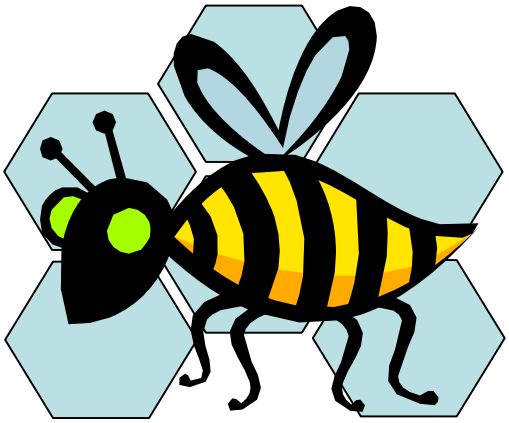
ComponentBee v. 1.0beta



Usage Instructions

Draft 30.11.2008

1 The overview of the ComponentBee

The initial aim of the ComponentBee is to support unit-level reliability testing that measures Probability of Failure values (the PoF or p_{ij} values) for SW components. The p_{ij} value is calculated for an implementation component i (IC_i) and a use case j (UC_j) with the following formula:

$$p_{ij} = \frac{UC_jCount_{IC_i\_Failure\_in\_UC_j}}{UC_jCount_{IC_i\_Participates\_in\_UC_j}} \qquad (1)$$

where $UC_jCount_{IC_i\_Failure\_in\_UC_j}$ defines the total number of UC_j in which the IC_i has caused a failure. $UC_jCount_{IC_i\_Failure\_in\_UC_j}$ defines the total number of UC_j to which the IC_i has participated in.

As can be seen from the Formula 1, the calculation of the p_{ij} value requires the dynamic behaviour of the SW component to be evaluated: Firstly, the UC_j must be recognized from the execution paths. Secondly, it must be recognized when the IC_i participates in the UC_j. Thirdly, it must be identified when the IC_i causes failure in the UC_j. The ComponentBee tool helps in these tasks and is capable of evaluating concurrent behaviour of Java components (evaluation steps are presented in Figure 1) and finally calculating the measured p_{ij} values (PoF_M values) for tested use cases and SW components.

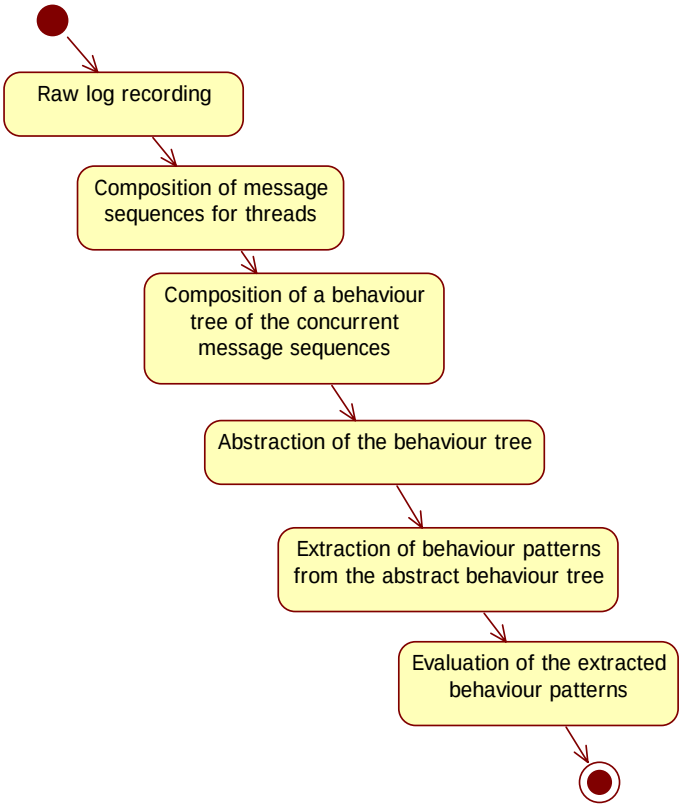


Figure 1. The concurrent behavior evaluation of SW components.

The Component Behaviour Evaluation (ComponentBee) tool (previously called RT-tool, see [ImP07]) works in the Eclipse platform and evaluates the dynamic behaviour of SW components in the following steps (Figure 1). The ComponentBee:

1. Collects raw log data about the dynamic behaviour of SW components first, then
2. Takes the raw log as an input and composes message sequences for threads,
3. Recognizes how these different threads interact and composes an overall behaviour tree for the concurrent message sequences,
4. Abstracts the overall behaviour tree so that the abstract behaviour tree will contain only those messages that are known in the test model,
5. Extracts behaviour patterns from the abstract behaviour tree, and finally
6. Evaluates the extracted behaviour patterns and calculates reliability values for the SW components and for the executed use cases.

The ComponentBee uses test models in the dynamic behaviour evaluation and supports composition of generic and reusable test models that can be refined to support dynamic testing of various kinds of SW components.

1.1 Features

The ComponentBee provides the following features:

1. **Support evaluation of concurrent behaviours executed in various threads.** The ComponentBee facilitates behaviour evaluation of SW components doing concurrent processing.

2. **Support for run-time utilisation of Java plug-ins that are developed in the workspace of the Eclipse.** The ComponentBee provides a framework for reusable plug-ins that either collect, process, or evaluate raw log data in the Eclipse environment. The Eclipse plug-ins are typically developed with the tools of the Eclipse. After a plug-in is implemented, a new Eclipse instance must be started and the plug-in is tested in it. However, it is quite a time and memory consuming task to start an Eclipse instance. For example, it took 1 minute and 17 seconds to start an Eclipse instance in a laptop computer (Dell Latitude D800, with a 1.7GHz Intel Pentium M Processor 735 and 2 GBytes of RAM. In order to make the development of plug-ins more fluent, the ComponentBee offers a dynamic invoking mechanism for plug-ins that are manually coded or generated within the Eclipse workspace. An SW integrator can create a Java project in the Eclipse workspace, write or generate new plug-ins, compile these, and then attach the plug-ins to the test model, and finally dynamically utilise these plug-ins. The elements of the test model will finally configure and invoke the plug-ins to support the reliability evaluation of the SW components. This all can be performed without starting a new Eclipse instance. This leads to the outstandingly faster development and test cycles of plug-ins.

3. **Support for reusable and configurable plug-ins.** The input elements of the test model can adapt the ComponentBee plug-ins to support behaviour testing of different kinds of SW component.

4. **Tool support for both generic and domain and application-specific test model creation.** The ComponentBee contains a visual editor for test models that supports creation of test models at different abstraction levels. For example, the domain or application-specific test models can refine the more abstract test models. As a result, it is possible to reuse test models in different applications or application domains.

5. **Dynamic instrumentation.** The probes of TPTP enable the ComponentBee to observe the dynamic behaviour of components without adding test code to the actual SW components.

6. **Utilisation of information embedded in source code.** The ComponentBee uses JDT for obtaining information about Java interfaces and classes.

1.2 Dependencies to other OS projects

The ComponentBee is based on the components of several OS projects. The following lists the most important OS techniques that are utilised in the ComponentBee:

1. **Eclipse Test & Performance Tools Platform Project (TPTP)** (http://www.eclipse.org/tptp/)
   The dynamic behaviour logging of SW components is based on TPTP's probes.
2. **Graphical Editing Framework (GEF) (**http://www.eclipse.org/gef/**) and Graphical Modelling Framework (GMF) (**http://www.eclipse.org/modeling/gmf/). The visual editor of the ComponentBee is based on GMF and GEF.
3. **Eclipse Java Development Tools (JDT) (**http://www.eclipse.org/jdt/**)** enables the ComponentBee to access information embedded in the source code of Java components. The JDT is a part of the Eclipse and is not thus needed to be separately downloaded.
4. **Java Compiler Compiler (JavaCC)** (https://javacc.dev.java.net/) is a Java Parser Generator. The ComponentBee uses the JavaCC for generating a behaviour parser for a test model. The JavaCC classes are included in the ComponentBee installation package and thus the JavaCC is not needed to be separately downloaded and installed to the Eclipse environment.

2 Installation of the ComponentBee

The easiest way is to obtain the ComponentBee's installation packages is to download them directly from the Web page (http://www.vtt.fi/proj/cosi/cosi_ComponentBEE.jsp?lang=en) of the ComponentBee. Do the installation of the ComponentBee in the following steps:

1. **Download the ComponentBee installation package first.**

2. **Download consolidated requirements zip for the ComponentBee. The zip includes all the required bundles and an Eclipse platform as a single zip.**

   Alternatively, the required OS components can be downloaded from the Web pages of separate OS projects, too. The following installation packages are needed for the ComponentBee:

   1. **TPTP 4.5.0M3:**
      Download TPTP all in one package and its requirements from the address:
      http://www.eclipse.org/tptp/home/downloads/?buildId=TPTP-4.5.0M3

   2. **GEF 3.4M3:**
      Download GEF 3.4M3 from the address:
      http://www.eclipse.org/gef/downloads/

   3. **GMF Base Build: B-M20071123:**
      Download GMF Base Build: B-M20071123 from the address:
      http://download.eclipse.org/modeling/gmf/downloads/drops/B-M-B-M20071123-200711230000/index.php

   4. **GMF SDK:**
      Download GMF SDK (GMF-sdk-M20071127.zip) from the address:
      http://www.eclipse.org/modeling/gmf/

      Notice! Exactly this version must be downloaded, for example, the Maintenance Build: M20080215 version of GMF is not compatible with the ComponentBee.

3. **Unzip the downloaded installation packages (Figure 2 and Figure 3) and the ComponentBee installation package (componentbee_install.zip) to the Eclipse directory.**

4. **Edit the eclipse.ini file (exist in eclipse directory) and extend the heap memory size of the Eclipse.** This is required if the Eclipse is used for handling very large raw log files (the files that size is 100 Mbytes or more). Thus, edit the eclipse.ini file (Figure 3) that exist in the Eclipse directory to look like this:

   ```
   -showsplash
   org.eclipse.platform
   --launcher.XXMaxPermSize
   256m
   -XstartOnFirstThread
   -vmargs
   -Xms256m
   -Xmx512m
   ```

5. **Start the Eclipse with –clean option (eclipse.exe –clean) for ensuring that all the installed plug-ins work properly.**

6. **Select a location for the Eclipse workspace. After that the ComponentBee is ready for use.**

7. **You can optionally download a simple example for the ComponentBee from the homepage of the ComponentBee.** The example shows how the reliability values can be measured for an asynchronous client-server application with the ComponentBee. The usage instructions for the example are provided in the example package.

Figure 2. Unzip the required bundles to the Eclipse directory.

Figure 3. Unzip the componentbee_install.zip in the Eclipse directory.

3 Usage of the ComponentBee

The ComponentBee supports reliability testing of Java components. The testing steps are described in the following subsections:

3.1 Creation of a test bed

The ComponentBee is an evaluation tool that does not directly support test bed or test data creation. One solution is to create the test test bed with Eclipse tools. This can be made in the following steps:

1. Create a Java project in the Eclipse workspace for the test bed.
2. Select the project properties and include the **common.runner.jar, java.runner.jar,** and **componentbeetoolkit.jar** libraries to the Java build path (Figure 4).

   The componentbeetoolkit.jar exists in the path:
   INSTALL_PATH/eclipse/plugins/ComponentBee_1.0.0/componentbeetoolkit.jar.

   The commonrunner.jar and java.runner.jar packages exist in the path:
   INSTALL_PATH/eclipse/plugins/org.eclipse.hyades.test.tools.core_4.4.100.v200710300400

3. Insert (or create) your own test bed implementation (e.g. required classes and test data) to the created project path. The ComponentBee can be utilised after the test bed and required components are available in the Java project.

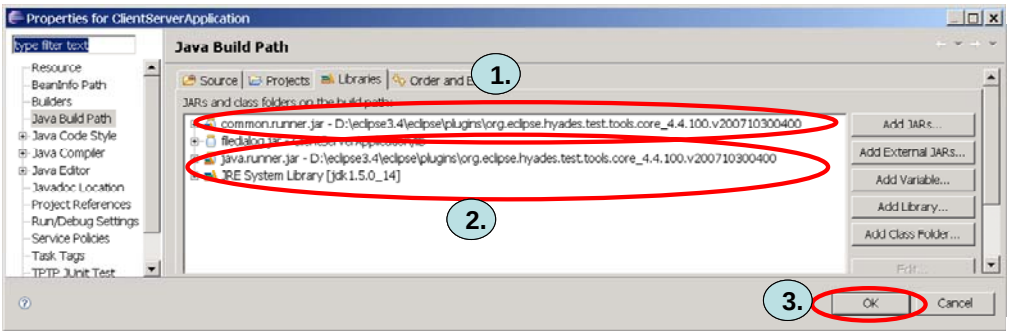


Figure 4. The project properties dialog of the Eclipse.

3.2 Creation of the test model

The ComponentBee provides a visual editor for constructing test models that define expected behaviours for the use cases and for the SW components under testing. The adaptor elements of the test model can later adapt the behaviour model for the actual component implementations.

[NOT SUPPORTED FEATURE IN THE ComponentBee 1.0.0 Beta
UML sequence diagrams can be imported to the test models of the ComponentBee. An *UMLImporter* plug-in that imports TOPCASED UML models to the test models of the ComponentBee is under construction.**]**

3.3 Adaptation of the test model

The behaviour test editor of the ComponentBee is capable of importing a test model, refining it for the target SW components (it adds adaptor elements to the test model), and finally recording raw log data about the dynamic behaviour of the components. This is done in the following steps:

1. **Creation of adaptors.** The behaviour test editor provides a tree view of the available SW components and their methods. The tester can select methods, attach them to the messages of the test model, and by this way add new adaptor elements to the test model.

2. **Selection of raw log writers.** It is possible to decrease the size of the raw log by selecting those raw log writers that will add the needed data to the raw log. Raw log writers are defined in the adaptor elements of a test model. The ComponentBee provides the following ready-made raw log writers:
   1) The *state data writer* records the state data of a component,
   2) The *input data writer* records data about the input parameters of a method,
   3) The *output data writer* records data about the return values or thrown exception of a method,
   4) The *default writer* records state data of a component and input parameters and a return value or a thrown exception of a method, and
   5) The *trace data writer* records the trace data to the raw log.

   In addition, it is possible to implement new raw log writer plug-ins to add additional data to the raw log. For example, the defined message classifier plug-ins can later utilise this data when they are classifying the logged messages to different classes.

3. **Selection of message classifiers.** The adaptor element can define one or more message classifiers plug-ins that use data provided in the raw log (e.g. state data of components and input parameters, return values, and thrown exceptions of methods) and finally classify messages to different classes.

   The message classifier defines identifiers for data that it needs in message classification. The *input elements* define data sources for the data identifiers and thus can configure message classifiers to classify messages delivered between different kinds of SW components.

4. **Generation of a Probekit.** The behaviour test editor is capable of generating a Probekit for the test model. The profiler tool can later use the Probekit and collect raw log data about dynamic behaviour of components.

3.4 Execution of the behaviour test

The Raw log data recording is done in the following steps:
1. Open the Profile view of the Eclipse.
2. Create new launch configuration and select project and Java main class for it.
3. Add **commonrunner.jar, java.runner.jar,** and **componentbeetoolkit.jar** packages (and the other JAR packages that the target SW needs) to the class path before profiling.
4. If you use your own RawLogWriters in profiling, add these plug-ins to a JAR package and insert the JAR package to the class path.
5. Go to probe insertion dialog and select the generated Probekit.
6. Define a filter set for the application (the following client-server example shows how it must be done).
7. Click the *profile* button that will now start the test execution.
8. Click the stop button in the profiling monitor view after the test sequence is executed. The raw log file is now available in the test project path.

3.5 Evaluation of the dynamic behaviour of the SW components

The ComponentBee uses pre-processor components (actually a pipe of pre-processors) in raw log data evaluation. Raw log data is evaluated in the following steps:

1. **Generation of a behaviour parser.** The ComponentBee tool uses a BNF-based parser for extracting behaviour patterns from the collected raw log file. Select a Java project for the behaviour parser classes first. Then, click the "*generate parser classes*" button. The ComponentBee will now generate a behaviour parser for the test model.

2. **Creation of evaluator plug-ins.** The reliability evaluation requires plug-ins capable of recognizing the failed messages and to define which SW components have caused failures in the executed use cases. The generated *BehaviourParser* extracts behaviour patterns from the pre-processed raw log data. *Evaluators* navigate in the behaviour pattern tree, evaluate and add evaluation data to the behaviour patterns, and finally calculate reliability values for the components and executed use cases. The *abstract use case evaluator* plug-in of the ComponentBee is capable of calculating and finally recording p_{ij} values to the test report. Before this plug-in can be used, we must extending the *abstract use case evaluator* plug-in with methods that evaluate the messages related to the use case and identifying the components that have participated in or caused failures in the tested use cases.

3. **Adding of the evaluators plug-ins to the test model.** The evaluator plug-ins are attached to the test model and later invoked to calculate p_{ij} values for the SW components and tested use cases.

4. **Creation of a test report.** Next, (if needed) select the parser class path (the generated parser class path is set automatically) and raw log file to be evaluated. If the analysis is successfully completed, the measured evaluation results are shown and an analysis report file is finally written in an XML format to the defined path.

4 Example – Reliability testing of the dynamic behaviour of a client-server application

We constructed a small client-server application to show how the ComponentBee can support in reliability testing of Java components. The client-side sends a request for the server that delivers a correct or failure response for the client. The correct and failure behaviours of the client-server application are shown in Figure 5.

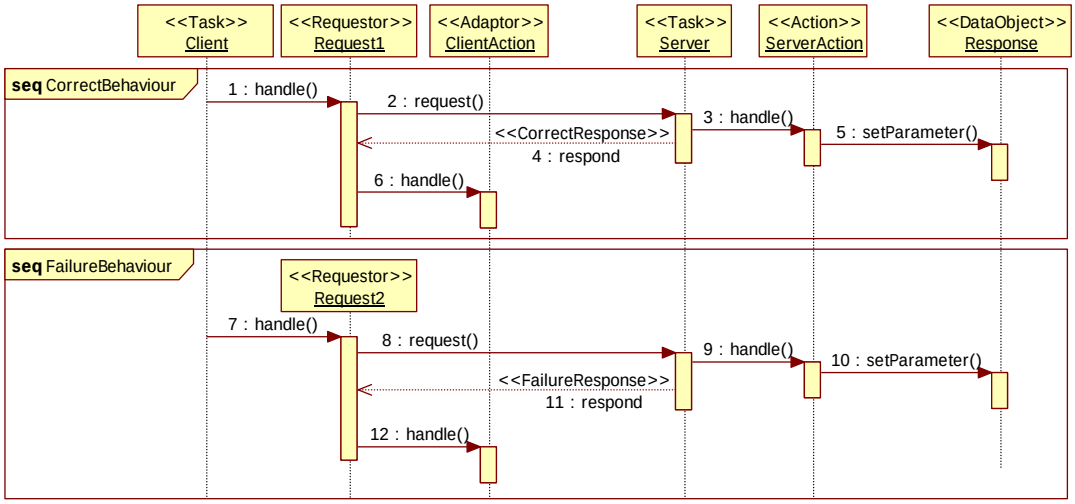


Figure 5. The correct and failure behaviour sequences of the task-based client-server application.

Both the client and server-side were constructed with the task-based composition technique [Pal07] that facilitates creation of asynchronous processing structures. We created first an XML-based composition schema that describes the client and server tasks and then Java-based client and server action plug-ins to do processing related to the application. The failure rate of the server action is configured in the composition schema. Thus it is easy to effect to the "reliability" of the client-server application. The testing steps are described in the following subsections.

4.1 Creation of a test bed

The testbed was created in the following steps:

1. A Java project for the task-based client-server application was created in the Eclipse first.
2. We selected the project properties view and added the external **common.runner.jar**, **java.runner.jar,** and **componentbeetoolkit.jar** libraries to the Java build path (Figure 6).
3. The Java classes must be located in the root of the project path. The probes are used in raw log data recording. The utilisation of probes requires that the Java classes are in the root of the java project (not in projectname/src folder). By default the Eclipse creates a source (*src/*) folder and adds source files to the folder. In order to prevent this, go to the source tab in the Java Build Path view (Figure 7), select the default src/ folder, and click remove. Push the "Add Folder" button and then select the project root to be the source folder. Finally, (if needed) copy the Java packages under src folder to the root of the Java project.
4. The classes of the task-based client-server application were inserted to the created Java project.

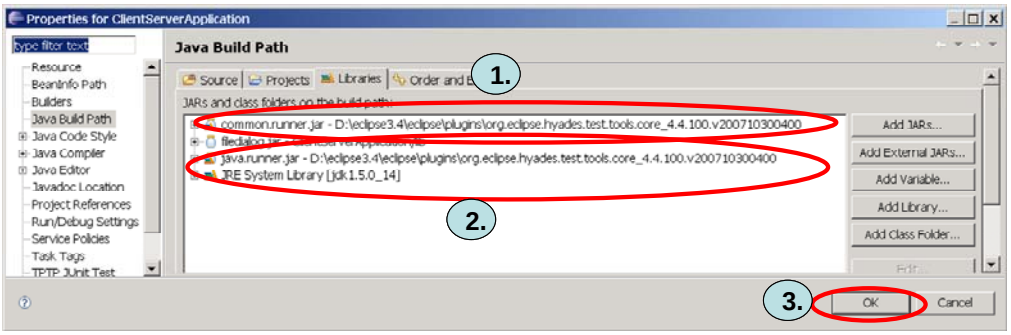


Figure 6. The project properties dialog of the Eclipse.

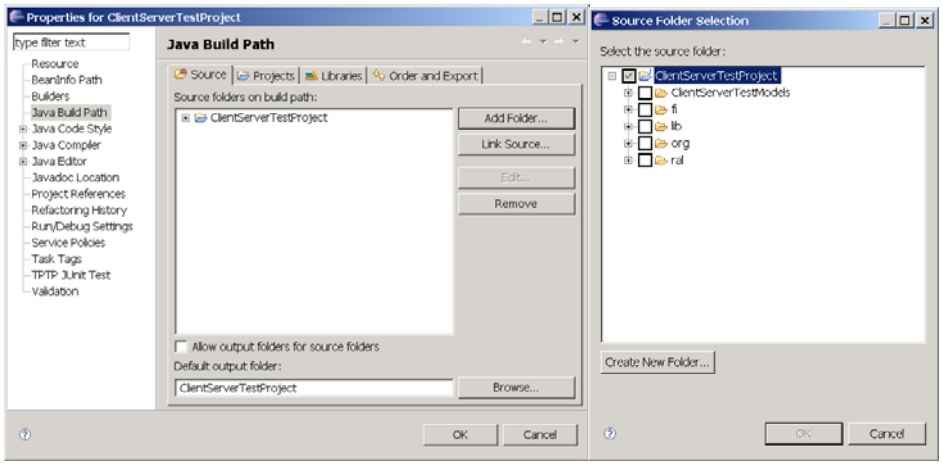


Figure 7. The project properties dialog of the Eclipse.

4.2 Creation of the test model

The ComponentBee uses test models in reliability evaluation. The test model is created in the following steps for the client-server application:

1. We used the create test model wizard (Figure 8, Figure 9, and Figure 10) and created an abstract test model for reliability testing with the visual test editor of the ComponentBee (Figure 11) first.
2. We created a domain-specific test model for client-server applications that refines the generic test model with domain-specific elements (Figure 12). We defined the following elements to the test model to import new elements to the test model:
   1. The behaviour model element (Element 1) imports the generic behaviour model first. (the test model are located in the test model package directory). The source is formatted as follows:

      source = ModelId :// ModelElementId

      The *ModelId* defines the filename for the abstract model and the *ModelElementId* is an identifier for the model element to be imported.

   2. The extension elements (Elements 2 and 3) attach their child elements to defined (e.g. imported) model elements. The target attribute specifies the model elements to which the child elements are attached. The target attribute is formatted as follows:

      Target =TestModelName://ChildModel/ElementId[#ConnectionType]
      ChildModel =[ModelId|ADAPTORS|EVALUATORS]"+
      ElementId = TargetElementID]|ROOT|]"+*
      ConnectionType = START | NEXT | CALL | ASYNCBEHAVIOUR | ADAPTOR | EVALUATOR

3. Finally, we created a test model for the actual Java components of the client-server application (Figure 13). It imports the domain-specific test model first. The test model is later refined with adaptor and evaluator elements for the actual implementation components.

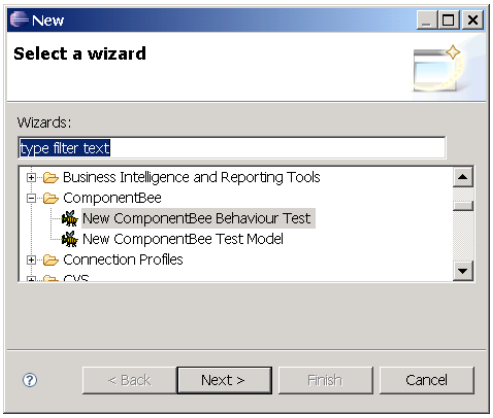


Figure 8. Step 1: A test model is created with the new wizard.

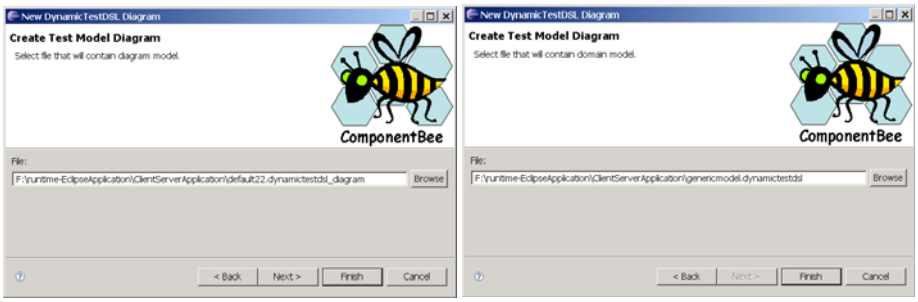


Figure 9. Step 2. The test model diagram and test model names are entered.

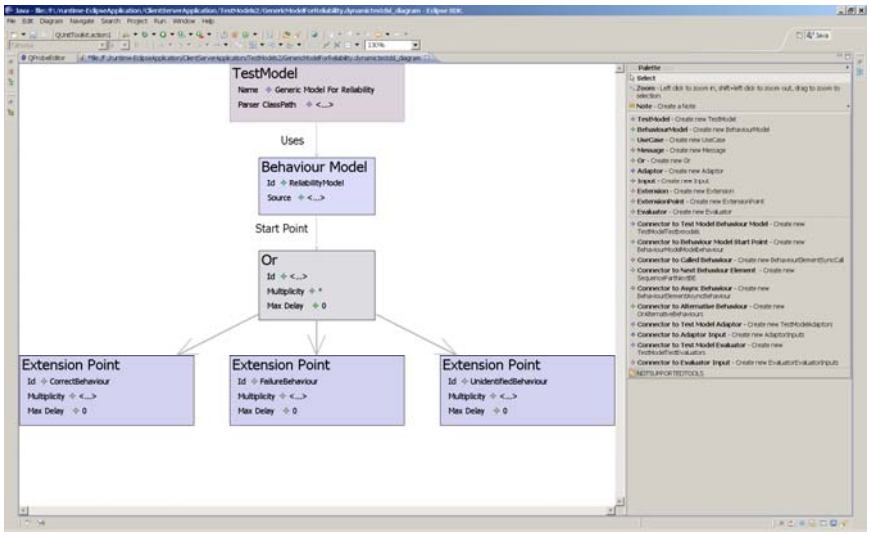


Figure 10. Step 3. Test models are edited with the visual editor of the ComponentBee.

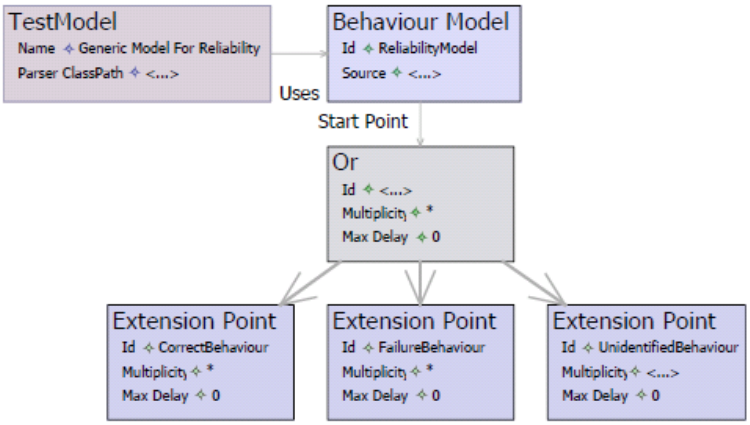


Figure 11. The generic model for reliability testing provides extension points for correct, failure, and unidentified behavior.

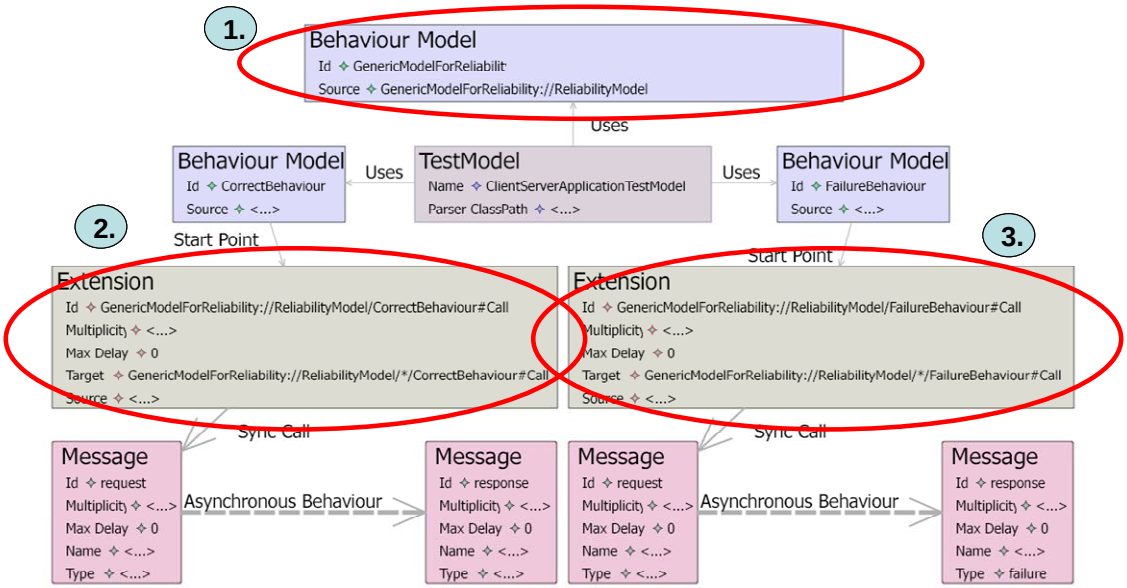


Figure 12. A domain-specific model that refines the abstract test model.

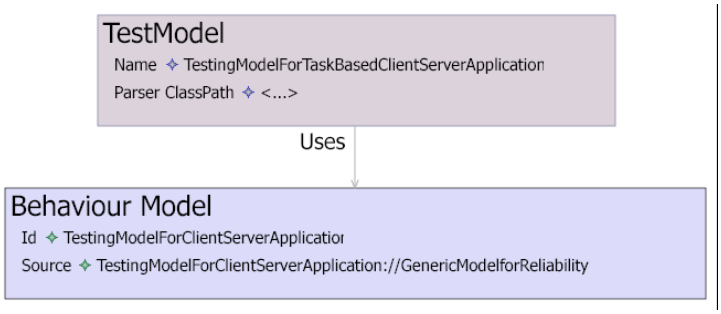


Figure 13. A test model for the task-based client-server application.

4.3 Creation of the behaviour test

The test model is refined for the actual SW components with the behaviour test editor of the ComponentBee. The behaviour test is created in the following steps:

1. **Behaviour test creation.** An empty behaviour test is created with the new wizard to the test bed project path (Figure 14).
2. **Test model loading.** The test model is loaded to the behaviour test editor (Figure 15). The behaviour test editor takes a backup of the loaded test model before the elements of the abstract and domain-specific test models are composed to the loaded test model. The refined test model is be later illustrated and edited with the visual test editor (Figure 16).
3. **Test model adaptation.** The behaviour test editor of the ComponentBee is used for adapting the downloaded test model for the target SW components. The adaptors view shows a tree view of the available SW interfaces and components and their methods. By selecting a method and by clicking a right mouse button it is possible to open a popup menu and to create an adaptor element for a message defined in the test model (Figure 17).
4. **Selection of message classifiers.** The right-side of the adaptors view shows the adaptor elements. By selecting an adaptor element it is possible to open a popup menu that shows the raw log writer or message classifier plug-ins that are available in the Eclipse workspace and to insert a raw log writer or a message classifier to the adaptor element (Figure 18). These plug-ins are used later when the raw log is recorder and evaluated.
5. **Defining inputs for message classifiers.** A message classifier defines identifiers for data that it needs in message classification. In the adaptor view it is possible to insert new input elements to message classifiers (Figure 19). The *input elements* define data sources for the data identifiers and thus can configure message classifiers to classify messages delivered between different kinds of SW components. For example, it is possible to define a data source for id “Input1”. The message classifier plug-in can now obtain the configured data from the raw log by using the identifier “Input1” (Figure 20).
6. **Creation of a ProbeKit for target SW components.** In the ProbeKit view of the behaviour test editor it is possible to generate a ProbeKit for the test model (Figure 21).

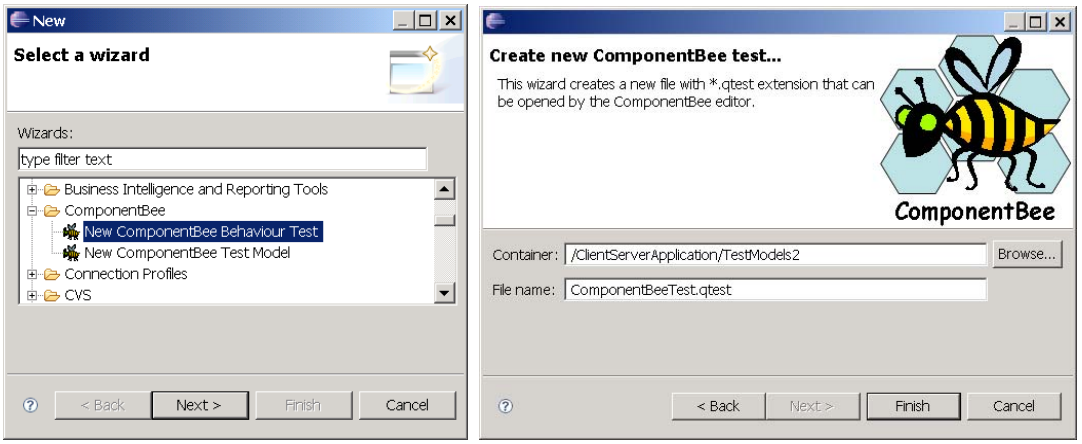


Figure 14. The behavior test is created with the new wizard.

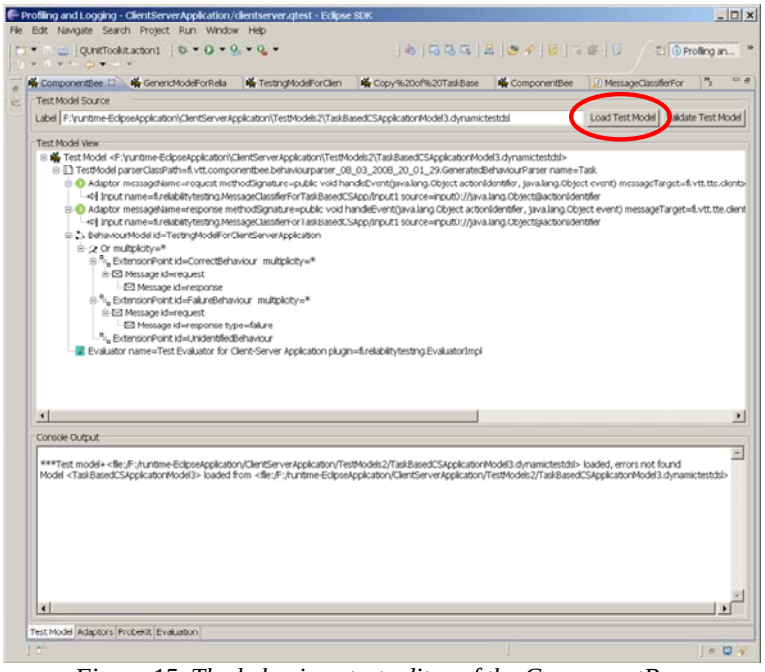


Figure 15. The behaviour test editor of the ComponentBee.

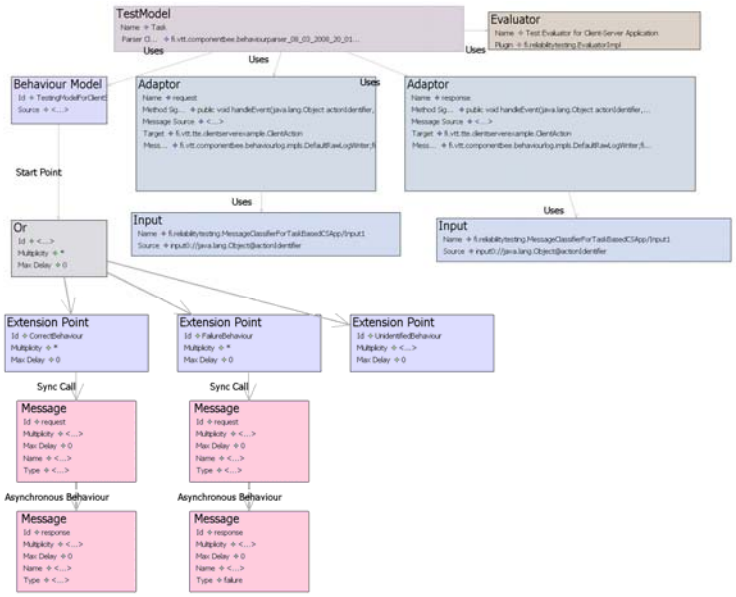


Figure 16. The overall test model for the task-based client-server application.

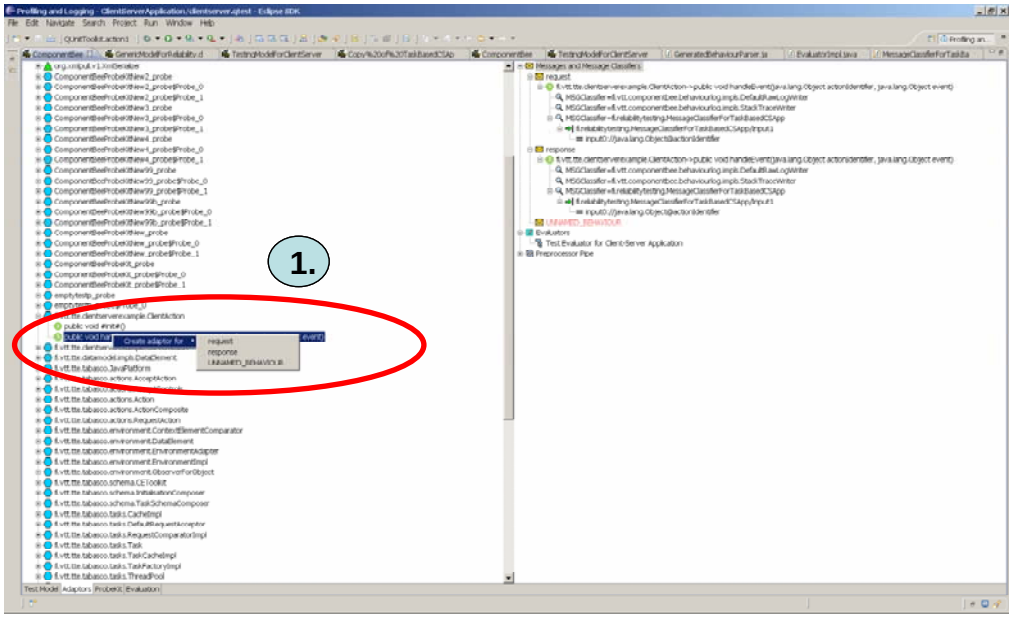


Figure 17. Adding an adaptor element for a message that is defined in a test model.

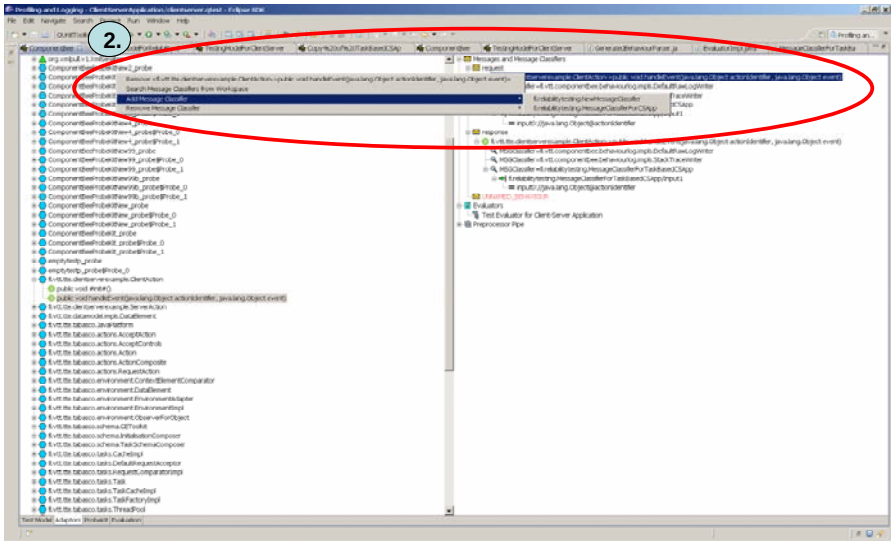


Figure 18. Adding a raw log writer or message classifier plug-in for an adaptor element.

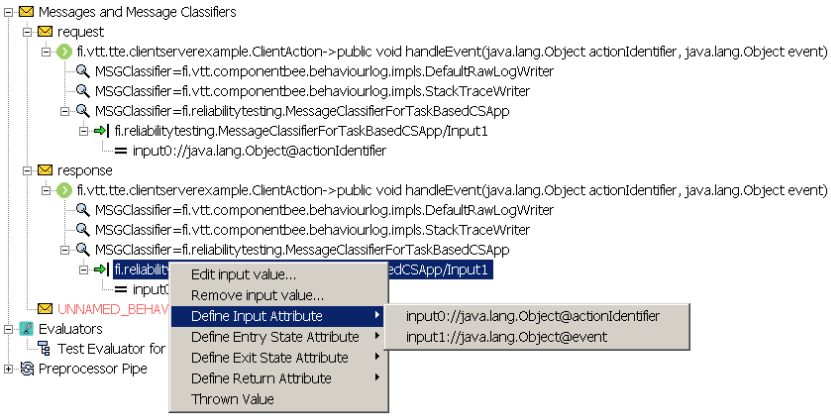


Figure 19. Defining a data source for a message classifier plug-in.

```
public final class MessageClassifierForTaskBasedCSApp implements MessageClassifier,RawLogWriterExtension {

		//... RAW LOG WRITER CODE IS WRITTEN HERE...

		// This method accepts all messages that are delivered for this message classifier
		public boolean accept(final Message message){return true;}

		// Returns an enumeration of inputs that this message classifier plugin needs.
		public Enumeration getRequiredInputIds(){
				Vector inputs=new Vector();
				inputs.add("Input1");
				return inputs.elements();

		}

		// Plugin classifies the given message here...
		public void classify(Message executedMethod,LogInformation logInformation, Observer observer)  throws Throwable{
				// Fetches required data from the log...
				String string=(String) logInformation.getData( "Input1", executedMethod);

				// Classifies message name to be a request or a response
				if(string.startsWith("request"))  executedMethod.classify( "request", Message.CLASSIFICATIONSTATUS_PLUGIN_BASED_CLASSIFICATION );
				else if(string.startsWith("respond"))  executedMethod.classify( "response", Message.CLASSIFICATIONSTATUS_PLUGIN_BASED_CLASSIFICATION );

				// Raw log writer extension component has written additional data to the raw log.
				// This data can be used when the message type is classified.
				Object msgType=executedMethod.getMethodEntry().getRaw LogExtensionData("MSGTYPE");
				if(msgType!=null && msgType.equals("failure")){
						executedMethod.setMessageType(Message.TYPE_ILLEGAL_REQUEST);
				}
		}
}
```

Figure 20. An example code snippet of a message classifier implementation.

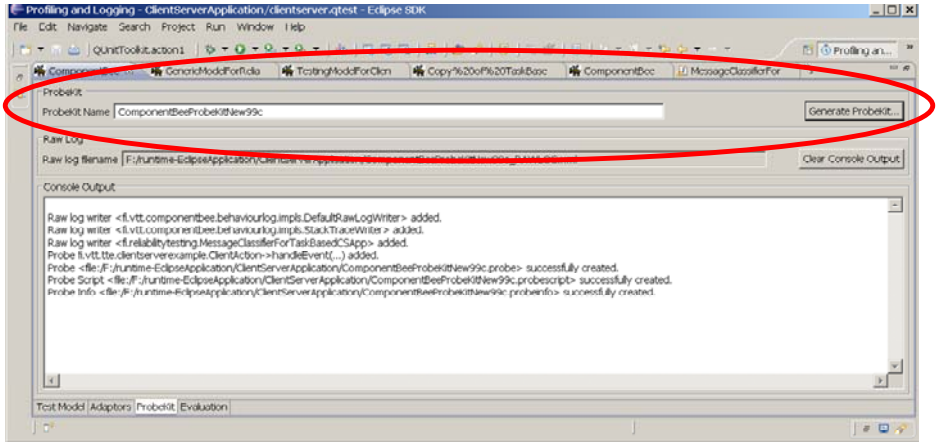


Figure 21. The ComponentBee is capable of generating a Probekit for the test model.

4.4 Behaviour test execution

The profiler tool of the Eclipse uses the generated ProbeKit and collects data about dynamic behaviour of Java components. In order to profile the target software, select "*Profile Configurations*" (Figure 22). The profiling is then done in the following steps:

1. Create a new profile configuration in the profile view (Figure 23) and then define a project and main class for the test bed that you are going to use in profiling.

2. Add the **commonrunner.jar, java.runner.jar,** and **componentbeetoolkit.jar** packages to the class path before profiling (Figure 24).
   If you use your own RawLogWriter extensions in profiling, add these plug-ins to a JAR package and insert the JAR package to the class path in the profiling view of the Eclipse (see Figure 24).

3. Select the profiling tab (Figure 24), click the "*Java profiling*" item, and define a filter set for the application to be profiled (Figure 25). Insert the following rows to the beginning of the filter set:

	CLASS	METHODNAME	RULE
a.	*	start	INCLUDE
b.	java.lang.Thread	start	INCLUDE
c.	java.lang.Object	*	INCLUDE

   NOTICE! It is important that the filter set is like in the Figure 26, in order to ensure that the method calls related to thread starting (*start* method calls) and synchronization (*wait*, *notify*, and *notifyAll* method calls) are recorded to the raw log. In addition, make sure that the methods of the components under testing are not excluded in the filter set.

4. Select the monitor tab (Figure 25) and double click the "*probe insertion*" item. Select the Probekit that you have generated with the ComponentBee and click the "*Finish*" button (Figure 27).

5. Try to connect to the server that is used in profiling by clicking the "*test availability*" button of the monitor tab. If the connection opening is failed to the server, you must start the server (see, troubleshooting section).

6. Finally, after the profiling settings are made, click the "*Profile*" button. The Eclipse will show the profiling monitor view now (Figure 28).

7. After the test sequence is executed, click the "*Stop*" button in the profiling monitor view (Figure 28). The raw log file is now available in the test project path.

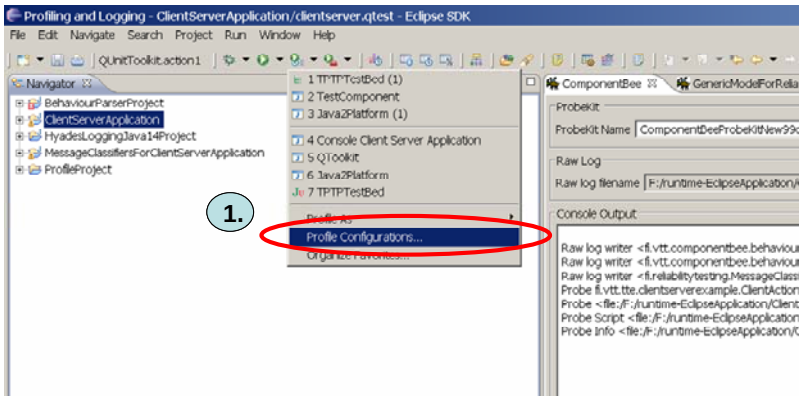


Figure 22. A profile configuration must be defined before profiling.

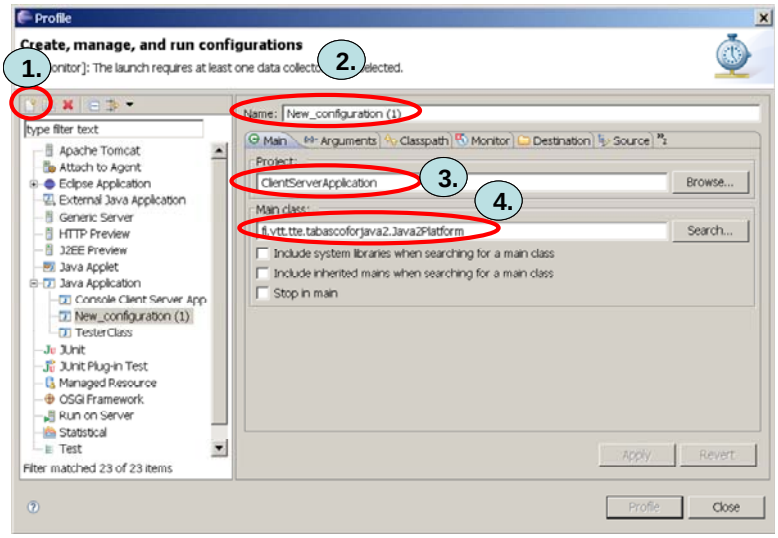


Figure 23. The profile dialog of the Eclipse.

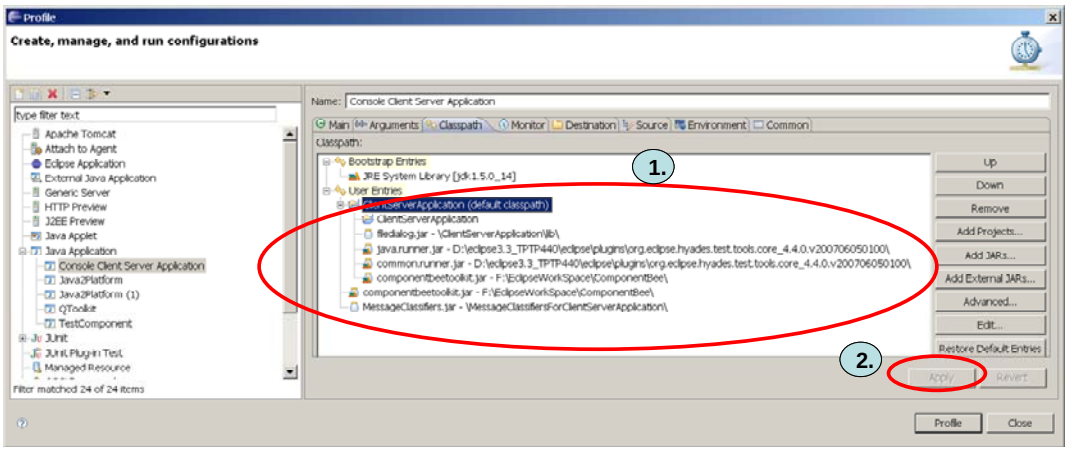


*Figure 24. The **commonrunner.jar**, **java.runner.jar**, and **componentbeetoolkit.jar** packages must be added to the class path before profiling.*

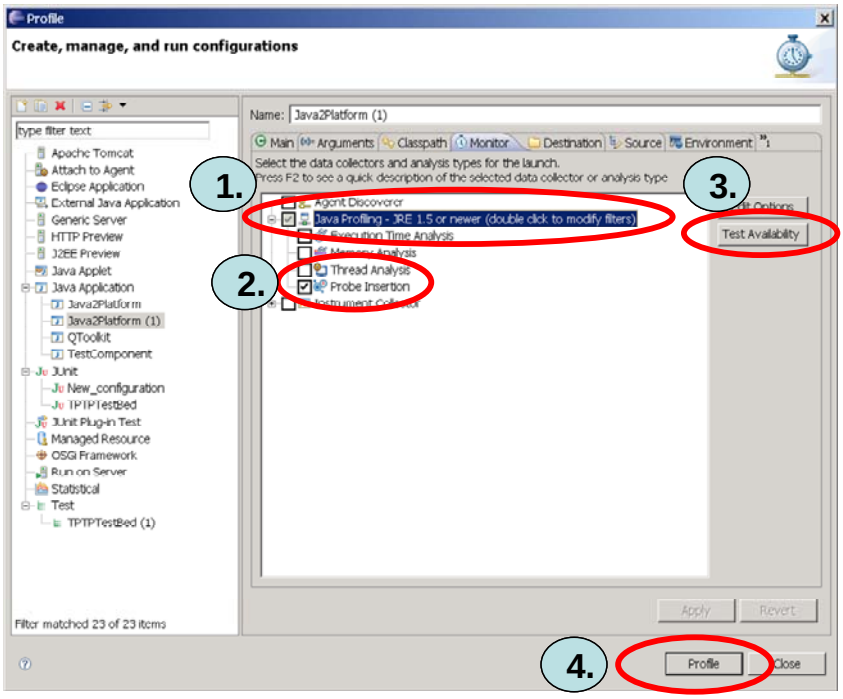


Figure 25. The monitor tab of the profile dialog.

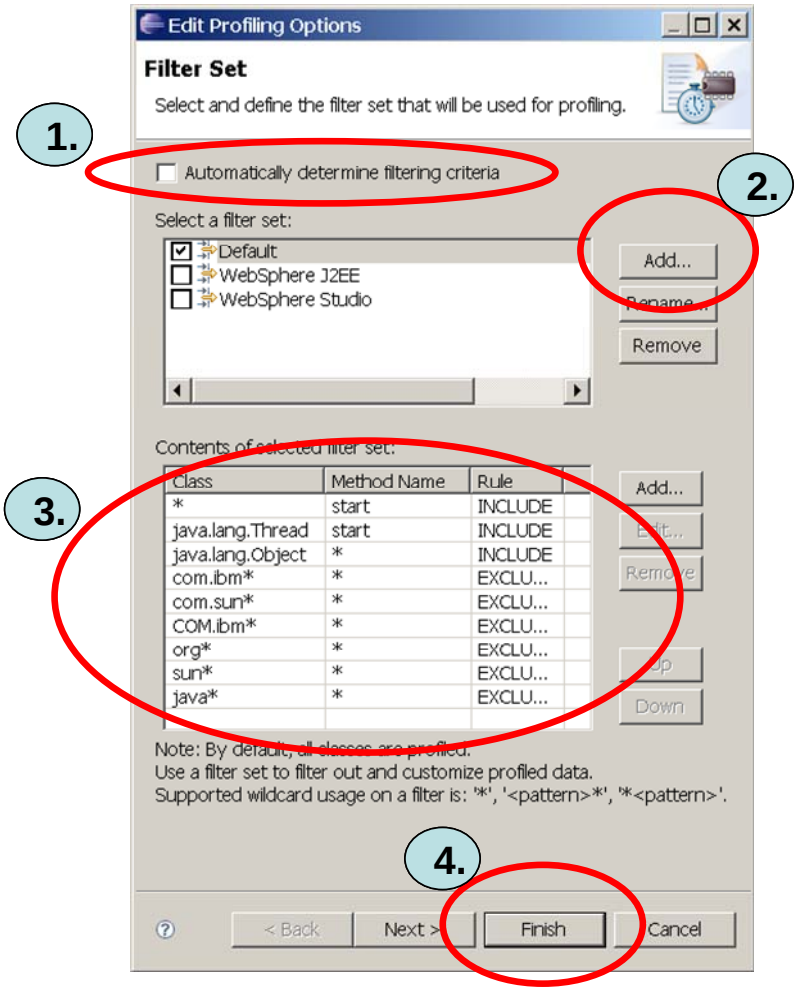


Figure 26. Edit profiling options dialog.

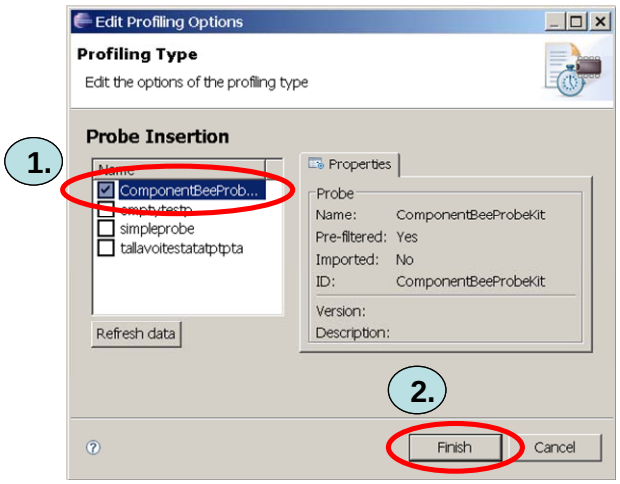


Figure 27. Edit profiling options dialog.

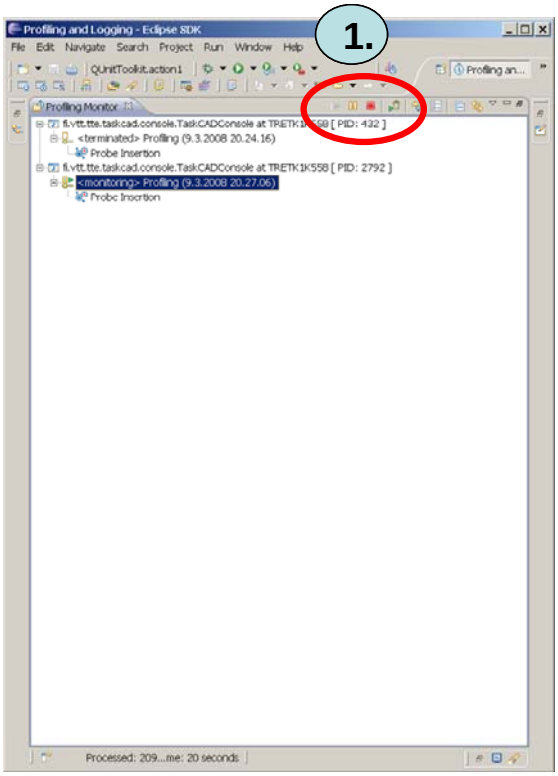


Figure 28. The Profiling monitor view of the Eclipse.

4.5 Evaluation of the dynamic behaviour of the task-based client-server application

The ComponentBee uses pre-processor components (actually a pipe of preprocessors) in raw log data evaluation. Raw log data is evaluated in the following steps:

1. **Generation of a behaviour parser.** The ComponentBee uses a BNF-based parser for extracting behaviour patterns from the collected log. Before the raw log can be analysed, you must select a Java project for the behaviour parser and then click the generate parser classes button (Figure 29) in order to generate a behaviour parser for the test model.

2. **Creation of evaluator plug-ins.** The reliability evaluation requires plug-ins capable of recognizing the failed messages and to define which SW components have caused failures in the executed use cases. The generated *BehaviourParser* extracts a behaviour patterns from the pre-processed raw log data. *Evaluators* navigate in the behaviour pattern tree, evaluate and add evaluation data to the behaviour patterns, and finally calculate reliability values for the components and executed use cases. The *abstract use case evaluator* plug-in of the ComponentBee is capable of calculating and finally recording p_{ij} values to the test report. Before this plug-in can be used, we must extending the *abstract use case evaluator* plug-in with methods that evaluate the messages related to the use case and identifying the components that have participated in or caused failures in the tested use cases. The evaluator plug-ins are attached to the test model and later invoked to calculate p_{ij} values for the SW components and tested use cases.

3. **Creation of a test report.** Next, (if needed) select the parser class path (the generated parser class path is set automatically) and raw log file to be evaluated. Click the create test report button (Figure 29). If the analysis is successfully completed, the measured evaluation results are shown and an analysis report file is finally written in an XML format to the defined path.

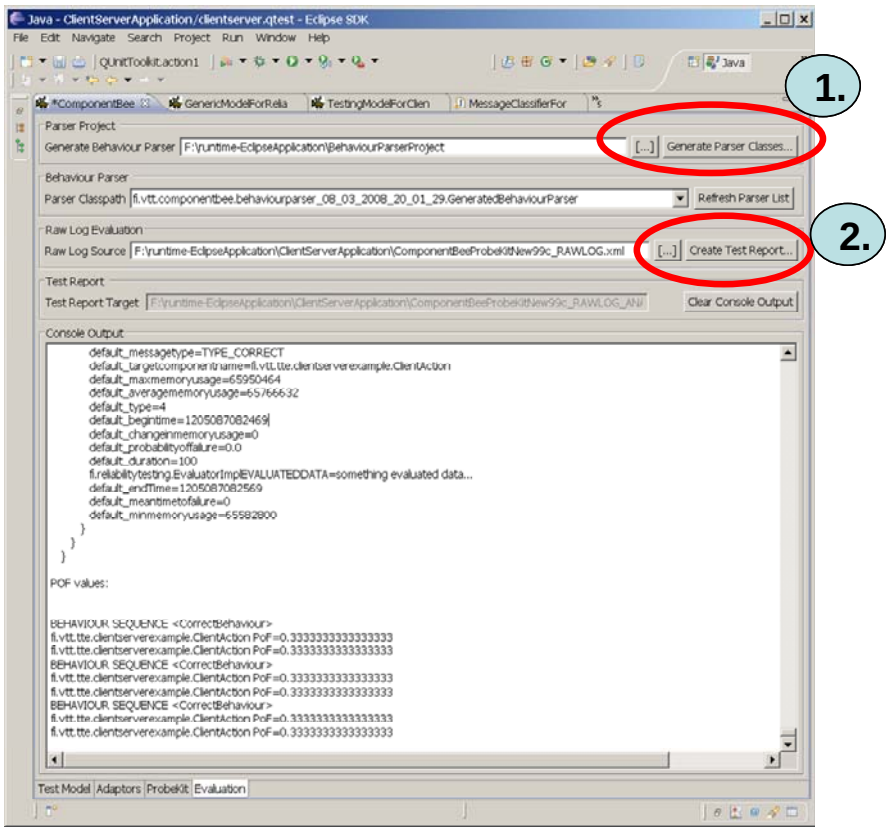


Figure 29. The raw log data is evaluated in the Evaluation view of the ComponentBee.

5 Troubleshooting

This section lists known bugs or problems that affect the usage of the ComponentBee:

1. **An old diagram file can be opened with the visual test model editor of the ComponentBee but adding new elements to it causes a stack overflow exception.** The component diagram must always be initialised with the *initialise test model diagram* menu (Figure 30) before it can be edited with the visual test model editor of the ComponentBee. The future releases of GMF may fix this bug.

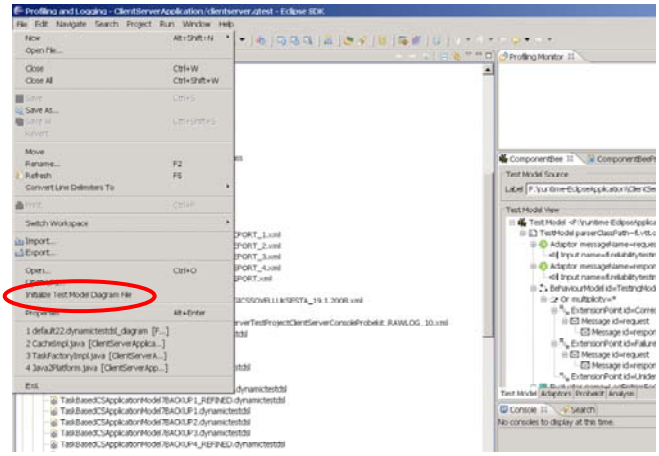


Figure 30. A new diagram file must be initialized for a test model before it can be edited with the visual editor of the ComponentBee.

2. **The adaptor view does not show the available SW components.** The project>clean will rebuild the Java classes. Do this and then open the popup menu in the adaptor view and click "*fetch available classes*"- menu item.

3. **Problems in profiling with probes.** The following issues may cause errors in profiling:

    1. **Different JRE versions.** If you use Java profiler and probes, **the same version of JRE must be used** in the both Eclipse environment and in profiling environment (=> Separate JRE versions may prevent profiling). JDK1.5.0_14 seems to work well with the Eclipse profiler and TPTP's ProbeKits.

    2. **Required classes are missing.** The **commonrunner.jar, java.runner.jar, and componentbeetoolkit.jar** libraries must be added to the class path before profiling. Select the project properties and add these libraries to the Java build and class paths (Figure 31 and Figure 32). **NOTICE!** If you use your own *RawLogWriter* extensions in profiling, add these plug-ins to a JAR package and insert the package to the class path.

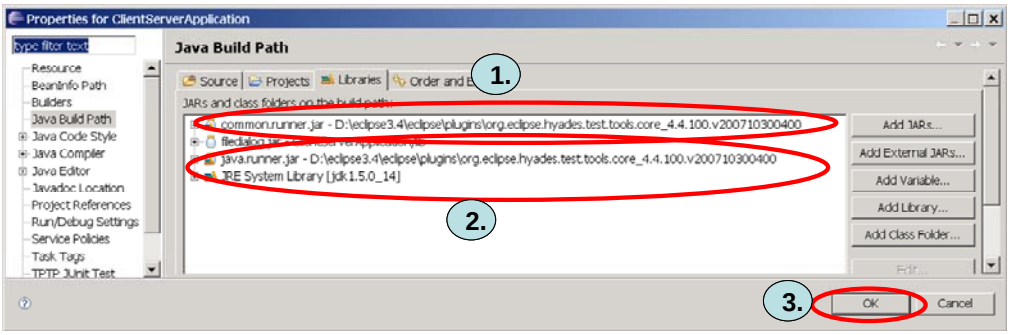


Figure 31. The project properties dialog of the Eclipse.

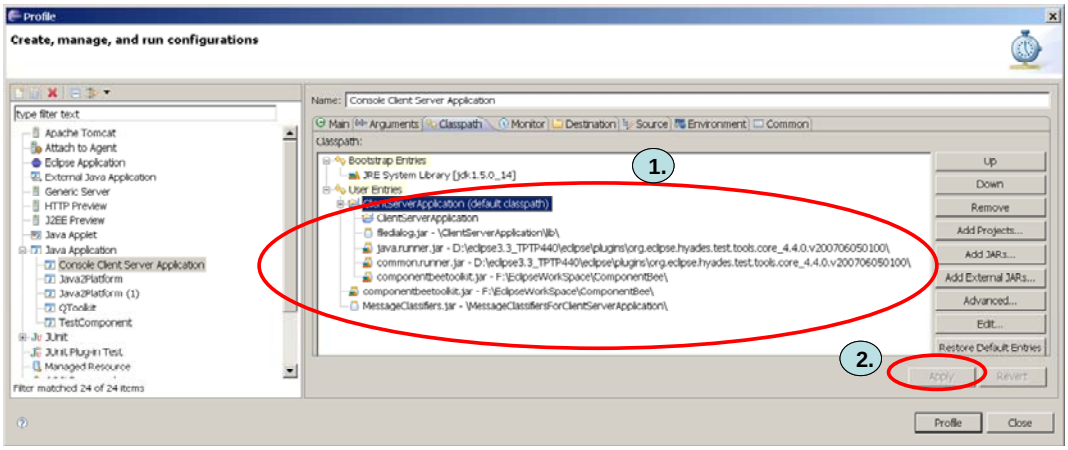


*Figure 32. The **commonrunner.jar**, **java.runner.jar**, and **componentbeetoolkit.jar** packages must be added to the class path before profiling.*

3. **Connection opening is failed to the server used in profiling when the *"Test availability"*-button is pressed in the profiling view of the Eclipse**. In order to utilise probes, you must start the ACServer.exe at the beginning (the directory is win_ia32* in Windows XP) (Figure 33). This must be done only if the ACServer.exe is not already running.

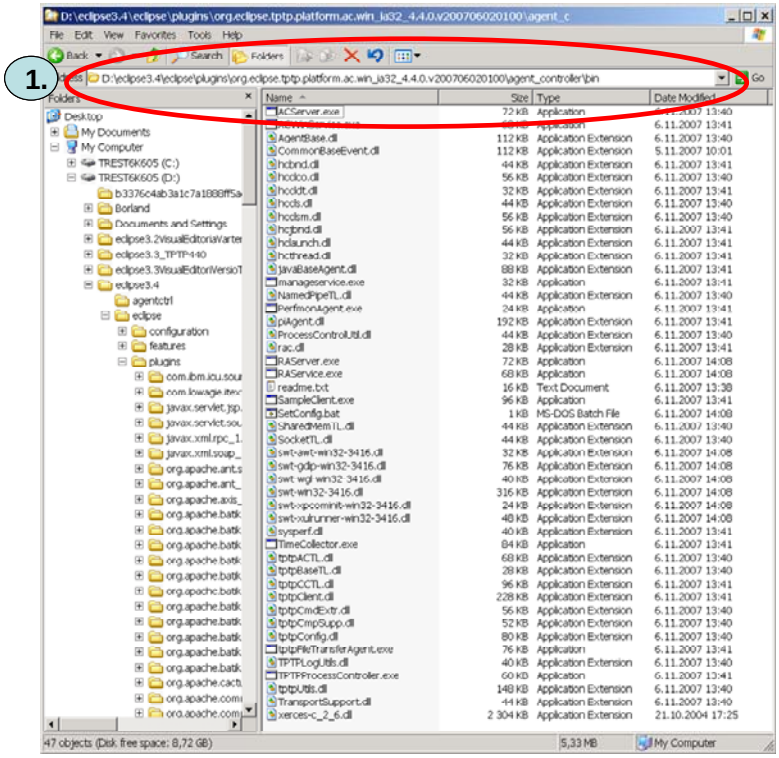


Figure 33. In order to utilise probes, you must start the ACServer.exe at the beginning (the directory is win_ia32 in Windows XP). This must be done only if the ACServer.exe is not already running.*

4. ***"An application for the process that was about to be launched is not registered with the Agent Controller. This can be caused by a missing or corrupted configuration file."*-failure (**Figure 34**).** Attaching the org.eclipse.tptp.* JAR libraries to the build path will cause this failure.

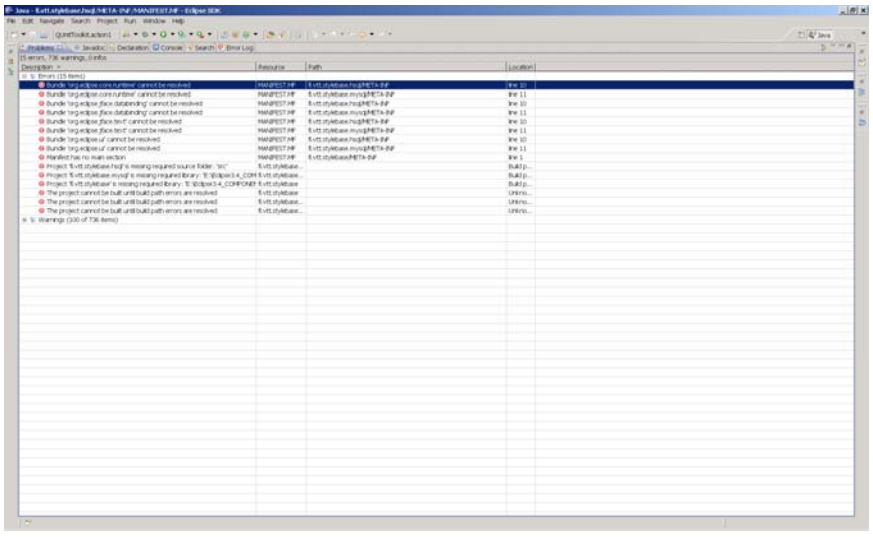
Figure 34. A failure that arises if the org.eclipse.tptp. JAR libraries are attached to the project build path.*

5. **Profiling of Eclipse plug-ins is not possible.** Attaching the org.eclipse.tptp.* packages to the build path will cause this problem. There is a problem in the OSGI implementation causing a null pointer exception in the SystemBundleLoader class at:

   org.eclipse.osgi.framework.internal.core.SystemBundleLoader.<init>(SystemBundleLoader.java:65)

   This problem prevents the launching of an Eclipse instance to be profiled.

6. **Bundle can not be resolved.** The build path can contain multiple JAR libraries containing the same bundle. Thus, it is not possible to resolve the JAR library that should be used. The Eclipse workspace can go mixed and the Eclipse does not necessarily find the plug-ins (does not show new plug-ins in the dependencies view when the plugin.xml is opened). Remove .metadata directory and define the required projects again. Finally, check that the "*Build Automatically*" option is checked in the Eclipse.



7. **The assert statements in the program under test.**

   Go to Window -> Preferences -> Java -> Compiler -> Compliance and Classfiles and set the Compiler Compliance Level to 1.4. This tells the compiler to recognize and allow assert statements, but does not enable them.

In the Eclipse 3, go to Window -> Preferences -> Java -> Compiler -> Compliance and Classfiles and set:

Compiler Compliance Level: to 1.4
Use default compliance settings to unchecked
Generated .class files compatibility: to 1.4
Source compatibility: to 1.4
Disallow identifiers called 'assert': to Error
Compiler Compliance Level to 1.4

To enable (make active) assert statements, you must set a flag to the compiler.

1. Go to Run -> Run... -> Arguments, and in the box labelled VM arguments,
2. Enter either -enableassertions or just –ea, and
3. Accept the changes and close the dialog.

References

[ImP07] Immonen, A. and M. Palviainen. *Trustworthiness Evaluation and Testing of Open Source Components*. in *The 7th International Conference on Quality Software (QSIC 2007)*. 2007. Portland, Oregon, USA.

[Pal07] Palviainen, M. Task-based composition of the context-sensitive UIs of physical environments. Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAS 2007). 2007. Athens, Greece.

[RSP03] Reussner, R.H., Schmidt, H.W., and Poernomo, I.H., Reliability prediction for component-based software architectures. Journal of Systems and Software, 2003. Vol.66,(No. 3) Pp: 241-252.