from build or distribution files of .NET components from TopCoder software catalog. If accepts default.build and *.zip files in extract() methods. Please see section 1.3.2 to understand how dependency information is extracted. Logging is used in this class if the user provides a Log instance in the constructor.

JavaDependenciesEntryExtractor

This subclass of BaseDependenciesEntryExtractor can extract dependencies from build or distribution files of Java components from TopCoder software catalog. If accepts build.xml and *.jar files in extract() methods. Please see section 1.3.1 to understand how dependency information is extracted. Logging is used in this class if the user provides a Log instance in the constructor.

BuildFileParsingHelper

This is a package helper class that can be used by DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor when parsing build files. Developers can put more helper methods to this class.

MultipleFormatDependenciesEntryExtractor

This implementation of DependenciesEntryExtractor supports multiple source file formats while extracting dependencies from them. This class just holds the list of other BaseDependenciesEntryExtractor subclasses. When the user calls extract(), it finds an extractor that can process the given file (with use of isSupportedFileType() method) and passes user's call to it. Logging is used in this class if the user provides a Log instance in the constructor. By default, this class holds DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor instances, i.e. can process both Java and .NET component dependencies.

PathList

This is a container for path list information. It holds references to another path list collections and stores the own list of file paths. This class is used by DotNetDependenciesEntryExtractor to store the parsed content of <references>, <fileset>, <assemblyfileset> elements and JavaDependenciesEntryExtractor - <path> and <classpath> elements. This class is not public, thus it returns the stored collections directly, but not their shallow copies.

BuildFileProvider [interface]

This interface must be implemented by classes that can provide build files to DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor and other possible implementations of DependenciesEntryExtractor from various source. Each provider must hold the virtual path of the main build file and should be able to retrieve this file or other file by its relative path. This interface allows implementations of DependenciesEntryExtractor to provide the same algorithm for extracting dependencies from both build files and distribution archive files.

LocalBuildFileProvider

This implementation of BuildFileProvider retrieves relative build files from the local file system. This provider can be used when the main build file and its dependency files are stored in usual file system on the local disk.

ZipBuildFileProvider

This implementation of BuildFileProvider retrieves relative build files from ZIP-archive. It is used by DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor when dependencies are extracted from a ZIP/JAR component distribution file.

ComponentDependencyExtractorUtility

This is the main class of standalone application that can be used for extracting component dependencies with use of command line. It reads all configuration data from the configuration file. Some of this data can be overridden by the user with use of command line switches and arguments. By default, this utility uses MultipleFormatDependenciesEntryExtractor to extract dependencies from the specified files and DefaultXmlDependenciesEntryPersistence to save the extracted dependencies in the file. But the user can specify another DependenciesEntryExtractor and DependenciesEntryPersistence implementations to be used and name of the file where dependencies must be written to. The user can specify the list of input files for the extractor with use of file path masks. These masks can contain wildcard characters in folder and file names (e.g. "tcs/*/*/dist/*.zip"). Please see section 3.2.3 for the full list of configuration parameters and command line switches supported by this component.

1.5 Component Exception Definitions

ComponentDependencyException

This is a base class for all other custom exceptions defined in this component. This exception is never thrown directly; instead its subclasses are used.

DependenciesEntryExtractionException

This exception is thrown by implementations of DependenciesEntryExtractor when fatal error occurred while extracting dependencies from a file (e.g. when the specified file doesn't exist, has invalid or corrupted format, or when target component name cannot be detected).

DependenciesEntryPersistenceException
This exception is thrown by DependenciesEntryPersistence implementations when error occurs while loading dependency entries from persistence or saving them to persistence.

ComponentDependencyConfigurationException
This exception can be thrown by implementations of DependenciesEntryPersistence when error occurs while reading the configuration data (e.g. when required property is missing or has invalid format).

BuildFileProvisionException
This exception is thrown by implementations of BuildFileProvider when accessing the main or relative build file (e.g. when requested file doesn't exist). DotNetDependenciesEntryExtractor and JavaDependenciesEntryExtractor can ignore this error if it's not fatal.

1.6 Thread Safety

This component is not thread safe. When using it programmatically, the user must synchronize calls to all methods of classes defined in this component or use each class instance only from a single thread. Standalone application defined in this component is thread safe because it uses a single thread. But the user must be careful when using multiple instances of this application. They should not use the same persistence file.

2. Environment Requirements

2.1 Environment

Development language: Java 1.5
Compile target: Java 1.5 & 6
QA Environment: Solaris 7, RedHat Linux 7.1, Windows 2000, Windows 2003

2.2 TopCoder Software Components

Command Line Utility 1.0 – is used by standalone application to parse command line arguments.

Logging Wrapper 2.0 – is used by extractors to log errors and debug information.

Configuration API 1.0 – is used to configure classes of this component.

Configuration Persistence 1.0.1 – is used by standalone utility to load configuration from file.

Base Exception 2.0 – is used by custom exceptions of this component.

Object Factory 2.0.1 – is used by standalone application to create log factory.

Object Factory Configuration API Plugin 1.0 – is used by standalone application to create log factory.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.dependency – the main package.
com.topcoder.util.dependency.persistence – implementations of DependenciesEntryPersistence.
com.topcoder.util.dependency.extractor – implementations of DependenciesEntryExtractor.
com.topcoder.util.dependency.extractor.utility – main class of standalone extractor application.
com.topcoder.util.dependency.extractor.fileprovider – build file provider implementations.

3.2 Configuration Parameters

3.2.1 Configuration of DefaultXmlDependenciesEntryPersistence and BinaryFileDependenciesEntryPersistence

Current configuration format for both classes is the same. The following table describes the structure of ConfigurationObject passed to the constructor of DefaultXmlDependenciesEntryPersistence or BinaryFileDependenciesEntryPersistence.

Parameter	Description	Values
file_name	The name of the file where the list of dependency entries is (can be) stored.	String. Not empty. Required.

3.2.2 Configuration file of ComponentDependencyExtractorUtility

Configuration of ComponentDependencyExtractorUtility consists of two parts: data stored in the configuration file and command line parameters. The full configuration can be stored in configuration file only (i.e. utility can successfully work with no command line arguments specified), thus command line parameters are used just to override or complement the parameters from the configuration file. This section describes the configuration file parameters. See section 3.2.2 for the command line parameters.

The following table describes the structure of ConfigurationObject that is retrieved with use of Configuration Persistence from the configuration file of the standalone utility. Note that configuration file should have a <Config> element with name “com.topcoder.util.dependency.extractor.utility. ComponentDependencyExtractorUtility”. Its properties are described below:

Parameter	Description	Values
object_factory_config_name	The name of the section that holds Object Factory configuration. Default is “object_factory_config”.	String. Not empty. Optional.
object_factory_config	The actual name of this section is specified with object_factory_config_name attribute. This section contains Object Factory configuration for this application.	ConfigurationObject. Required if OF config name or log factory key attribute is specified.
persistence_class	The full class name of the DependenciesEntryPersistence implementation that is used by the utility when saving the extracted dependencies list. Can be specified or overridden in the command line. This attribute must not be specified together with “persistence_key”. Default is “com.topcoder.util.dependency.persistence. DefaultXmlDependenciesEntryPersistence”.	String. Not empty. Optional.
persistence_config	The configuration for the used DependenciesEntryPersistence implementation. Some attributes of this object can be specified or overridden in the command line.	ConfigurationObject. Required.
log_factory_key	The Object Factory key that is used to create LogFactory instance. If not specified default log factory from LogManager is used.	String. Not empty. Optional.
logger_name	The name that is passed to LogManager to create a Log instance.	String. Not empty. Required.

extractor_class	The full class name of the DependenciesEntryExtractor implementation that is used by the utility when saving the extracted dependencies list. Can be specified or overridden in the command line. Default is "com.topcoder.util.dependency.extractor.MultipleFormatDependenciesEntryExtractor".	String. Not empty. Optional.
dependency_types	The list of dependency types those must be extracted from build/distribution files. The semicolon-separated list of case-insensitive DependencyType values. This parameter is used only if extractor is of BaseDependenciesEntryExtractor type. Default is "interna;,external".	String. Not empty. Optional.
dependency_categories	The list of dependency categories those must be extracted from build/distribution files. The semicolon-separated list of case-insensitive DependencyCategory values. This parameter is used only if extractor is of BaseDependenciesEntryExtractor type. Default is "compile;test".	String. Not empty. Optional.
file_masks	The semincolon separated string list of file path masks. These masks are used to locate build/distribution files. Then component dependencies are extracted from the located files. This parameter accepts wildcard characters in folder and file names. Wildcard character ('*') matches any file path substring that doesn't contain path separators ('/' and '\'). E.g. "./tcs/*/dist/*.jar",;"C:\TCS\dot_net_\object_factory*\dist\object_factory-*.zip",;"./*/build.xml".	String. Not empty. Optional.

3.2.3 *ComponentDependencyExtractorUtility command line parameters*

The following table describes the command line switches and arguments those are supported by ComponentDependencyExtractorUtility standalone application.

Switch and arguments	**Description**
-c <file_name>	Optional. Provides the name of the configuration file for this utility. This file is read with use of Configuration Persistence component. The structure of this file is described in the section 3.2.2. Default is "config.xml".
-pclass <class_name>	Optional. The full class name of the DependenciesEntryPersistence implementation to be used. This switch specifies or overrides the parameter "persistence_class" from the section 3.2.2. This switch must not be specified together with "-f" switch. DefaultXmlDependenciesEntryPersistence is a default persistence implementation.
-o <file_name>	Optional. The output file name for the list of extracted dependency entries. Can override the parameter "file_name" mentioned in the section 3.2.1. In this case persistence configuration must be specified in section "persistence_config" mentioned in the section 3.2.2.

-f <format_name>	Optional. The name of the output file format to be used. Currently the following values are supported: “xml” – represents DefaultXmlDependenciesEntryPersistence and “binary” – represents BinaryFileDependenciesEntryPersistence. This switch specifies or overrides the parameter “persistence_class” from the section 3.2.2. This switch must not be specified together with “-pclass” switch. It duplicates functionality of “-pclass”, but is much more easy-to-use. Default format is “xml”.
-eclass <class_name>	Optional. The full class name of the DependencyEntryExtractor implementation to be used. This switch specifies or overrides the parameter “extractor_class” from the section 3.2.2. MultipleFormatDependenciesEntryExtractor is a default extractor implementation.
-dtype <types_list>	Optional. The list of dependency types those must be extracted from build/distribution files. The semicolon-separated list of case-insensitive DependencyType values. This switch can override the “dependency_types” parameter mentioned in the section 3.2.2. Default is “internal;external”.
-dcat <categories_list>	Optional. The list of dependency categories those must be extracted from build/distribution files. The semicolon-separated list of case-insensitive DependencyCategory values. This switch can override the “dependency_categories” parameter mentioned in the section 3.2.2. Default is “compile;test”.
-i <file_masks>	Optional. The semicolon-delimited list of file path masks. These masks are used to locate build/distribution files. This parameter accepts wildcard characters in folder and file names. Wildcard character (‘*’) matches any file path substring that doesn’t contain path separators (‘/’ and ‘\’). This switch can override the “input_path” parameter mentioned in the section 3.2.2. E.g. “./tcs/*/dist/*.jar; ./tcs/*/dist/*.zip”,;“C:\TCS\dot_net_\ object_factory*\dist\object_factory-*.zip”;“./*/build.xml;./*/default.build”.
-nolog	Indicates that logging should not be used. If this switch is not provided, logging is used only if logger name is specified in the configuration file (see the section 3.2.2).
-ignore_errors	Indicates that the utility should not terminate when an error occurs. Instead application tries to extract dependencies from other files.
-help -? -h	When one of the specified switches is provided, the application prints out the usage string to the standard output and terminates immediately.

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Please see the demo.

4.3 Demo

4.3.1 API usage

```
// Create dependency entry manually
// Component A depends on component B
Component componentA = new Component("A", "1.0", ComponentLanguage.JAVA);
ComponentDependency componentB = new ComponentDependency("B", "1.0", ComponentLanguage.JAVA,
    DependencyCategory.COMPILE, DependencyType.INTERNAL, "b.jar");
List<ComponentDependency> dependencyList = new ArrayList<ComponentDependency>();
dependencyList.add(componentB);
DependenciesEntry entry = new DependenciesEntry(componentA, dependencyList);
List<DependenciesEntry> entryList = new ArrayList<DependenciesEntry>();
entryList.add(entry);

// Save dependencies to XML file
DependenciesEntryPersistence persistence =
    new DefaultXmlDependenciesEntryPersistence(DEMO_DIR+File.separator+"dependencies.xml");
persistence.save(entryList);
// Load dependencies from binary file
persistence = new
BinaryFileDependenciesEntryPersistence(DEMO_DIR+File.separator+"dependencies.dat");
persistence.save(entryList);
entryList = persistence.load();

// Extract compile dependencies for Java component
BaseDependenciesEntryExtractor extractor = new JavaDependenciesEntryExtractor(null);
Set<DependencyCategory> categories = new HashSet<DependencyCategory>();
categories.add(DependencyCategory.COMPILE);
extractor.setExtractedCategories(categories);
// Extract from build.xml
entry = extractor.extract(DEMO_DIR+File.separator+"file_upload/build.xml");
// Extract from distribution file
entry = extractor.extract(DEMO_DIR+File.separator +"file_upload-2.2.0.jar");

// Extract external component dependencies for .NET component
extractor = new DotNetDenpendenciesEntryExtractor(null);
Set<DependencyType> types = new HashSet<DependencyType>();
types.add(DependencyType.EXTERNAL);
extractor.setExtractedTypes(types);
// Extract from default.build
entry = extractor.extract(DEMO_DIR+File.separator +"object_factory/default.build");
// Extract from distribution file
entry = extractor.extract(DEMO_DIR+File.separator + "object_factory-1.2.1.zip");

// Extract Java and .NET dependencies with a single multi-format extractor
extractor = new MultipleFormatDependenciesEntryExtractor(null);
// Get source file name
String fileName =DEMO_DIR+File.separator+"build.xml";
// Check whether file format is supported
boolean isSupported = extractor.isSupportedFileType(fileName);
// Extract dependencies
if (isSupported) {
    entry = extractor.extract(fileName);
}
```

4.3.2 Sample configuration file

```
<?xml version="1.0"?>
<CMConfig>
        <Config name="com.topcoder.util.dependency.extractor.utility.ComponentDependencyExtractorUtility">
                <property name="object_factory_config_name">
                        <value>object_factory_config</value>
                </property>
                <property name="persistence_class">

        <value>com.topcoder.util.dependency.persistence.DefaultXmlDependenciesEntryPersistence</value>
                </property>
                <property name="log_factory_key">
                        <value>lf</value>
```

```
                </property>
                <property name="logger_name">
                        <value>logger</value>
                </property>
                <property name="extractor_class">

        <value>com.topcoder.util.dependency.extractor.MultipleFormatDependenciesEntryExtractor</value>
                </property>
                <property name="dependency_types">
                        <value>internal;external</value>
                </property>
                <property name="dependency_categories">
                        <value>compile;test</value>
                </property>
                <property name="file_masks">
                        <value>test_files/scripts/*</value>
                </property>
                <property name="object_factory_config">
                        <property name="lf">
                                <property name="type">
                                        <value>com.topcoder.util.log.basic.BasicLogFactory</value>
                                </property>
                        </property>
        </property>
                <property name="persistence_config">
                    <property name="file_name">
                        <value>test_files/report1.xml</value>
                    </property>
        </property>
    </Config>
</CMConfig>
```

4.3.3 *Sample XML persistence file*

The following XML file contains information about Java Logging Wrapper 2.0.0 and .NET Command Line Executor 1.0. Please see distribution files from TCS catalog for dependency extraction source information.

```
<?xml version="1.0"?>
<components>
  <component name="logging_wrapper" version="2.0.0" language="java">
    <dependency type="internal" category="compile" name="base_exception" version="2.0.0"
path="../tcs/lib/tcs/base_exception/2.0.0/base_exception.jar" />
    <dependency type="internal" category="compile" name="typesafe_enum" version="1.1.0"
path="../tcs/lib/tcs/typesafe_enum/1.1.0/typesafe_enum.jar" />
    <dependency type="internal" category="compile" name="object_formatter" version="1.0.0"
path="../tcs/lib/tcs/object_formatter/1.0.0/object_formatter.jar" />
    <dependency type="external" category="compile" name="log4j" version="1.2.14"
path="../tcs/lib/third_party/log4j/1.2.14/log4j-1.2.14.jar" />
    <dependency type="external" category="compile" name="junit" version="3.8.2"
path="../tcs/lib/third_party/junit/3.8.2/junit.jar" />
    <dependency type="internal" category="test" name="base_exception" version="2.0.0"
path="../tcs/lib/tcs/base_exception/2.0.0/base_exception.jar" />
    <dependency type="internal" category="test" name="typesafe_enum" version="1.1.0"
path="../tcs/lib/tcs/typesafe_enum/1.1.0/typesafe_enum.jar" />
    <dependency type="internal" category="test" name="object_formatter" version="1.0.0"
path="../tcs/lib/tcs/object_formatter/1.0.0/object_formatter.jar" />
    <dependency type="external" category="test" name="log4j" version="1.2.14"
path="../tcs/lib/third_party/log4j/1.2.14/log4j-1.2.14.jar" />
    <dependency type="external" category="test" name="junit" version="3.8.2"
path="../tcs/lib/third_party/junit/3.8.2/junit.jar" />
  </component>
  <component name="command_line_executor" version="1.0" language="dot_net">
    <dependency type="external" category="test" name="NUnit" version="2.1" path="C:\Program
Files\NUnit\2.1\bin\nunit.framework.dll" />
  </component>
</components>
```

4.3.4 *Command line utility usage*

This command line can be used to print out the usage string:

```
java ComponentDependencyExtractorUtility -help
```

If all required configuration for the utility is stored in file config.xml, then the application can be executed without additional arguments:

```
java ComponentDependencyExtractorUtility
```

To use the custom configuration file the user can use “-c” switch:

```
java ComponentDependencyExtractorUtility -c custom_config.xml
```

The user can specify the input file mask and output file names:

```
java ComponentDependencyExtractorUtility -i build.xml –o dependencies.xml
```

Multiple input file masks with wildcards can be specified:

```
java ComponentDependencyExtractorUtility -i */*/dist/*.zip;*/*dist/*.jar
```

The user can specify the custom persistence options:

```
java ComponentDependencyExtractorUtility –pclass myPackage.CustomDependenciesEntryPersistence
–o custom.dat
```

Extractor implementation class can be specified as command line argument:

```
java ComponentDependencyExtractorUtility –eclass myPackage.CustomDependencyExtractor
```

For easiness “-f” switch can be used to specify one of XML and binary formats:

```
java ComponentDependencyExtractorUtility –f binary –o dependencies.dat
```

To specify that only external component dependencies must be extracted the user can use “-dtype” switch:

```
java ComponentDependencyExtractorUtility –dtype external
```

To specify that only test dependencies must be extracted the user can use “-dcat” switch:

```
java ComponentDependencyExtractorUtility –dcat test
```

To specify that logging must not be used the user should use “-nolog” switch:

```
java ComponentDependencyExtractorUtility –nolog
```

To specify that application should not terminate immediately when error occurs and should process other source files instead, the user should use “-ignore_errors” switch:

```
java ComponentDependencyExtractorUtility –ignore_errors
```

Usuall the command line will look like the following one:

```
java ComponentDependencyExtractorUtility -c custom_config.xml –f binary –i *.zip;*.jar –o
dependencies.xml –ignore_errors
```

4.3.5 *Extracted dependencies samples*

The following table contains extracted dependencies for some components from TC software catalog. Please download BuildScript.zip from Design Distribution Documents thread on the forum to see build files that are used as input information. Dependency path is omitted. All dependency types and categories are shown. Column “category” contains “both” if both compile and test dependencies are present for the specified dependency component. Note that some categories are not correct, but it’s not a fault of this component, but mistakes in build files.

Language	Target component, version	Dependencies		
		Component, version	Type	Category
Java	alert_factory 1.0	junit 3.8.2	external	both
		configuration_manager 2.1.4	internal	both
		base_exception 1.0	internal	both
		id_generator 3.0	internal	both
		document_generator 2.0	internal	both
		message_board 1.0	internal	both
		object_factory 2.0	internal	both

		db_connection_factory 1.0	internal	both
		command_line_utility 1.0	internal	both
		mysql 3.1.12	external	both
		xerces 1.4.4	external	both
Java	csv_conversion 1.0.0	base_exception 2.0.0	internal	both
		configuration_api 1.0.0	internal	both
		file_conversion_framework 1.0.1	internal	both
		object_factory 2.1.0	internal	both
		object_factory_configuration_api_plugin 1.0	internal	both
		result_set_collection 1.0	internal	both
		junit 3.8.2	external	both
		mysql 3.1.8a	external	both
.NET	auditor 1.0	logging_wrapper 1.0	internal	both
		configuration_manager 1.1	internal	both
		connection_factory 1.0	internal	both
		user_profile 1.0	internal	both
		nunit.framework	external	test
.NET	self-documenting_exception 2.0.0	log4net 1.2.9	external	both
		configuration_manager 2.0.1	internal	both
		logging_wrapper 2.0.1	internal	both
		exception_manager 1.0.1	internal	both
		set_utility 1.0	internal	both
		object_factory 1.2.1	internal	both
		configuration_api 1.0	internal	both
		file_based_configuration 1.0.1	internal	both
		simple_xsl_transformer 1.0	internal	both
		configuration_persistence_manager 1.0	internal	test
		application_based_configuration 1.0	internal	test
		nunit.framework	external	test

5. Future Enhancements

Support more formats of build file.
Support more languages (C++, VB, etc.).
Support different build tools and so build file formats.