JEE - Analysis of pure Java applications

Summary: this page provides information about how the JEE Analyzer handles pure Java applications

Analysis definition

In the context of a pure Java analysis an analysis is defined in terms of:

- a set of primary files
- a set of classpath entries

CAST fully analyses all primary files and uses classpath entries to resolve external references. External files are only analyzed on demand and only objects that are referenced in primary files are retrieved. In others words, primary objects (with all their incoming and outgoing references) are always available independently of their usage, whilst external objects are stored only when used and with only incoming but no outgoing references. For instance if an external class contains several methods and only one is used within the application, the remaining methods will not be stored. External objects can be identified in the viewer by their grey shading.

An external class can only be found under a classpath entry if the commonly adopted file deployment convention is respected where package hierarchy maps to directory structure and where file names identify classes. This restriction obliviously does not apply to primary files.

Very large applications (tens of thousands of files) that cannot be handled in a single analysis can be split into several smaller ones and analyzed independently of each other. This break down can almost be done arbitrarily and will not lead to any missing or incorrect information (see more in the **Obje ct Sharing** section below). When specifying each "sub-analysis", simply ensure that the whole application is accessible via classpath.

CAST allows each analysis to use their own "core API libraries" and cannot run if system classes (i.e. "java lang.*") are not specified in the Classpaths field - see **JEE - Analysis configuration**.

Supported file types

The JEE Analyzer supports java source files (.java), class files (.class) and SQLJ files (.sqlj).

Class files are reversed (or "decompiled") before being processed by the analyzer. By design, only stub files are then regenerated to extract object hierarchy. However, method implementations and variable initializers are not reversed.

Java archives (.jar and .zip) cannot be specified as such as analysis files. To include zipped files, the archive that contains them must be unzipped first. On the other hand archives are supported as class path entries. However, for archive processing, software requires the "java" executable program (delivered with any SDK or JDK) to be installed and accessible via the "Path" system variable on the computer where the analysis is run. Note that if the actual implementation source code for a given itemis provided in a Java archive, then the analyzer will handle it as external code and will only be able to decompile the objects as signatures.

Object sharing

All objects revealed by the analyzer are not only identified by their Java qualified name (including mangling to distinguish overloaded methods) but also by the full path of their corresponding definition file. This allows the reflection of dynamic class loading paradigm and distinguishes between several implementations of the same class.

However objects are not identified by the analysis in which they are retrieved. Hence if a file is processed in the context of distinct analyses the same set of objects will be shared between those analyses. This allows the creation of consistent results when processing an application in several analyses.

Each analysis keeps for each file an image corresponding to a snapshot at the time when file was last processed within the current analysis. Consequently if a shared file is modified it needs to be analyzed in the context of each and every analysis that includes the shared file. All non-resynchronized analyses will otherwise reflect the file as it was before modification.

Run Time Prediction (Inference Engine)

The J2EE Analyzer uses an inference engine to compute run time type information in order to simulate program behaviour during its execution. To find out more about activating and deactivating it in the CAST Management Studio, see **JEE - Analysis configuration**. This information is used to address the two following issues:

- Resolution of virtual calls
- Evaluation of character strings

Virtual call resolution

Method calls are de-virtualized by computing the actual run time type of the object on which invocation is performed. Software navigates from instantiation sites to method calls using executable paths. In case of alternatives both branches are considered so that all run time possibilities are retained.

Example:

```
interface I { void foo(); }
class A implements I { void foo() {...} }
class B implements I { void foo() {...} }
class T {
        void doSmth(int i) {
                I anObject = null;
                if (i > 0)
                        anObject = new A();
                else
                        anObject = new B();
                anObject.foo();
        }
```

On anObject.foo() software will trace a call towards I.foo() by compile time type analysis but will also trace dynamic calls towards A.foo() and B.foo() using run time type information.

Strings evaluation

String evaluation is used within the **Parametrization** feature (see the **JEE - Environment Profiles**) to compute the actual value of arguments of parametrized methods. Here also all possibilities are evaluated and all possible links are traced accordingly.

Example:

Lets suppose that T.execSQL(java.lang.String) is parametrized to recognize its parameter as a server object.

```
    --> class T {
         void execSQL(String s) { ... }
         void doSelect(String s) {
                 String req = "select * from My";
                 execSQL(req + s)
         }
         void doSmth() {
                 doSelect("Table");
         }
 }
```

On doSelect("Table") software will trace a "Use Select" link towards table MyTable if any.

Applet and SQJ support

Applet

Whenever an applet is detected (either via an HTML <APPLET> tag or via a class extending java.applet.Applet) an applet component is created in the Analysis Service. Applets are identified by the full path of their corresponding class file. The **Prototype** link is traced between the applet and its applet class.

SQLJ Support

The JEE Analyzer supports the **SQLJ extension**. It does not preprocess code to its JDBC equivalent as the SQLJ translator does, instead, it enriches underlying roadmaps with objects implicitly defined by SQLJ declarations.

The following SQLJ language elements are supported:

- SQLJ declaration: iterator (named and positional) and context declaration both with "implements" and "with" clauses
- SQLJ executable statement: statement or assignment clause

SQLJ Declarations

Implicit Inheritance is recreated as follows on each declaration:

	Implicit super-class	**Implicit super-interface**
Positional iterator	sqlj.runtime.ref.ResultSetIterImpl	sqlj.runtime.PositionedIterator
Named iterator	sqlj.runtime.ref.ResultSetIterImpl	sqlj.runtime.NamedIterator
Context	sqlj.runtime.ref.ConnectionContextImpl	sqlj.runtime.ConnectionContext

For all declarations that use the "with" clause, a class attribute is created on each "with" constant. The type of this attribute is given by the resolved type of its initializer.

The following **implicit constructors** are defined for an **iterator declaration**:

- public <*iterator name*> (sqlj.runtime.profile.RTResultSet)

For each column of an **iterator declaration** an **implicit accessor** is defined as follows:

- named iterator: public <*column type*> get<*column name*> ( )
- unnamed iterator: public <*column type*> getCol<*column index*> ( )

The following **implicit methods** are defined for a **context declaration**:

- public static <*context name*> getDefaultContext ( )
- public static void setDefaultContext (<*context name*>)
- public static sqlj.runtime.profile.Profile getProfile (java.lang.Object)
- public static java.lang.Object getProfileKey (sqlj.runtime.profile.Loader , java.lang.String)

The following **implicit constructors** are defined for a **context declaration**:

- public <*context name*> (java.sql.Connection)
- public <*context name*> (sqlj.runtime.ConnectionContext)
- public <*context name*> (java.lang.String , boolean)
- public <*context name*> (java.lang.String , java.util.Properties , boolean)
- public <*context name*> (java.lang.String , java.lang.String , java.lang.String , boolean)

SQLJ Statements

While processing **SQLJ clauses**, Java **host expressions**, **context expressions** and **result expressions** are passed to the analyzer as normal Java code. At the same time, the embedded SQL code is scanned for server side object references using dynamic link resolution algorithms as explained in the corresponding section.