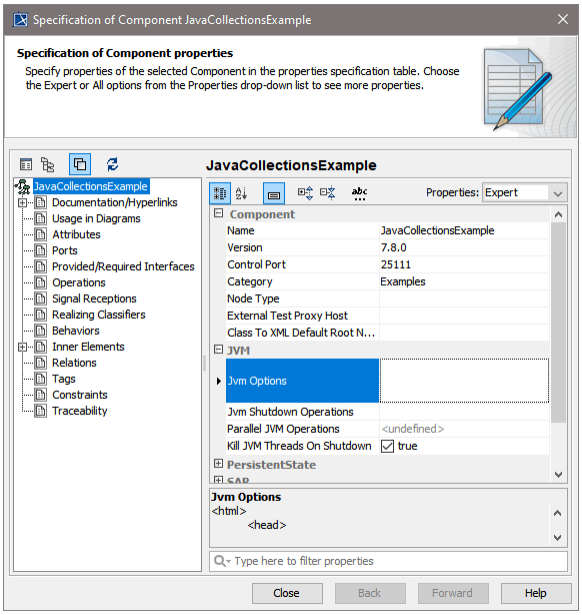
Modeling the Java Components

Each xUML service starts its own instance of a Java Virtual Machine (JVM) at service startup, if the composite contains one or more <<JavaComponent>>s.

The settings the JVM used to execute the imported Java code can be configured on the composite:



Its tagged values are:

Tagged Value	Description
Kill JVM Threads On Shutdown	If threads are still running on shutdown they are killed.
Jvm Options	Java Virtual Machine options. The option strings has one of the following formats: "-D=...", "-X...", "_...", The system properties java.class.path and java.library.path are set by the model compiler and can not be overridden in the model.

On this Page:

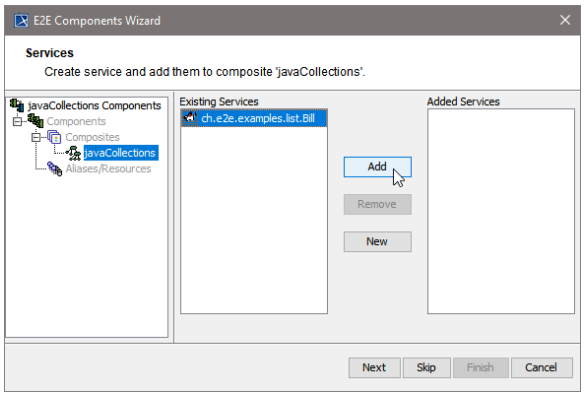
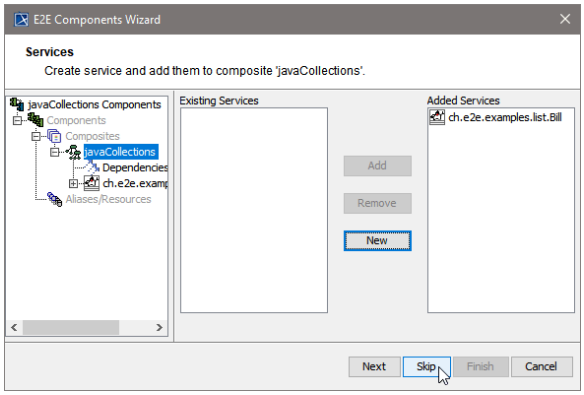
Related Pages:

- The Components Wizard
- Deploying and Managing Java Archives
- Importing Java™ Classes and Properties Resource Files

Jvm Shutdown Operations	Java operation being called before shutdown.
Parallel JVM Operations	Number of parallel open JavaVM threads within the runtime. Default: 100. If the limit is reached, the runtime tries for 60 seconds to obtain a free JVM thread. If it does not succeed, an error (JAVAADLM/19) is thrown.

A component diagram can be drawn with the help of the Component Wizard (for more information refer to The Components Wizard). To ensure that the component diagram is correct, we recommend the usage of the wizard.

The following pictures show how to add java components to component diagram.

	All imported Java Components are listed in column **Existing Services**. Take all the items the service needs for its execution from the list on the left hand side to the right hand side and click **Add**.
	Upon completion, the component wizard will draw the component diagram as exemplified by the first figure.

Java Archive Artifacts

<<JarFile>> artifacts have two tagged values, **deploy** and **boot**.

In the example above, the artifact **JavaCollections.jar** will be deployed together with the compiled xUML service (the tagged value **deploy** is set to **true**). It is also possible to deploy Java archives via the Bridge to prevent transferring big amounts of data when deploying them together with the service repository via the Builder. Another advantage of deploying Java archives via the Bridge is that they are not only used locally by the xUML service as it would be the case with the Java archive **JavaCollections** in the example above, but also globally by all xUML services deployed to the node instance. For more details on deploying Java archives via the Bridge, refer to Deploying and Managing Java Archives.

The tagged value **boot** indicates if the Java archive is to be contained in the Java boot class path. In the example above, the tagged value is set to **false**.

The Importer will prompt you to define both tagged values for each Java archive you want to import. For more details, see Importing Java™ Classes and Properties Resource Files.

> In the Builder, you must always import all Java archives that are necessary at runtime, no matter if you deploy them via the Builder or the Bridge. This way, all involved Java archives will be documented in the component diagrams, too. All imported Java archives reside in the <<JavaComponent>> created by the Java Importer.

Resource File Artifacts

The Importer can also import properties resource files, which imported Java class may need at run-time. In the component diagram a resource file is represented by an artifact stereotyped as <<ResourceFile>>. It is a resident of an instance of a Java Component artifact. Logically, a resource file artifact is a manifestation of a class having the stereotype <<Resource>>.