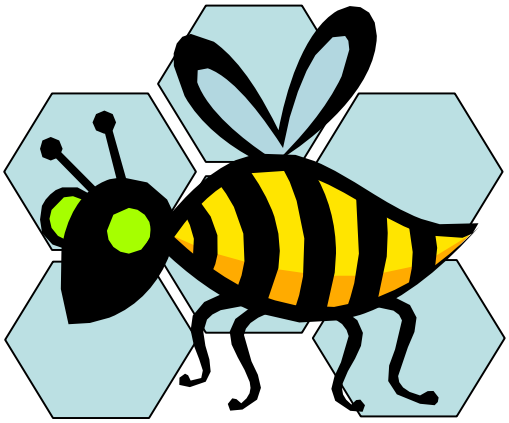
ComponentBee v. 1.0beta



Example

Reliability testing of a client-server application

Draft 30.9.2008

1 The client-server example

The client-server example package contains a small example that illustrates how the ComponentBee can be utilised in reliability evaluation of a client-server application. The usage of Web services requires utilization of network connections and a Web server. Network requests may take several seconds and cause delays, decreasing the usability of the application. Thus, these requests should be executed asynchronously in threads, in order to prevent them from blocking the usage of the UI of the application. Unfortunately, the utilization of threads and Web connections increases the complexity and may decrease the reliability of the SW system.

Figure 1 shows execution paths for a client-server application. The asynchronous content delivery is an execution path where the client requests the server to deliver a correct or failure response for the client. The response is finally shown in the refreshed view for the user. In an *ignored request* use case the server-side does not deliver a response for the request. In the *ignored response* use case the received response is not shown for the user.

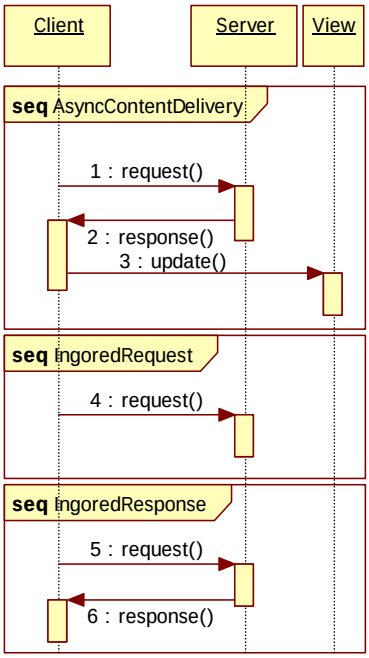


Figure 1. Execution paths for a client-server application.

2 Creation of a unit-level reliability test

2.1 Import the client-server project to the Eclipse workspace

Start the Eclipse first. The *ClientServerExample.zip* package provides a Java project that contains a ready-made test bed for the client-server application. Import the project to the Eclipse workspace first. Select **File->Import…** and then choose *import existing projects into Workspace* item (Figure 2).

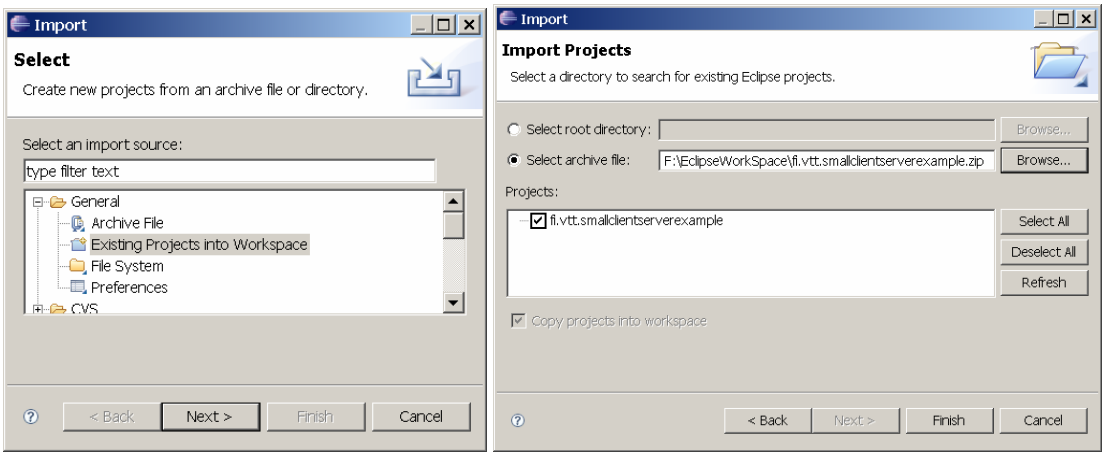


Figure 2. Importing the client-server example to the Eclipse workspace.

After the project is imported, select the project properties and include **common.runner.jar**, **java.runner.jar,** and **componentbeetoolkit.jar** libraries to the Java build path (Figure 3).

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

Notice! The probes are used in raw log data recording. The utilisation of probes requires that the Java classes are in the root folder 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 3), select the default src/ folder, and click remove. Push the "Add Folder" button now and 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.

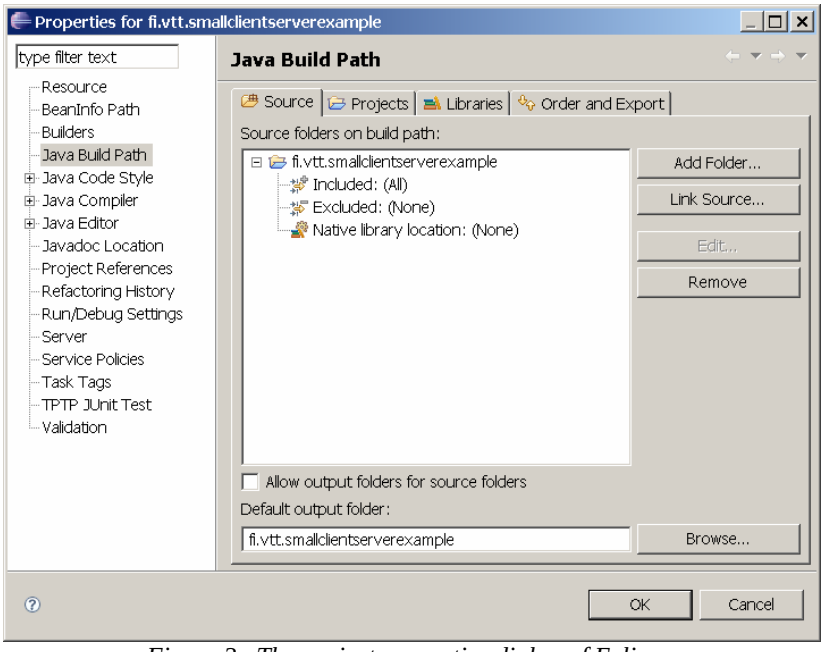
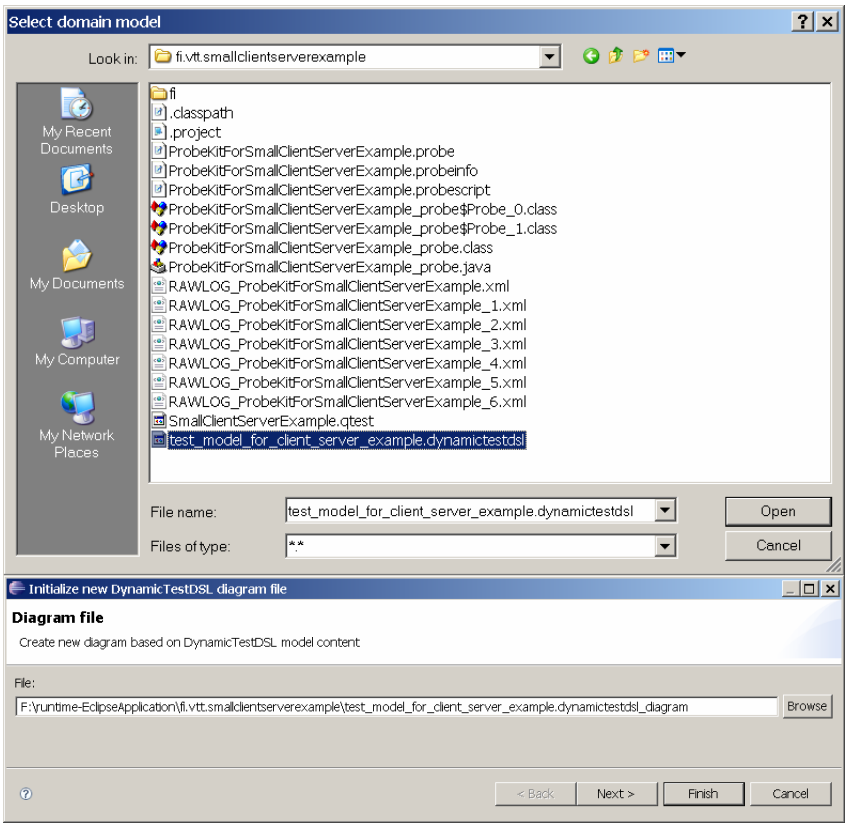


Figure 3. The project properties dialog of Eclipse.

2.2 Creation of a visual diagram for the test model

In order to test the reliability we must have a test model for the client-server application. The ComponentBee offers a visual editor for test models defining expected behaviours (named message sequences) for the components under tests and adaptor elements to adapt the behaviour model for the actual SW components. The project provides a ready-made test model (*test_model_for_client_server_example.dynamictestdsl*) for the client-server application. Initialise a visual diagram for the test model. Select **File->Initialise Test Model Diagram file**, choose the *test_model_for_client_server_example.dynamictestdsl* file in the select domain model dialog, and finally click **"Finish"**. The visual editor will now show the test model (Figure 4).



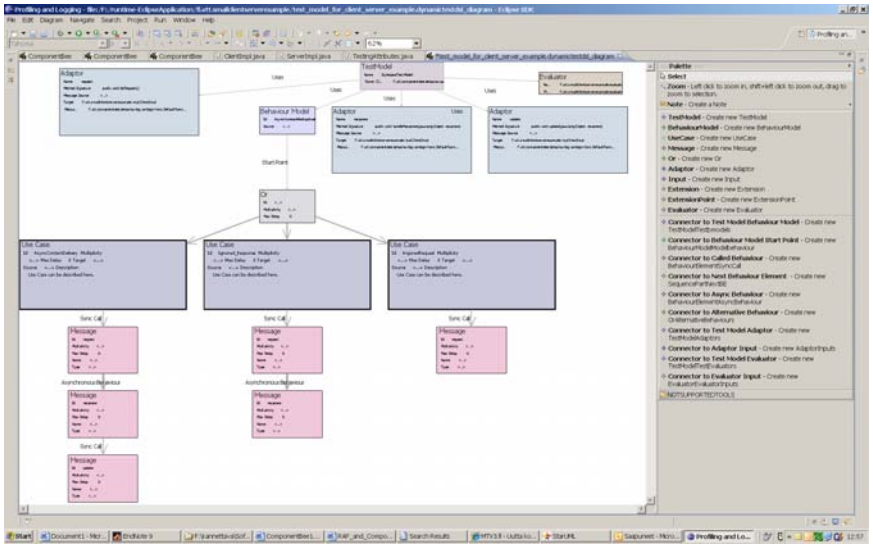


Figure 4. Creation of a visual diagram file for the test model.

2.3 Test model adaptation

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

The behaviour test editor provides a tree view for the available software 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. The adaptors view shows the methods of available software interfaces and components. By selecting a method and by clicking a right mouse button it is possible to open a popup menu and create an adaptor element for a message defined in a test model (Figure 5).

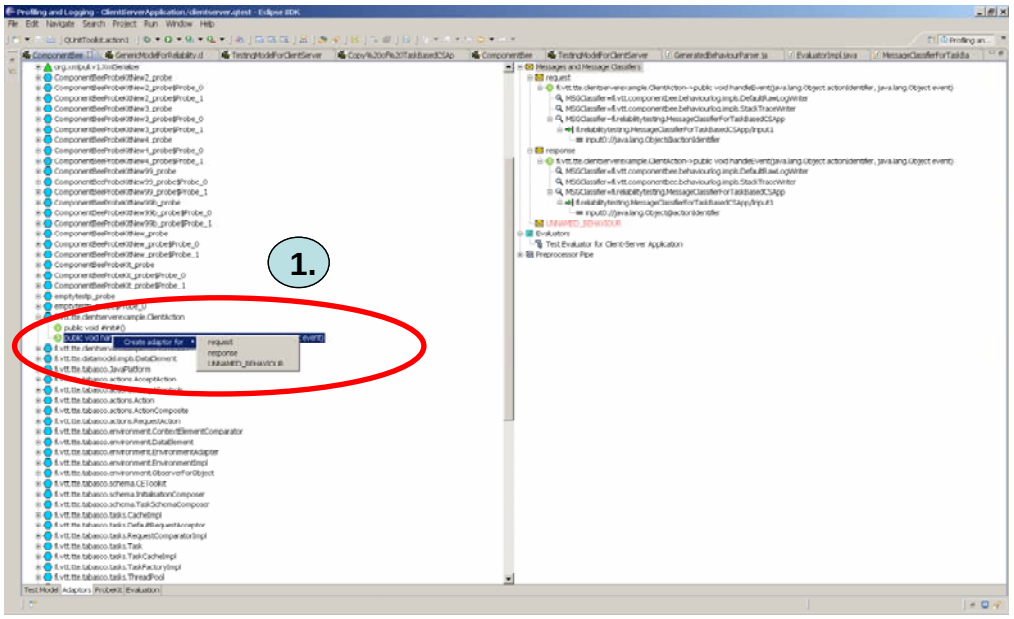


Figure 5. Adding an adaptor element for a message.

The size of the raw log can be large, if all the (e.g. state) information of the dynamic behaviour of SW components is recorded to the log. 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. 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.

Raw log writers are defined in the adaptor elements of a test model. The right-side of the view displays 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 6). These plug-ins are used later when the raw log is recorder and evaluated.

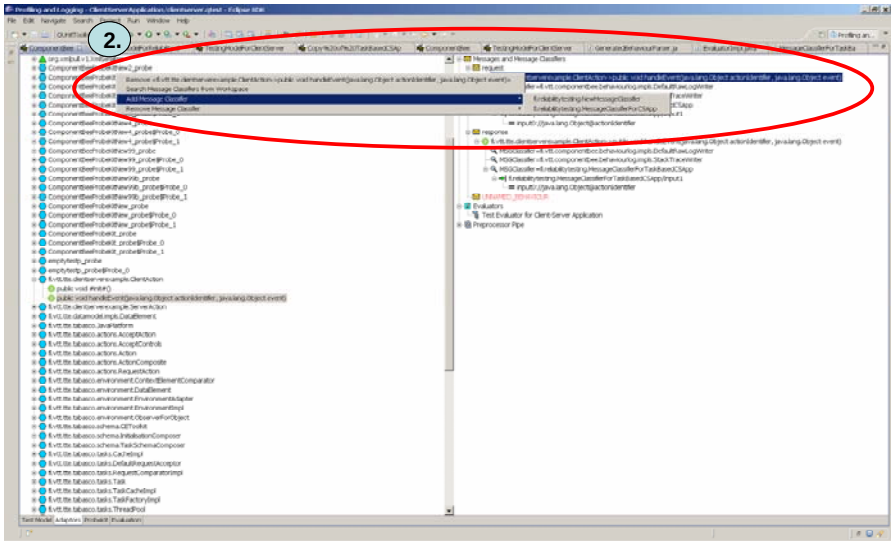


Figure 6. Adding a raw log writer or message classifier plug-in for an adaptor element.

A message classifier defines identifiers for data that it needs in message classification. The *input elements* can define data sources for the data identifiers and thus configure message classifiers to classify messages delivered between different kinds of software components. In the adaptor view it is possible to insert new input elements to message classifiers.

The test model example defines the required raw log writer plug-ins for the selected methods.

3 Recording raw log data about the dynamic behaviour of SW components

The probabilities for different execution paths are configured with the attributes that are defined in the *fi.vtt.smallclientserverexample.TestingAttributes* interface. Thus, by changing these attributes it is easy to effect to the “reliability” of the client-server application.

In the ProbeKit view of ComponentBee it is possible to generate a ProbeKit for the test model (Figure 7). The profiler tool of the Eclipse inserts the ProbeKit’s probes at the entry and exit of the selected methods of

the components and then runs the instrumented Java program. The probes will now monitor the execution of the program execution and call the raw log writers to add data to the raw log file.

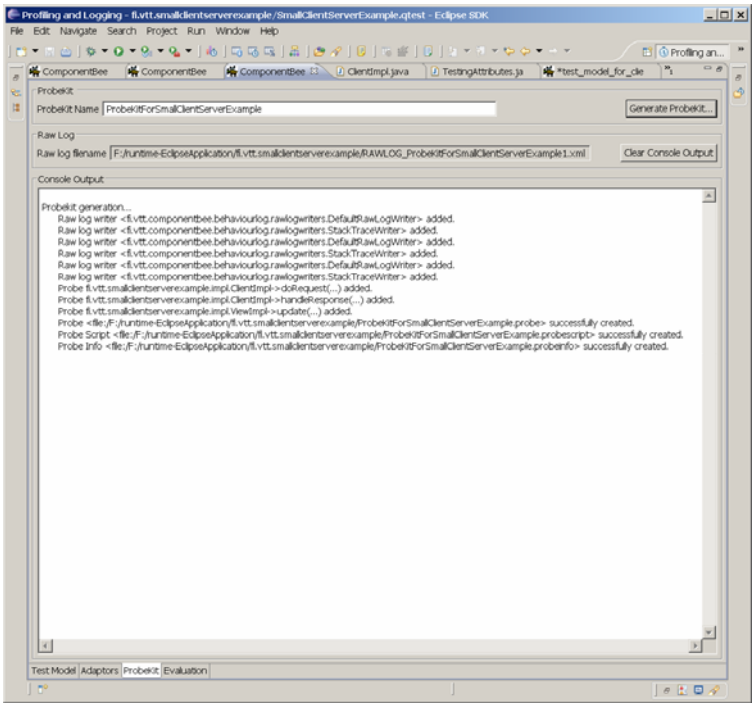


Figure 7. A ProbeKit is generated with the ComponentBee.

In order to record raw log data about the target software components, select "*Profile Configurations*" (Figure 8).

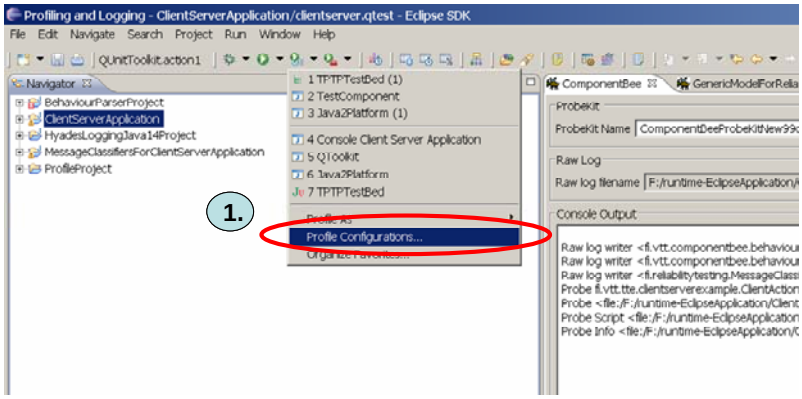


Figure 8. A profile configuration must be defined before profiling.

Create a new profile configuration in the profile view (Figure 9) and then define a project and main class for the test bed that you are going to use in profiling.

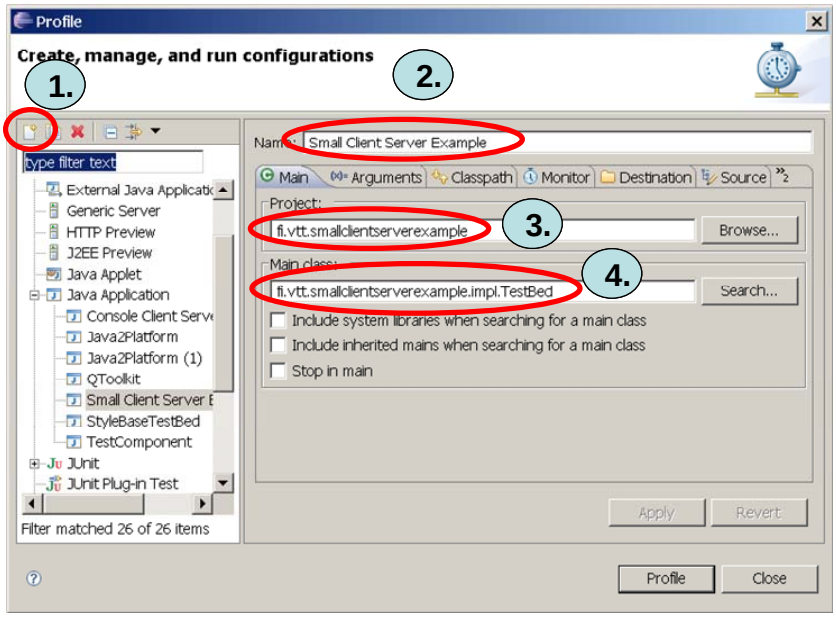


Figure 9. The profile dialog of Eclipse.

The **commonrunner.jar, java.runner.jar, and componentbeetoolkit.jar** packages must be added to the class path before profiling (Figure 10).

NOTICE! If you use your own raw log writer plug-ins in profiling, you must add these plug-ins to a JAR package and insert the JAR package to the class path in the profiling view of Eclipse (see Figure 10).

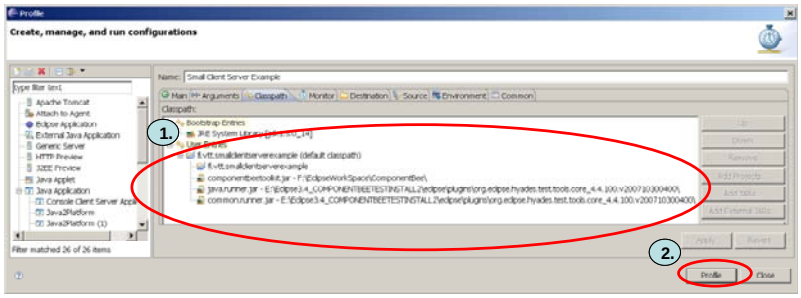


*Figure 10. The **commonrunner.jar**, **java.runner.jar**, and **componentbeetoolkit.jar** packages must exist in the class path.*

Select the profiling tab (Figure 11), click the "*Java profiling*" item, and define a filter set for the application to be profiled (Figure 12).

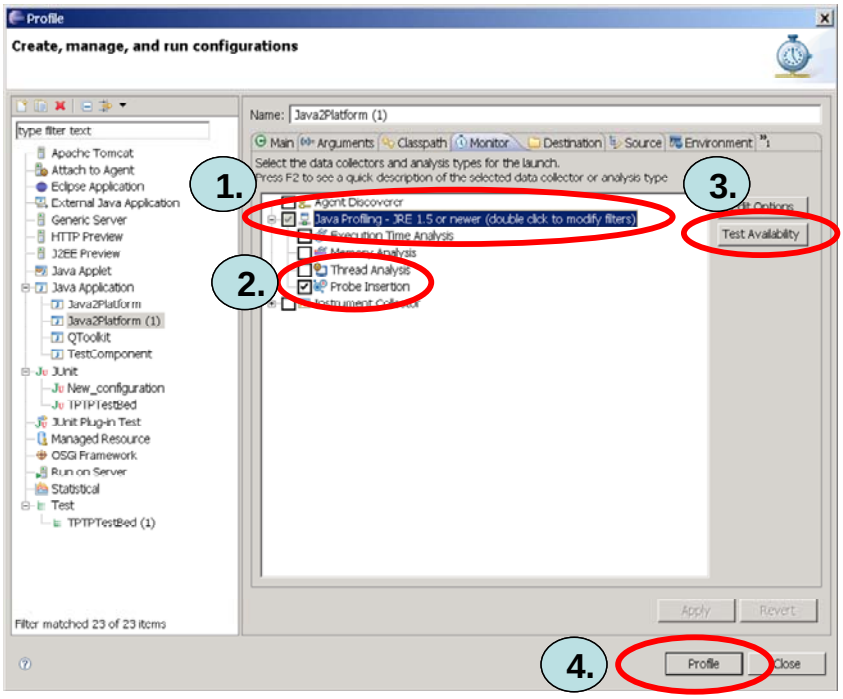


Figure 11. The monitor tab of the profile dialog.

NOTICE! It is important that the filter set is like in the Figure 13, 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.

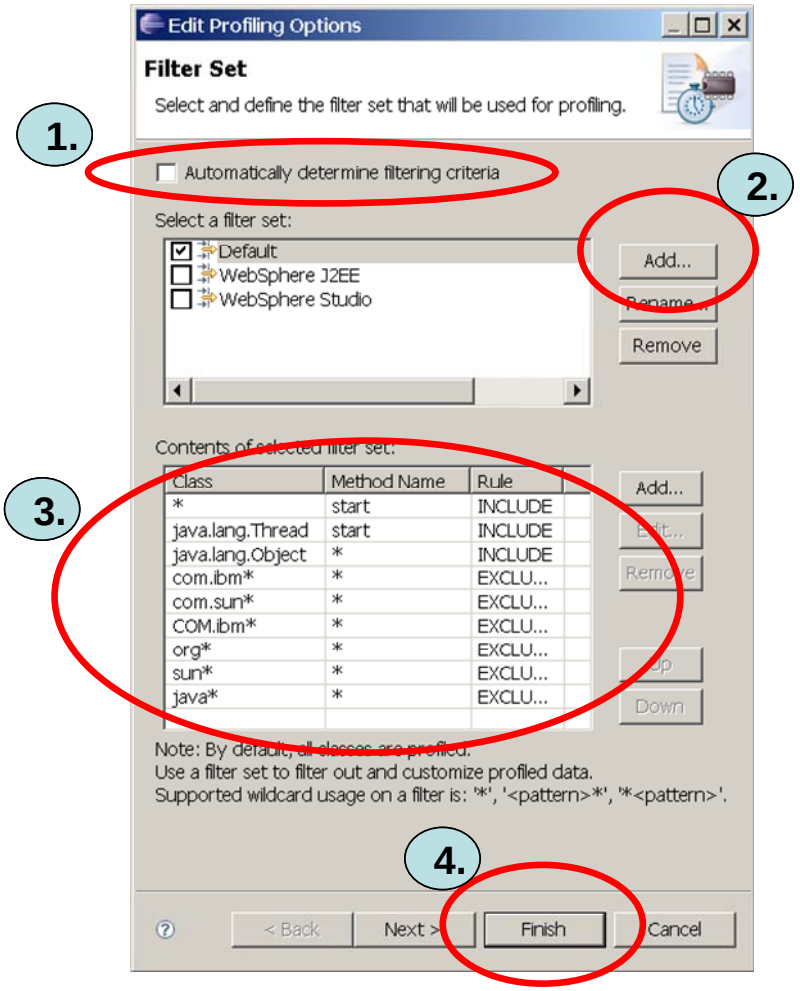


Figure 12. Edit profiling options dialog.

Then, select the monitor tab (Figure 11) and double click the *"probe insertion"* item. Select the probekit that you have generated with the ComponentBee and click "*Finish*" button (Figure 13).

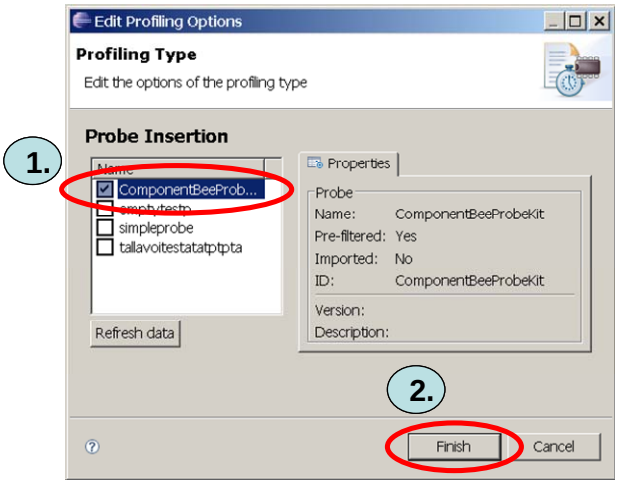


Figure 13. Edit profiling options dialog.

Finally, you can 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 in the usage instructions of the ComponentBee). Finally, after the profiling settings are defined, click the "*Profile*" button. The Eclipse will show the profiling monitor view now. Click the stop button in the profiling monitor view after the test sequence is executed (Figure 14). The raw log file is now available in the test project path.

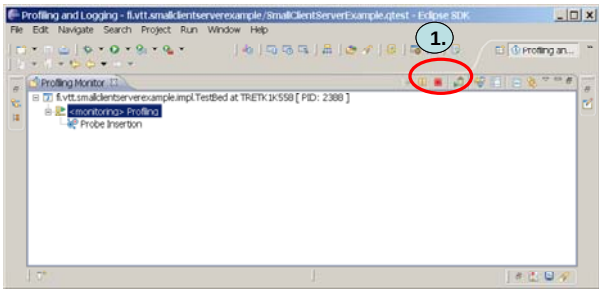


Figure 14. The profiling monitor view of Eclipse.

4 Evaluation of the raw log data

Unit-level reliability testing measures Probability of Failure values (the 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. Thus, 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 calculates the measured p_{ij} values (PoF_M values).

The ComponentBee uses pre-processor components (actually a pipe of pre-processors) in raw log data evaluation. The raw log data is evaluated in the Evaluation view of the ComponentBee (Figure 17). 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 (Figure 17). 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 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 example package provides an evaluator plug-in that extends the abstract use case evaluator plug-in and recognizes the components that have caused failures in the tested use cases (Figure 15).

```
package fi.vtt.smallclientserverexample.evaluators;

import fi.vtt.componentbee.behaviourlog.BehaviourPattern;
import fi.vtt.componentbee.behaviourlog.Evaluator;
import fi.vtt.componentbee.behaviourlog.Message;
import fi.vtt.componentbee.behaviourlog.ProcessingMonitor;
import fi.vtt.componentbee.behaviourlog.abstractevaluator.AbstractPatternEvaluator;
import fi.vtt.componentbee.behaviourlog.abstractevaluator.UseCaseEvaluationResult;

public class EvaluatorForSmallClientServerApplication
                        extends AbstractPatternEvaluator
                        implements Evaluator{
        /**
         * Returns true if the input values that are provided in the message are correct.
         * If not the message sender is set to be a failure participant in the use case.
         *
         * @param message
         * @param useCase
         * @param messageBehaviourPattern
         * @param useCaseEvaluationResults
         * @param processingMonitor
         * @return
         */
        public boolean evaluateInputValues(
                final Message message,
                final BehaviourPattern useCase,
                final BehaviourPattern messageBehaviourPattern,
                final UseCaseEvaluationResult useCaseEvaluationResults,
                final ProcessingMonitor processingMonitor){

                if(message.getMessageName().equals("response"))
                {
                // The server has caused a failure if it does send a failure response.
                String responseValue=(String) message.getInputAttribute(0);
                boolean isFailureParticipant=responseValue.startsWith("failure_response");
                useCaseEvaluationResults.addParticipant(
                                "fi.vtt.smallclientserverexample.impl.ServerImpl",
                                isFailureParticipant);
                }
                if(useCase.getName().equals("IngoredRequest"))
                {
                                // The server is caused a failure if it does not deliver a
                response.
                                useCaseEvaluationResults.addParticipant(
                                "fi.vtt.smallclientserverexample.impl.ServerImpl",
                                true);
                }
                return true;

        }
        ...
```

Figure 15. An evaluator implementation for the client-server application.

The evaluator plug-in implementation is attached to the test model and later invoked to calculate p_{ij} values for the SW components and tested use cases (Figure 16).

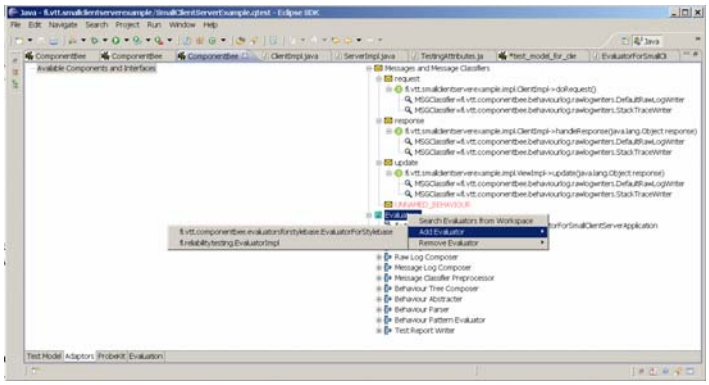


Figure 16. The evaluator plug-in is added to the test model.

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 17). 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.

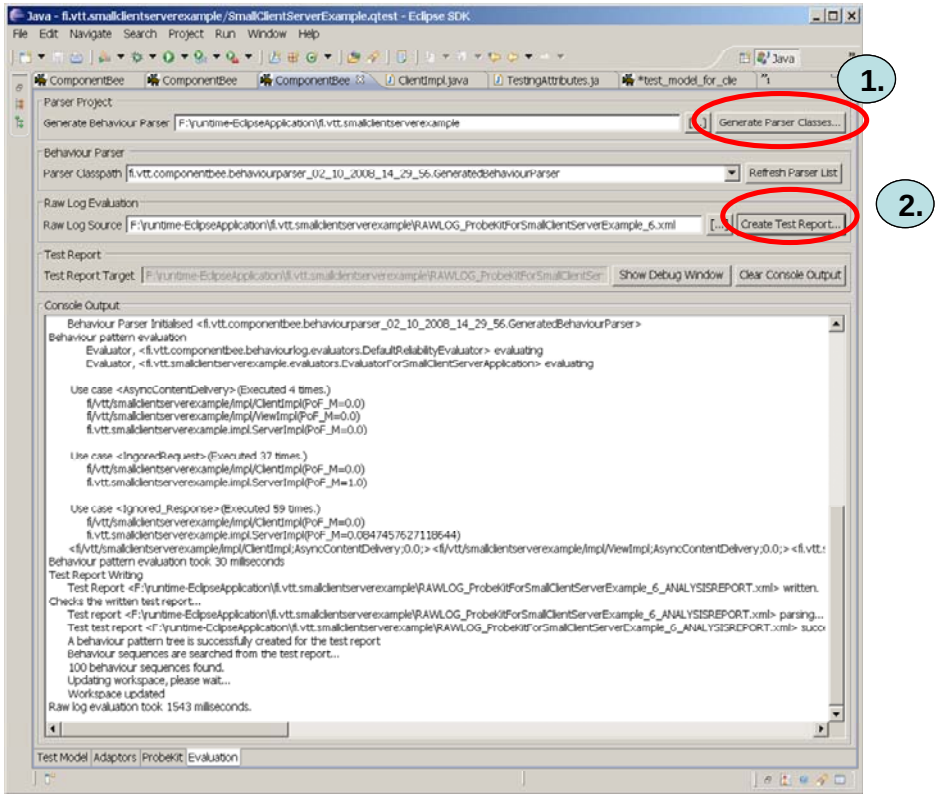


Figure 17. The Evaluation view of the ComponentBee.