Advanced J2EE instrumentation

This section contains the following topics:

About using custom instrumentation

To target your application with fine granularity, you can create custom instrumentation configurations. Custom instrumentation lets you manually instrument classes that are not included in the default Precise for J2EE monitored classes. It also lets you exclude classes that are monitored by default. To use custom instrumentation, you must edit the appropriate instrumentation configuration file and verify that the edited file is uncommented in the InstrumenterConfigList.xml file, the configuration file that Precise for J2EE uses to keep track of your instrumentation configurations.

There are some minor variations to code format between using Monitoring Configuration or performing custom instrumentation manually. Nevertheless these minor variations are acceptable.

Using custom instrumentation requires knowledge of Java language constructs. Refer to an introductory Java language or programming primer for an explanation of Java language concepts.

Custom instrumentation is only recommended for very specific instrumentation needs. In other cases, it is recommended to use the Monitoring Configuration or Adaptive Instrumentation options, available in the Monitor Settings dialog box (**Settings>Monitor Settings**). For more information regarding these instrumentation options, the “About Monitoring Configuration” section in the *Precise for J2EE User’s Guide*.

About modifying instrumenter configuration files

Instrumentation instructions are contained in a series of XML files that can be found in the Precise for J2EE agent installation on the host where the monitored JVM runs. The InstrumenterConfigList.xml file, which also exists in each monitored JVM’s configuration directory, references the instrumentation configuration files. See About instrumenter configuration file reference .

How you modify your instrumentation configuration depends on what you try to do:

- To enable a single existing instrumentation configuration that was not enabled by default, you can enable this configuration by uncommenting the reference to the file in the InstrumenterConfigList.xml file.
- To change the instrumentation instructions within a specific instrumentation configuration file, for example to instrument a specific method, you would modify the Custom.xml file (or any other instrumenter configuration XML file). In addition, you must verify that this instrumentation configuration file is enabled in the InstrumenterConfigList.xml file.

  By default, the custom instrumentation configuration files are located in the instrumenter directory:

  `<i3_root>/products/i3fp/registry/products/j2ee/config/instrumenter`.

  All relative paths of referenced configuration files are relative to this directory.

You may also copy these files to JVM ID-specific directories to customize them on a per JVM basis. You can use the following special variables to specify the absolute path of these per-JVM configuration files:

- `${indepth.j2ee.home}` expands to `<i3_root>/products/j2ee`
- `${indepth.j2ee.server_id}` expands to the raw JVM ID
- `${indepth.j2ee.jvm_id}` expands to the JVM ID (with serial number)

For example, if you have copied the Custom.xml file into the `<i3_root>/products/j2ee/config/petstore` directory, you can reference it with:

```
<config-file>
  ${indepth.j2ee.home}/config/${indepth.j2ee.server_id}/Custom.xml
</config-file>
```

Enabling an additional configuration file

The following procedure describes how you can enable an additional configuration file.

To enable an additional configuration file

1. Open the InstrumenterConfigList.xml file in the JVM ID-specific configuration directory for your application server:
   `<i3_root>/products/i3fp/registry/products/j2ee/config/JVMID/`
2. In the InstrumenterConfigList.xml file, locate the XML block that defines the file you want to enable. For example, to enable the JNDI custom instrumentation file, locate the following section:
   <!--
   JNDI
   Uncomment to instrument.

```
-->
<!--
    <config-file>
    JNDI.xml
    </config-file>
-->
```

3. Remove the XML comments surrounding the file name. For example:

```
<!--
    JNDI
    Uncomment to instrument.
-->
<config-file>
    JNDI.xml
</config-file>
```

Adding your own custom instrumentation configuration file

The following procedure describes how you can add your own custom instrumentation configuration file.

To add your own custom instrumentation configuration file

1. Create a new user-defined XML configuration file, for example:
   UserDefined.xml, in the following way:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    User Defined Instrumenter Configuration File
-->
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name>
                </class-name>
                <methods>
                    <method>
                        <name>
                        </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

2. To add the file to the InstrumenterConfigList.xml file, use the following syntax:

```
<!--
    User defined instrumenter config file
-->
    <config-file>
        UserDefined.xml
    </config-file>
```

3. Modify the new XML configuration file to include the interfaces, classes, and methods you want to instrument. See the next section for more information.

About custom instrumenter configuration

Precise for J2EE allows you great flexibility in instrumenting Java applications. You can instrument specifically named packages, interfaces, classes, and methods. In addition, you can instrument all classes in a package or all methods in a class using the wildcard character. Sub-elements of <instrumenter-config> also allow you to target the behavior of methods so that you can instrument all calls to a method as well as all calls the method makes, or calls from a specific method to another specific method. Precise for J2EE instrumentation also takes advantage of various Java language properties, allowing you, for example, to instrument all classes that extend a common subclass or interface, or to instrument classes and methods marked with a runtime-visible annotation.

Using custom instrumentation requires knowledge of Java language structures. Refer to an introductory Java language or programming primer for an explanation of Java language concepts.

The Workflow Framework example is used to illustrate how instrumentation is applied to Java language constructs. The Workflow Framework example is based on the following classes and interfaces:

```
package xmp.wfm.task;

@Retention(RetentionPolicy.RUNTIME)
public @interface TransactionProvider {}

@Retention(RetentionPolicy.RUNTIME)
public @interface Transaction {}
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface RecoverableTransactionProvider {}

@Retention(RetentionPolicy.RUNTIME)
public @interface Resource {}

@TransactionProvider
public interface Task {

        @Transaction
        public void start();
        public void stop();
        public void stop(boolean force);

}

@RecoverableTransactionProvider
public interface RecoverableTask extends Task {
    public void start(RecoverableTaskContext context);
}

public abstract class AbstractTask implements Task {

        @Transaction
        public void start() { }
        protected void start(TaskContext taskContext) { }
        public void stop() { }
        public void stop(boolean force) { }

}

package xmp.wfm.server;

@Resource
public class NonRecoverableTaskAdapter extends AbstractTask {
}

@Resource
public class RecoverableTaskAdapter extends AbstractTask implements RecoverableTask {
    public void start() { }
    public void start(RecoverableTaskContext context) { }
    protected void recover(RecoverableTaskContext context) { }
    public void stop() { }
    public void stop(boolean force) { }
}
```

About instrumenting interfaces

Instrumentation can be applied to methods in abstract and concrete classes that implement specific interfaces.

About instrumenting methods in implementations of an interface

The <custom-config> element can be used to cause instrumentation to be applied to methods of abstract and concrete classes that implement an interface.

This instrumenter configuration file causes instrumentation to be applied to the start and stop methods of abstract and concrete classes that implement the Task interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
            <methods>
                <method>
                    <name> start </name>
                </method>
                <method>
                    <name> stop </name>
                </method>
            </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The matched class implements, directly or indirectly, the interface that is specified in the <class-name> element.
- The name of the method matches the name that is specified in one of the <name> elements.
- The method, including signature, was declared in the matching interface.

Based on these rules, custom-type instrumentation are applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.stop()`
- `AbstractTask.stop(boolean)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop()`
- `RecoverableTaskAdapter.stop(boolean)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.start(TaskContext)` because the method was not declared in the Task interface.
- `RecoverableTaskAdapter.start(TaskContext)` because the method was not declared in the Task interface.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method was not declared in the Task interface.

About restricting instrumentation to methods that match a signature

The <params> element can be included within a <method> element to restrict instrumentation to a method that is based on a signature. See About method signature matching.

This instrumenter configuration file causes instrumentation to be applied to the start and stop(boolean) methods of abstract and concrete classes that implement the Task interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
                <methods>
                    <method>
                        <name> start </name>
                    </method>
                    <method>
                        <name> stop </name>
                        <params>
                            <param> boolean </param>
                        </params>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The matched class implements, directly or indirectly, the interface that is specified in the <class-name> element.
- The name of the method matches the name that is specified in the <name> element.
- The method was declared in the matching interface.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.stop(boolean)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop(boolean)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.start(TaskContext)` because the method was not declared in the Task interface.
- `AbstractTask.stop()` because the method does not match the specified signature of (boolean).
- `RecoverableTaskAdapter.start(TaskContext)` because the method was not declared in the Task interface.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method was not declared in the Task interface.
- `RecoverableTaskAdapter.stop()` because the method does not match the specified signature of (boolean). See About method signature matching.

About extending instrumentation to interfaces matching a wildcard

Inheritance is not considered when wildcards are used with <class-name> element. It is not possible to use wildcards to cause instrumentation to be applied to all abstract or concrete class's that implement interface's that match a wildcard pattern.

However, wildcards can be used to apply instrumentation to all abstract or concrete classes whose names match the wildcard pattern that is specified in the <class-name> element.

See About using the wildcard character *.

About extending instrumentation to methods that match a wildcard

Wildcards can be used in the <name> element. This instrumenter configuration file causes instrumentation to be applied to the start() and stop() methods of abstract and concrete classes that implement the Task interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
                <methods>
                    <method>
                        <name> s* </name>
                        <params> <params/>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

When wildcards are used in the <name> element and the matching construct is an interface, instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The name of the method matches the wildcard pattern that is specified in the <name> element.
- The signature of the method matches the signature that is specified in the <params> element (if present).
  Specifying empty <params> </params> elements indicates that methods with zero arguments are matched, as in a signature of (). If <params> < /params> is omitted, all signatures are matched.
- The method is declared in the matched interface.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.stop()`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop()`

However, instrumentation is not applied to the following methods:

- `AbstractTask.start(TaskContext)` because the method was not declared in the Task interface.
- `AbstractTask.stop(boolean)` because the method does not match the specified signature of ().
- `RecoverableTaskAdapter.start(TaskContext)` because the method was not declared in the Task interface.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method name does not match the wildcard pattern.
- `RecoverableTaskAdapter.stop(boolean)` because the method does not match the signature of ().

About extending instrumentation to methods that are declared in extending interfaces

By default, instrumentation is restricted to methods that are declared in the matching interface. The <apply-to-subtypes> element can be used to extend instrumentation to apply to methods that are declared in extending interfaces.

This instrumenter configuration file causes instrumentation to be applied to the start methods of abstract and concrete classes that implement the Task interface or an interface that extends the Task interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
                <methods>
                    <method>
                        <name> sta* </name>
                        <apply-to-subtypes>true</apply-to-subtypes>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The name of the method matches the wildcard pattern that is specified in the <name> element.
- The signature of the method matches the signature that is specified in the <params> element (if present).
- The method is declared in the matched interface.

- The matched class has one or more interfaces are the same as or extend the interface that is specified in the <class-name> element.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(RecoverableTaskContent)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.start(TaskContext)` because the method was not declared in the Task interface or in an interface that extends the Task interface.
- `AbstractTask.stop()` because the method does not match the specified wildcard pattern of sta*.
- `AbstractTask.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method name does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop()` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.

About extending wildcards to apply to methods that are declared in implementations

By default, instrumentation is restricted to methods that are declared in the matching interface. The <apply-to-implementations> element can be used to extend instrumentation to apply to methods that are declared in abstract and concrete classes that implement the matching interface.

This instrumenter configuration file causes instrumentation to be applied to the start methods of abstract and concrete classes that implement the Task interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
                <methods>
                    <method>
                        <name> sta* </name>
                        <apply-to-implementations>true
                        </apply-to-implementations>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The name of the method matches the wildcard pattern that is specified in the <name> element.
- The signature of the method matches the signature that is specified in the <params> element (if present).
- The method is declared in a class that extends the abstract class or implements the interface that is identified in the <class-name> element.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContent)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(RecoverableTaskContent)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.stop()` because the method does not match the specified wildcard pattern of sta*.
- `AbstractTask.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method name does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop()` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.

About instrumenting classes

Instrumentation can be applied to methods in abstract and concrete classes.

About instrumenting methods in abstract and concrete classes

The <custom-config> element can be used to cause instrumentation to be applied to methods of abstract and concrete classes.

This instrumenter configuration file causes instrumentation to be applied to the start and stop methods of abstract and concrete classes that extend the AbstractTask class.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
                <methods>
                    <method>
                        <name> start </name>
                    </method>
                    <method>
                        <name> stop </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The matched class extends the class specified in the <class-name> element.
- The name of the method matches the name that is specified in one of the <name> elements.
- The method was declared in the matching class.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContent)`
- `AbstractTask.stop()`
- `AbstractTask.stop(boolean)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(TaskContent)`
- `RecoverableTaskAdapter.stop()`
- `RecoverableTaskAdapter.stop(boolean)`

However, instrumentation is not applied to the following methods:

- `RecoverableTaskAdapter.recover(RecoverableTaskContent)` because the method was not declared in the AbstractTask class

About restricting instrumentation to methods that match a signature

The <params> element can be included within a <method> element to restrict instrumentation to a method that is based on signature.

This instrumenter configuration file causes instrumentation to be applied to the start and stop(boolean) methods of abstract and concrete classes that extend the AbstractTask class.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
                <methods>
                    <method>
                        <name> start </name>
                    </method>
                    <method>
                        <name> stop </name>
                        <params>
                            <param> boolean </param>
                        </params>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The matched class extends the interface that is specified in the <class-name> element.
- The name of the method matches the name that is specified in the <name> element.
- The method signature matches the types that are specified using the <param> elements.
- The method was declared in the matching class.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`

- `AbstractTask.start(TaskContent)`
- `AbstractTask.stop(boolean)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(TaskContent)`
- `RecoverableTaskAdapter.stop(boolean)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.stop()` because the method does not match the specified signature of (boolean).
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method was not declared in the AbstractTask class.
- `RecoverableTaskAdapter.stop()` because the method does not match the specified signature of (boolean) See About method signature matching.

About extending instrumentation to classes matching a wildcard

Inheritance is not considered when wildcards are used with a <class-name> element. It is not possible to use wildcards to cause instrumentation to be applied to all abstract or concrete classes that extend classes that match a wildcard pattern.

However, wildcards can be used to apply instrumentation to all abstract or concrete classes whose names match the wildcard pattern that is specified in the <class-name> element.

About extending instrumentation to methods that match a wildcard

Wildcards can be used in the <name> element.

This instrumenter configuration file causes instrumentation to be applied to the start() and stop() methods of abstract and concrete classes that extend the AbstractTask class. See About using the wildcard character *.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
                <methods>
                    <method>
                        <name> s* </name>
                        <params> <params/>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

When wildcards are used in the <name> element and the matching construct is a class, instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The name of the method matches the wildcard pattern that is specified in the <name> element.
- If the <params> element is present, the signature of the method matches the signature that is specified in the <params> element. Specifying empty <params> </params> indicates that methods with zero arguments are matched, as in a signature of ( ). If <params> </params> is omitted, all signatures are matched.
- The method is declared in the matched class.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContent)`
- `AbstractTask.stop()`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(TaskContent)`
- `RecoverableTaskAdapter.stop()`

However, instrumentation is not applied to the following methods:

- `AbstractTask.stop(boolean)` because the method does not match the specified signature of ().
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method name does not match the wildcard pattern.
- `RecoverableTaskAdapter.stop(boolean)` because the method does not match the specified signature of ().

About extending instrumentation to methods that are declared in extending classes

By default, instrumentation is restricted to methods that are declared in the matching class. The <apply-to-subtypes> element can be used to extend instrumentation to apply to methods that are declared in extending classes.

This instrumenter configuration file causes instrumentation to be applied to the start methods of abstract and concrete classes that extend the AbstractTask class or a class that extends the AbstractTask class.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
                <methods>
                    <method>
                    <name> sta* </name>
                    <apply-to-subtypes>true</apply-to-subtypes>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that satisfy the following criteria:

- The name of the method matches the wildcard pattern that is specified in the <name> element.
- The signature of the method matches the signature that is specified in the <params> element (if present).
- The method is declared in the matched class.
- The matched class is the same as or extends the class specified in the <class-name> element.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContent)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.start(RecoverableTaskContent)`

However, instrumentation is not applied to the following methods:

- `AbstractTask.stop()` because the method does not match the specified wildcard pattern of sta*.
- `AbstractTask.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.recover(RecoverableTaskContext)` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop()` because the method does not match the specified wildcard pattern of sta*.
- `RecoverableTaskAdapter.stop(boolean)` because the method does not match the specified wildcard pattern of sta*.

About instrumenting using annotations

Instrumentation can be applied to classes and interfaces that are annotated with a specific annotation

About instrumenting annotated classes and interfaces

The <annotation-name> element can be added to the <java-class> element to cause instrumentation to be applied to classes that are annotated with a specific annotation.

This instrumenter configuration file causes instrumentation to be applied to the start and stop methods of abstract and concrete classes that extend a class or implement an interface that is annotated with the TransactionProvider annotation.

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> * </class-name>
                <annotation-name> xmp.wfm.task.TransactionProvider </annotation-name>
                <methods>
                    <method>
                        <name> start </name>
                    </method>
                    <method>
                        <name> stop </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods of abstract and concrete classes that fill the following criteria:

- The matched class extends a class that is annotated with the annotation specified in the <annotation-name> element.
- The name of the method matches the name that is specified in the <name> element.

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start`
- `AbstractTask.stop`
- `RecoverableTaskAdapter.start`
- `RecoverableTaskAdapter.stop`

About instrumenting annotated methods

The <annotation-name> element can be added to the <method> element to cause instrumentation to be applied to methods that are annotated with a specific annotation.

This instrumenter configuration file causes instrumentation to be applied to all methods in the application that are annotated with @Transaction:

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> * </class-name>
                <methods>
                    <method>
                        <name> * </name>
                        <annotation-name> xmp.wfm.task.Transaction</annotation-name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods that fill the following criteria:

- The method is annotated with `@Transaction`

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start`

About instrumenting annotated methods in implementing or inheriting classes

The <apply-to-subtypes> element can be added to the <method> element along with the <annotation-name> element, to cause classes that override or implement an annotated method to be instrumented.

This instrumenter configuration file causes instrumentation to be applied to all methods in the application that are annotated with @Transaction, and to implementing and overriding methods:

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> * </class-name>
                <methods>
                    <method>
                        <name> * </name>
                        <annotation-name> xmp.wfm.task.Transaction</annotation-name>
                        <apply-to-subtypes>true</apply-to-subtypes>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods that fill the following criteria:

- The method is annotated with `@Transaction`
- Or - the method implements an interface method that is annotated with `@Transaction`
- Or - the method overrides a method that is annotated with `@Transaction`

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start`
- `RecoverableTaskAdapter.start`

About instrumenting only classes that are not assignable to a class or interface

The < not-assignable-to> element can be added to the <java-class> element to prevent the rule from being applied to classes that are assignable to specific types.

This instrumenter configuration file causes instrumentation to be applied to all the classes in the xmp.wfm.task package, except for the ones that are assignable to RecoverableTask:

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.* </class-name>
                <not-assignable-to> xmp.wfm.task.RecoverableTask </not-assignable-to>
                <methods>
                    <method>
                        <name> start </name>
                     </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
</instrumenter-config>
```

Instrumentation is applied to methods that satisfy the following criteria:

- The class is in the package `xmp.wfm.task`
- The class is not assignable to `xmp.wfm.task.RecoverableTask`

Based on these rules, custom-type instrumentation is applied to the following methods:

- `AbstractTask.start`

About instrumenting all calls from a method

Instrumentation can be applied to all calls from methods that match criteria specified in a <java-classes> element. This instrumenter configuration file causes instrumentation to be applied to all calls from the methods that match the <java-classes> element.

```
<?xml version='1.0'?>
<instrumenter-config>
    <all-calls-from-method>
        <java-classes> ]
            ...
        </java-classes>
    </all-calls-from-method>
</instrumenter-config>
```

The rules that apply to the <java-classes> element that is documented in preceding sections are applied when it is used inside the <all-calls-from-method> element.

About instrumenting calls to methods

The <all-calls-to-method> element can be used to apply instrumentation to all calls to specific methods.

This instrumenter configuration file causes instrumentation to be applied to all calls made to printIn methods in the java.io. package or to run () methods.

```
<?xml version='1.0'?>
<instrumenter-config>
    <all-calls-to-method>
        <methods>
            <method>
                <name> java.io.*.println </name>
            </method>
            <method>
                <name> *.run </name>
                <params> <params/>
            </method>
        </methods>
    </all-calls-to-method>
</instrumenter-config>
```

The <name> element has a different interpretation from its use in all other cases. When the <name> element is used inside an <all-calls-to-method> element, it must represent a fully-qualified method name. The portion of the <name> after the last dot (“.”) is considered to be the method name, and the portion of the <name> before the last dot is considered to be the class name. When you use wildcards, it is important to use a wildcard pattern that fits the scheme described. For example, the wildcard pattern *.set* matches all methods whose name starts with set of all classes, and xmp.task.*.* matches all methods in all classes whose name starts with xmp.task.

The <invocation-relationship> element can be used instead of the <all-calls-to-method> element to restrict instrumentation to calls to specific methods.

This instrumenter configuration file causes instrumentation to be applied to all calls made to println methods in the java.io. package or to run() methods from methods in abstract or concrete classes that implement the AbstractTask interface.

```
<?xml version='1.0'?>
<instrumenter-config>
    <invocation-relationship>
        <java-class>
            <class-name> xmp.task.AbstractTask </class-name>
        </java-class>
        <invoked-method> java.io.*.println </invoked-method>
        <invoked-method> *.run() </invoked-method>
    </invocation-relationship>
</instrumenter-config>
```

The rules that apply to the <java-class> element that is documented in preceding sections are applied when it is used inside the <invocation-relationship> element.

The <invoked-method> represents a fully-qualified method name. The portion of the <invoked-method> after the last dot (“.”) is considered to be the method name, and the portion of the <invoked-method> before the last dot is considered to be the class name. When you use wildcards, it is important to use a wildcard pattern that fits the scheme described. For example, the wildcard pattern *.set* matches all methods whose name starts with set of all classes, and xmp.task.*.* matches all methods in all classes whose name starts with xmp.task.

The <invoked-method> may also include a method signature. The method signature should follow these rules:

- The method signature must be enclosed in parentheses (“(“) and (“)”).
- The method signature must not contain spaces.
- A comma (“,”) must separate parameter types in the method signature.
- The method signature may include wildcards.

About preventing instrumentation for classes, methods, and calls to methods

Instrumentation can be prevented for specific packages and sub-packages, classes, methods, or calls to methods. The <ignore-config> element in instrumenter configuration files is used to illustrate how to prevent instrumentation from being applied to specific packages and sub-packages, classes, method, or calls to methods. See About instrumenter configuration file reference .

As with the <custom-config>, <all-calls-from-method>, <all-calls-to-method>, and <calls-from-method-to-method> elements, the <ignore-config> element can be placed in its own instrumenter configuration file. The examples should be considered to imply that the <ignore-config> element must be included in an instrumenter configuration file that contains other instrumentation directives.

About preventing instrumentation for all methods of classes in a package and sub-packages

The <java-classes> element can be used within an <ignore-config> element to prevent instrumentation from being applied to all methods of classes in a package and sub-packages using wildcards in the <class-name>.

This instrumenter configuration file prevents instrumentation from being applied to the AbstractTask class:

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.* </class-name>
                <methods>
                    <method>
                        <name> * </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config> ]
    <ignore-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.* </class-name>
            </java-class>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

Based on these rules, instrumentation is not applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContext)`
- `AbstractTask.stop()`
- `AbstractTask.stop(boolean)`

However, instrumentation is still applied to the following methods:

- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop()`
- `RecoverableTaskAdapter.stop(boolean)`

About preventing instrumentation for methods of a class

The <java-classes> element can be used within an <ignore-config> element to prevent instrumentation from being applied to some or all methods of abstract or concrete classes, or to all methods of a class in a specific package and its sub-packages.

This instrumenter configuration file prevents instrumentation from being applied to the AbstractTask class:

```
<?xml version'1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
                <methods>
                    <method>
                        <name> * </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
    <ignore-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.AbstractTask </class-name>
            </java-class>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

Based on this configuration, instrumentation is not applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.start(TaskContext)`
- `AbstractTask.stop()`
- `AbstractTask.stop(boolean)`

However, instrumentation is applied to the following methods:

- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop()`
- `RecoverableTaskAdapter.stop(boolean)`

The <methods> element can be supplied to prevent instrumentation from being applied to some methods of a class, but not others.

This instrumenter configuration file prevents instrumentation from being applied to all stop() methods:

```
<?xml version'1.0'?>
<instrumenter-config>
    <custom-config>
        <java-classes>
            <java-class>
                <class-name> xmp.wfm.task.Task </class-name>
                <methods>
                    <method>
                        <name> * </name>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </custom-config>
    <ignore-config>
        <java-classes>
            <java-class>
                <class-name> * </class-name>
                <methods>
                    <method>
                        <name> stop </name>
                        <params> <params/>
                    </method>
                </methods>
            </java-class>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

Based on this configuration, instrumentation is not applied to the following methods:

- `AbstractTask.stop()`

- `RecoverableTaskAdapter.stop()`

However, instrumentation is applied to the following methods:

- `AbstractTask.start()`
- `AbstractTask.stop(boolean)`
- `RecoverableTaskAdapter.start()`
- `RecoverableTaskAdapter.stop(boolean)`

About preventing instrumentation for calls from a method

The <all-calls-to-method> element can be used within an <ignore-config> element to prevent instrumentation from being applied to all calls to specific methods.

This instrumenter configuration file prevents instrumentation from being applied to all calls to Java runtime classes (java.*) methods.

```
<?xml version'1.0'?>
<instrumenter-config>
    <ignore-config>
        <java-classes>
            <all-calls-to-method>
            <methods>
            <method>
            <name> java.* </name>
            </method>
            </methods>
            </all-calls-to-method>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

The <all-calls-to-method> element is used within an <ignore-config> element, the <name> element of <method> elements is expected to be a qualified method name. The last dot in the <name> separates the class name from the method name.

This is also true if wildcards are used. This instrumenter configuration file prevents instrumentation from being applied to all calls to methods in the java class.

```
<?xml version'1.0'?>
<instrumenter-config>
    <ignore-config>
        <java-classes/>
            <all-calls-to-method>
                <methods>
                    <method>
                        <name> java.* </name>
                    </method>
                </methods>
            </all-calls-to-method>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

About preventing instrumentation for calls to a method

The <invocation-relationship> element can be used within an <ignore-config> element to prevent instrumentation from being applied to calls to specific methods from specific calling methods.

This instrumenter configuration file prevents instrumentation from being applied to all calls to Java runtime classes (java.*) methods that take an int[] parameter from the stop() methods in the xmp.wfm package and sub-packages.

```
<?xml version='1.0'?>
<instrumenter-config>
    <ignore-config>
        <java-classes/>
            <invocation-relationship>
                <java-class>
                    <class-name> xmp.wfm.* </class-name>
                    <methods>
                        <method>
                            <name> stop </name>
                            <params> <params/>
                        </method>
                    </methods>
                </java-class>
                <invoked-method> java.*.*(int[]) </invoked-method>
            </invocation-relationship>
        </java-classes>
    </ignore-config>
</instrumenter-config>
```

About instrumenting calls to EJB business method implementations

The J2EE specification does not require that an EJB implementation implement the remote interface directly. Application servers typically generate a skeleton that implements the remote interface directly and delegates calls to the EJB implementation.

Because there is no direct relationship between an EJB implementation and the remote interface, it is difficult to instrument the business methods of an EJB implementation without first identifying the remote interfaces and then manually instrumenting the methods.

About the Calls to EJB instrumentation feature

The “Calls to EJB implementations” instrumentation feature adds a configurable extension to the instrumenter to allow calls to EJB implementations to be instrumented.

When the feature is enabled, the “Calls to EJB implementations” instrumentation feature searches for classes that match certain marker classes or interfaces or name patterns. If a class matches, caller-side instrumentation is applied to any calls from a method that makes a call to another method of the same name and signature.

The marker classes and interfaces are expected to identify the EJB skeletons. As an example, consider these application classes:

```
public interface Account extends EJBObject {
    public void deposit(float amount);
}
public class AccountEJB implements SessionBean {
    public void deposit(float amount)
        updateAccount(getCurrentBalance() + amount);
    }
}
```

Under WebLogic 7.0, EJB skeletons extend the weblogic.rmi.internal.Skeleton class. The “Calls to EJB implementations” instrumentation feature can be configured to find the skeleton and instrument the call to deposit. As an example, the WebLogic skeleton might look as follows:

```
public class AccountEJB_bfqop_WKSkel
extends Skeleton
implements Account {
    public void init() { /* ... */ }
    public void invoke() { /* ... */ }
    public void deposit(float amount) {
        impl.deposit(amount);
    }
}
```

In this case, the call from AccountEJB_bfqop_WKSkel.deposit to AccountEJB.deposit is instrumented because the call is to a method with the same name and signature as the caller.

> init() and invoke() are probably not instrumented unless they call another method with the same name and signature.

Instrumentation that is applied by the “Calls to EJB implementations” instrumentation feature show up in the instrumenter log as FIXED-CALLER-SIDE instrumentation.

The EJBImpl.xml file looks like the following example:

```
<instrumenter-config>
    <planner-config>
        <class-name>com.precise.javaperf.instrument.planner. CallsToEJBImplementationPlanner
        </class-name>
        <property>
            <name> skeletonMarkerClass </name>
            <value> weblogic.ejb20.internal.BaseEJBObject </value>
        </property>
        <property>
            <name> logBeginMethodName </name>
            <value> logBeginEjbServer </value>
        </property>
        <property>
            <name> logEndMethodName </name>
            <value> logEndEjbServer </value>
        </property>
        <property>
            <name> invocationType </name>
            <value> EJBImpl </value>
        </property>
    </planner-config>
</instrumenter-class>
```

The <class-name> element identifies the name of the new “pluggable instrumentation planner.” The following table lists the recognized properties.

Table 18-1 Recognized properties

Property name	Multiplicity	Description
logBeginMethodName	Optional	Identifies the logger method to call to report calls to EJB business methods.
logEndMethodName	Optional	Identifies the logger method to call to report calls to EJB business methods.
invocationType	Optional	Identifies how the logger events show up in Precise for J2EE.
skeletonMarkerClass	Any Number	Identifies the classes that an EJB skeleton must extend. Only classes that extend a skeleton class or implement a skeleton interface are searched for calls to methods with matching name and signature.
skeletonMarkerInterface	Any Number	Identifies the classes that an EJB skeleton must implement. Only classes that extend a skeleton class or implement a skeleton interface are searched for calls to methods with matching name and signature.
implementationMarkerClass	Any Number	Identifies the classes that must be extended by calls with a matching name and signature.
implementationMarkerInterface	Any Number	Identifies the classes that must be implemented by calls with a matching name and signature.

Applying instrumentation using the "Calls to EJB" instrumentation feature

The following procedure describes how to apply instrumentation using the "Calls to EJB" instrumentation feature.

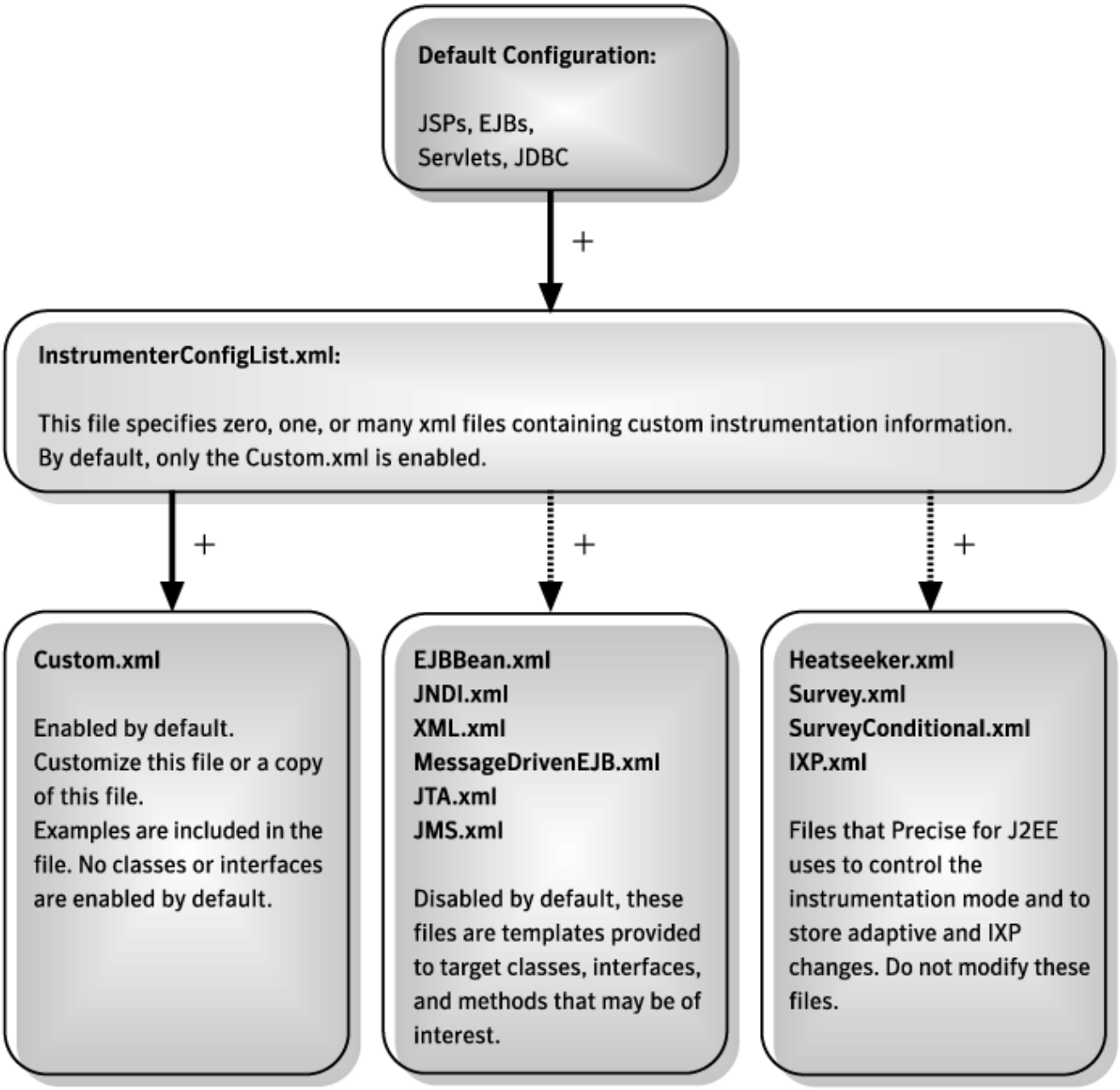
To apply instrumentation using the “Calls to EJB” instrumentation feature

1. Add the EJBImpl.xml file to the InstrumenterConfigList.xml file:
   ```
   <instrumenter-config-list>
       <config-file>
           EJBImpl.xml
       </config-file>
   </instrumenter-config-list>
   ```
2. Restart the application server.

About instrumenter configuration file reference

The purpose and schema of the various files that configure instrumentation for a monitored JVM is described in About the master configuration file, About the instrumenter configuration files, About the structure of instrumenter configuration files, and About common instrumenter configuration matching techniques.

Figure 18-1 Instrumenter Configuration



About the master configuration file

Instrumentation for each monitored JVM is configured from a master configuration file. The master configuration file, InstrumenterConfigList.xml, contains references to instrumenter configuration files that contain specific rules that are used to determine where and how to apply instrumentation to a monitored JVM.

The master configuration file can be found under the JVM-specific config directory for a monitored JVM. The master configuration file has the following structure:

```
<?xml version='1.0'?>
<instrumenter-config-list> ]
    <config-file> <!-- occurs 0 or more times -->
        instrumenter-config-file-path
    </config-file>
</instrumenter-config-list>
```

The instrumenter config file path specifies the absolute or relative path to an instrumenter configuration file. Paths are relative to `<i3_root>/products/i3fp/registry/products/j2ee/config/instrumenter` .

The following special variables can be used in the instrumenter config file path:

- `${indepth.j2ee.home} expands to  <i3_root>/products/j2ee`
- `${indepth.j2ee.server_id} expands to the JVM ID (with no sequence number)`
- `${indepth.j2ee.jvm_id} expands to the JVM ID (with a  sequence number)`

About the instrumenter configuration files

Instrumenter configuration files contain specific rules that are used to determine where and how to apply instrumentation. There are several default instrumenter configuration files located in the

```
<i3_root>/products/i3fp/registry/products/j2ee/config/instrumenter and
<i3_root>/products/i3fp/registry/products/j2ee/config/instrumenter/sample directories.
```

The following table lists and describes these files:

Table 18-2 Instrumentation configuration files

File name	Description
Logger.xml	Product configuration.
Planners.xml	Controls the order in which instrumentation is applied.
Heatseeker.xml	Adaptive instrumentation analysis results that are loaded at startup. This file is regenerated when adaptive instrumentation policies are run.
Ixp.xml	Instrumentation Explorer applied changes that are loaded at startup. This file is regenerated when you click the Instrumentation Explorer Apply Changes button.
Survey.xml	Adaptive survey instrumentation configuration.
SurveyConditional.xml	Adaptive conditional instrumentation configuration.
SurveySynchronization.xml	Adaptive synchronization instrumentation.
Servlet.xml	Default Java Servlet instrumentation configuration
GenericPortal.xml	Default Generic Portal-specific instrumentation configuration. Detects the portal and portlet configurations that implement javax.portlet.GenericPortlet.
GenericPortlet.xml	Default Generic Portlet-specific instrumentation configuration. Instruments the portlet lifecycle and action methods.
JSP.xml	Default JSP instrumentation configuration.
WebLogicJSP.xml	Default (BEA WebLogic™-specific) JSP instrumentation configuration.
WebSphereJSP.xml	Default (IBM® WebSphere®-specific) JSP instrumentation configuration.
WebLogicPortal.xml	Default BEA WebLogic Portal-specific instrumentation configuration. Detects the portal and portlet configuration. This file is populated when an application server portal version is selected in Precise Framework Installer.
WebLogicPortlet.xml	Default BEA WebLogic Portlet-specific instrumentation configuration. Instruments the detected portlets. This file is populated when an application server portal version is selected in Precise Framework Installer.
EJB.xml	Default EJB instrumentation configuration. Handles instrumentation of EJB stubs.
Ignore.xml	Default “ignored” instrumentation configuration.
CallsToJDBC.xml	Default “caller-side” JDBC instrumentation configuration.
OverInstrumentationProtection.xml	Over-instrumentation protection instrumentation configuration.
IndepthWeb.xml	Default instrumentation configuration that Precise for Web uses. This file is populated when Precise for Web is installed.
TACPeopleSoft.xml	Insight SmartLink for PeopleSoft instrumentation configuration.
TACWebApps.xml	Insight SmartLink for Web applications instrumentation configuration
Custom.xml	Contains an example instrumentation only. You should use this file as an example and edit this file.

CallsFromMethodToMethod.xml	Contains an example instrumentation only. You must edit this file.
LeakSeeker.xml	Configuration for Leak Seeker instrumentation.
WebLogicEJB.xml	Handles BEA WebLogic-specific EJB “lifecycle operation” instrumentation.
WebSphereEJB.xml	Handles IBM WebSphere-specific EJB “lifecycle operation” instrumentation.
OracleEJB.xml	Handles Oracle-specific EJB “lifecycle operation” instrumentation.
JNDI.xml	Default JNDI instrumentation configuration.
DataSource.xml	Default JDBC DataSource instrumentation configuration.
EJBBean.xml	Default EJB implementation instrumentation.
JTA.xml	Default Java Transaction instrumentation.
MessageDrivenEJB.xml	Default Message-Driven EJB instrumentation.
JMS.xml	Default Java Messaging Service instrumentation.
XML.xml	Default XML and XSL instrumentation.
Calls.xml	Template for configuring all calls to and all calls from methods.
EJBImpl.xml	Sample EJB implementation instrumentation.
Jolt.xml	Sample Jolt instrumentation.
MBeanImpl.xml	Sample MBean implementation instrumentation.
PeopleSoft.xml	Old PeopleSoft instrumentation.
SAP61.xml	SAP 6.1 instrumentation.
SmartuneInstrumentation.xml	Optional SmarTune instrumentation for servlet include and session analysis.

About the structure of instrumenter configuration files

The instrumenter configuration files have the following general structure:

```
<?xml version='1.0'?>
<instrumenter-config>
    <custom-config> </custom-config>
    <all-calls-to-method> </all-calls-to-method>
    <all-calls-from-method> </all-calls-from-method>
    <calls-from-method-to-method> </calls-from-method-to-method>
    <ignore-config> </ignore-config>
</instrumenter-config>
```

See About custom instrumentation configuration, About all calls to method instrumentation configuration, About all calls from method instrumentation configuration, About calls from method to method instrumentation configuration, and About ignore instrumentation configuration.

About custom instrumentation configuration

The <custom-config> element has the following structure:

```
<custom-config>
    <java-classes>
        <java-class> <!-- occurs 0 or more times -->
            <class-name> class-or-interface-name </class-name>
                <methods>
                    <method> <!-- occurs 0 or more times -->
                    <name> method-name </name>
                    <params> <!-- optional -->
                        <param> <!-- occurs 0 or more times -->parameter-type </param>
                    </params>
                    <capture-param-index> <!-- optional -->capture-parameter </capture-param-index>
                    </method>
                </methods>
        </java-class>
    </java-classes>
</custom-config>
```

About all calls to method instrumentation configuration

The <all-calls-to-method> element has the following structure:

```
<all-calls-to-method>
    <methods>
        <method> <!-- occurs 0 or more times -->
            <name> method-name </name>
            <params> <!-- optional -->
                <param> <!-- occurs 0 or more times -->
                parameter-type </param>
            </params>
        </method>
    </methods>
</all-calls-to-method>
```

About all calls from method instrumentation configuration

The <all-calls-from-method> element has this structure:

```
<all-calls-from-method>
    <java-classes>
        <java-class> <!-- occurs 0 or more times -->
            <class-name> class-or-interface-name </class-name>
            <methods>
                <method> <!-- occurs 0 or more times -->
                <name> method-name </name>
                <params> <!-- optional -->
                    <param> <!-- occurs 0 or more times -->
                    parameter-type </param>
                </params>
                </method>
            </methods>
        </java-class>
    </java-classes>
</all-calls-from-method>
```

About calls from method to method instrumentation configuration

The <calls-from-method-to-method> element has this structure:

```
<calls-from-method-to-method>
    <invocation-relationship> <!-- occurs 0 or more times -->
        <java-class>
            <class-name> class-or-interface-name </class-name>
                <methods>
                    <method> <!-- occurs 0 or more times -->
                        <name> method-name </name>
                        <params> <!-- optional -->
                            <param> <!-- occurs 0 or more times -->
                            parameter-type </param>
                        </params>
                    </method>
                </methods>
        </java-class>
        <invoked-method> <!-- occurs 0 or more times -->
        invoked-method-name </invoked-method>
    </invocation-relationship>
</calls-from-method-to-method>
```

See About method signature matching.

About ignore instrumentation configuration

Use the <ignore-config> element to configure rules that are used to determine when to prevent instrumentation from being applied to all methods in specifically matched classes or packages, to specifically matched methods, or to specifically matched calls to methods.

The <ignore-config> element has this structure:

```
<ignore-config>
    <java-classes> <!-- optional -->
        <!-- Purpose and structure described below. -->
    </java-classes>
    <invocation-relationship> <!-- occurs 0 or more times -->
        <!-- Purpose and structure described below. -->
    </invocation-relationship>
    <all-calls-to-method> <!-- optional -->
        <!-- Purpose and structure described below. -->
    </all-calls-to-method>
    </instrumenter-config>
```

Use the <java-classes> element to prevent instrumentation from being applied to all methods in specifically matched classes or packages or to specifically matched methods. The <java-classes> element has this structure:

```
<java-classes>
    <java-class> <!-- occurs 0 or more times -->
        <class-name> class-or-interface-name </class-name>
        <methods> <!-- optional -->
            <method> <!-- occurs 0 or more times -->
            <name> method-name </name>
            <params> <!-- optional --> </params>
            </method>
        </methods>
    </java-class>
</java-classes>
```

See About method signature matching.

For each method to be instrumented, the following rules are used to determine if instrumentation should not be applied to the method:

- If the method is declared in a class whose name matches the specified class or interface name of a <java-class> element, no instrumentation should be applied to the method.
- If the matched <java-class> includes a <methods> element, the method must also match the specified method name of a <method> element so that no instrumentation is applied to the method.
- If the matched <method> element includes a <params> element, the method must also match the specified signature so that no instrumentation is applied to the method.

Wildcards are permitted in the class or interface name and method name. See About using the wildcard character *.

Use the <invocation-relationship> element to prevent instrumentation from being applied to specifically matched calls to methods from a specifically matched calling method. The <invocation-relationship> element has the following structure:

```
<invocation-relationship> <!-- occurs 0 or more times -->
    <java-class>
        <class-name> class-or-interface-name </class-name>
        <methods>
            <method> <!-- occurs 0 or more times -->
            <name> method-name </name>
            <params> <!-- optional -->
            </params>
            </method>
        </methods>
    </java-class>
    <invoked-method> <!-- occurs 0 or more times -->
    qualified-method-name </invoked-method>
</invocation-relationship>
```

See About method signature matching.

For each call to a method to be instrumented, the following rules are used to determine if instrumentation should not be applied to the method:

- If the called method is declared in a class whose name matches the specified class or interface name of a <java-class> element, no instrumentation should be applied to the method.
- If the matched, <java-class> includes a <methods> element, the method must also match the specified method name of a <method> element so that no instrumentation is applied to the method. If the matched <method> element includes a <params> element, the method must also match the specified signature so that no instrumentation is applied to the method.

Use the <all-calls-to-method> element to prevent instrumentation from being applied to specifically matched calls to methods from any calling method. The <all-calls-to-method> element has the following structure:

```
<all-calls-to-method>
    <methods>
        <method> <!-- occurs 0 or more times -->
            <name> method-name </name>
            <params> <!-- optional -->
            </params>
        </method>
    </methods>
</invocation-relationship>
```

See About method signature matching.

The <name> element must represent a fully qualified method name. The portion of the <name> element after the last dot (“.”) is considered to be the method name, and the portion of the <name> element before the last dot is considered to be the class name. When you use wildcards, it is important to use a wildcard pattern that fits the scheme described. For example, the wildcards pattern *.* matches all methods of all classes.

About common instrumenter configuration matching techniques

Common instrumenter configuration matching techniques are method signature matching and using the wildcard character *.

About method signature matching

The <params> element configures rules that are used to match method signatures for processing by the instrumenter. The <params> element has the following structure:

```
<params> <!-- optional -->
    <param> <!-- occurs 0 or more times --> parameter-type </param>
</params>
```

The parameter type is the same as the abstract type declarator for the parameter type in Java. For example, the parameter-type for a parameter of type java.lang.String is java.lang.String, and the parameter type for a parameter of type int[][] is int[][].

The following primitive parameter types names are recognized:

- `boolean`
- `byte`
- `char`
- `double`
- `float`
- `int`
- `long`
- `short`
- `void`

The <invoked-method> elements in the <calls-from-method-to-method> element are used to match specific method signatures for specific calls from one method to another. All methods that match the signature are instrumented. Wildcard expressions are not supported and return types are not matched. The <invoked-method> format must be expressed using the same primitive types that were discussed for the <param> element.

For example:

<invoked-method> java.lang.Thread.sleep(long) </invoked-method> or

<invoked-method> com.acme.shared.comm.Connector.<init> (java.lang.String,int[][]) </invoked-method>

About using the wildcard character *

In addition to using specific names to reference items such as classes, interfaces, and methods, for instrumentation, you can also use the wildcard character * to instruct Precise for J2EE to instrument all classes, methods, or packages within the scope of the instruction.

The following table illustrates how the wildcard character can be used.

Table 18-3 Usage of wildcard character *

Wildcard	Instrumented elements	Non-instrumented elements
*	xmp.server.Main xmp.task.AbstractTask xmp.task.AbstractTask$1 xmp.task.util.TaskUtilities	
xmp.task.*	xmp.task.AbstractTask xmp.task.AbstractTask$1 xmp.task.util.TaskUtilities	xmp.server.Main

$	xmp.task.AbstractTask$1	xmp.server.Main xmp.task.AbstractTask xmp.task.util.TaskUtilities

Use wildcard characters only when discovering the methods to instrument. Otherwise, it may result in instrumentation that does not yield meaningful performance metrics but introduces unwanted overhead. Do not implement wildcarded instrumentation in production environments.

Including application server classes in Leak Seeker instrumentation

To obtain information on collections and arrays with the most elements, Leak Seeker instruments all user application classes but, by default, excludes application server classes. In special circumstances, you may want Leak Seeker to collect information on application server classes as well.

Instrumenting application server classes may increase excessively the startup time as well as the running time of the application.

To include Leak Seeker instrumentation for application server classes

1. In the *JVMID*/LeakSeeker.xml file, find the application server class prefix for the application server classes that you want to instrument. For example, to identify WebLogic application server classes to instrument, you would search for lines beginning with com.bea and weblogic.
2. Comment out the lines for the application server whose classes you want to instrument. For example, the following lines cause Leak Seeker to collect information for WebLogic application server classes.

```
<!--leakseeker-if-package>com.bea</leakseeker-if-package>
<!--leakseeker-if-package>weblogic</leakseeker-if-package>
```