CHAPTER VIII

TOOL IMPLEMENTATION & EVALUATION

8.1 Tool Implementation

The concepts discussed so far in the previous chapters are put together in this chapter to build a tool based on the concepts. The tool comprises of three parts: test case generator, instrumentation tool, and message sending sequence verification tool.

The test case generator is for generating test cases from UML sequence diagram written with the extension discussed in chapter 7. It also uses the adequacy criteria in chapter 4 for guiding test case generation. The instrumentation tool covers instrumentation of program under test to produce actual message sending sequence from program execution. The message sending sequence verification tool is for verifying message sending sequences based on the concept in chapter 6. Although the test case generator is independent to programming language, the instrumentation tool and the message sending sequence verification tool, are specific to Java technology.

The tool is implemented in Java. To achieve modular design, each functionality of the tool is designed as a component. Components have dependency on each other only through their interfaces. Spring framework [42] is applied as an IoC (Inversion of Control) framework which wires these components together in runtime using an XML configuration file. With this capability, it is easy to modify or extend the functionality of the tool. The tool is implemented based on JDK version 1.4; therefore, it can operate on any system with Java Runtime Environment version 1.4 or higher.

8.1.1 Test Case Generator

The test case generator is implemented by following the test case generation process shown in figure 8.1. For a test case generation session, a UML sequence diagram of an interaction under test and a polymorphic adequacy criterion must be selected by the user. A single test case generation session generates test cases for only

one interaction and considers only one polymorphic adequacy criterion. The test case generator performs test case generation steps as follows.

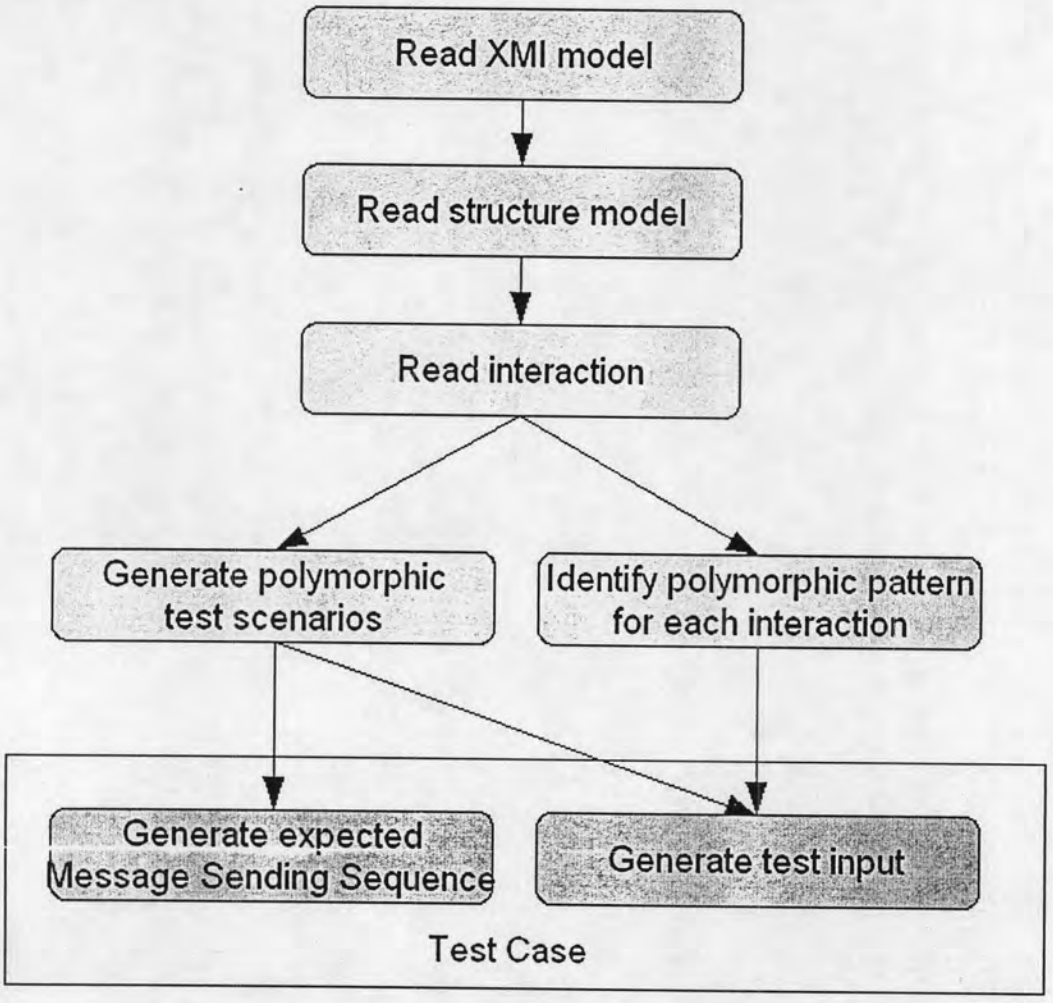


Figure 8.1 Test case generation process used in the tool

1. First the design model is read. Since UML diagrams are essentially human-readable graphical images, it is not easy to process a diagram as it is. XMI (XML Metadata Interchange) [9] is used instead due to its XML-based structure. Most of UML tools support saving or exporting to XMI format. The tool relies on UML 1.4 and XMI version 1.2, which is supported by ArgoUML. ArgoUML is a UML modeling tool, which is also an open source project. It is selected here because it is quite convenience for dealing with UML semantic. The test case generator also utilizes the XMI loading capability from ArgoUML, which uses JMI (Java

Metadata Interface) for working with XMI content. JMI is a Java API for mapping metadata, XMI in this case, to Java class so that it is easy for a Java program to manipulate metadata content.

2. After the XMI file is loaded, the structural model is read. The information about inheritance hierarchy is extracted from the model. The result is stored in the structural part of the message sending sequence model presented in chapter 6. Essentially each class or interface is read and each inheritance relationship between each other is resolved. This results in classifiers (classes, interfaces, and data types) which will be used for constructing expected message sending sequence in later steps.

3. An interaction is read next. A UML sequence diagram is represented as an Interaction element inside a Collaboration element. At this step, the selected Collaboration element is read, and a prototype of the expected message sending sequence is created from the interaction in the Collaboration element. The prototype is a message sending sequence with all information from the interaction; however, it has yet to be adapted for each polymorphic test scenario.

4. Next, two things happens. Polymorphic test scenarios can be generated from the interaction regarding the selected polymorphic adequacy criterion, and at the same time polymorphic assignment patterns can be identified. These two processes are independent to each other. The first process identifies all polymorphic entities in the interaction under test and creates test scenarios according to the selected adequacy criterion based on the concept explained in chapter 4. Each of the test scenarios contains the class name to be included in test for each of the polymorphic entities.

5. Identifying polymorphic assignment patterns includes locating and collecting all tagged values relating to each polymorphic assignment in the interaction under test so that the test case generator would know how to control each polymorphic assignment. Each polymorphic assignment is compared against each of the polymorphic patterns presented in chapter 5, and tagged values are identified

whether they match the usage described in chapter 7. The output of this step is essentially conditions, each for controlling a polymorphic assignment in the interaction.

6. After that test cases can be generated from the output of the previous step. Each test case consists of two parts: the expected message sending sequence and the test input. The prototype of the expected message sending sequence and the test scenarios are put together to form a final expected message sending sequence for each test case. For each test scenario, an expected message sending sequence is created by applying the classes to be tested for the scenario to the prototype of the expected message sending sequence. This will be used for evaluating test result after test execution. The final expected message sending sequence for each test case is serialized and written to an XML file using JavaBeans XML serialization mechanism.
7. Finally the test input is generated from test scenarios and the result from identifying polymorphic assignment patterns. For each test scenario, a set of classes to be tested is known. From the set and the polymorphic assignment patterns, the conditions which objects of the classes to be tested are assigned to the polymorphic entities are known. The conditions can be used for test input data generation. In this tool, data are generated randomly and evaluated against the conditions. If the conditions are not satisfied, the data are re-generated. This step results in test input for each test case. The generated test cases are written as a set of report files in HTML format.

Figure 8.2 shows a UML component diagram expressing components in the test case generator. As state earlier, dependency between components is at the interface level, and IoC is applied. Therefore, it is possible to introduce additional components for new implementation with minimum effect to the existing components.

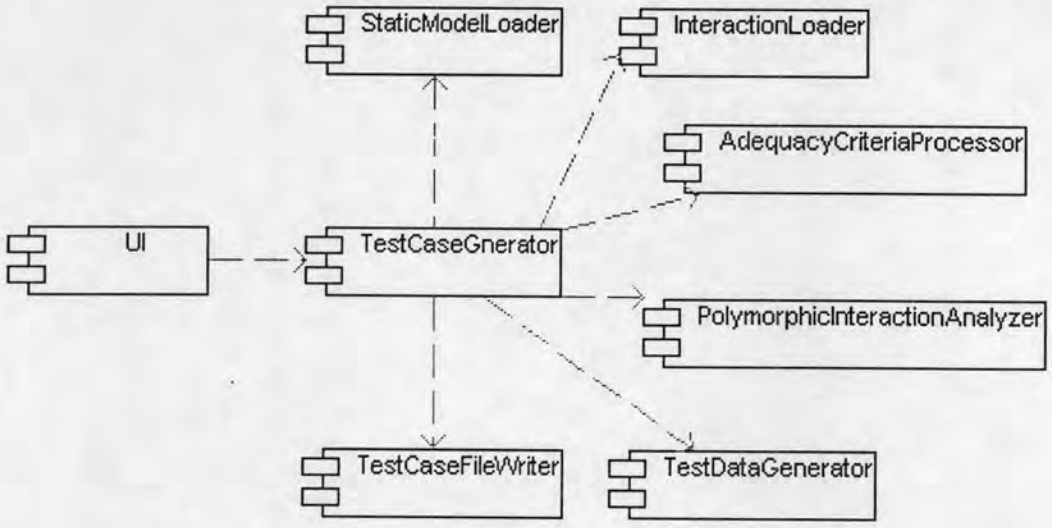


Figure 8.2 UML component diagram of the test case generator

Figure 8.3 shows the user interface of the test case generator. From the user interface, an XMI for the model is selected, and the name of the Collaboration element for test case generation is specified. Also an adequacy criterion is selected from the list of available adequacy criteria. Finally, the directory for writing the result from test case generation is selected.

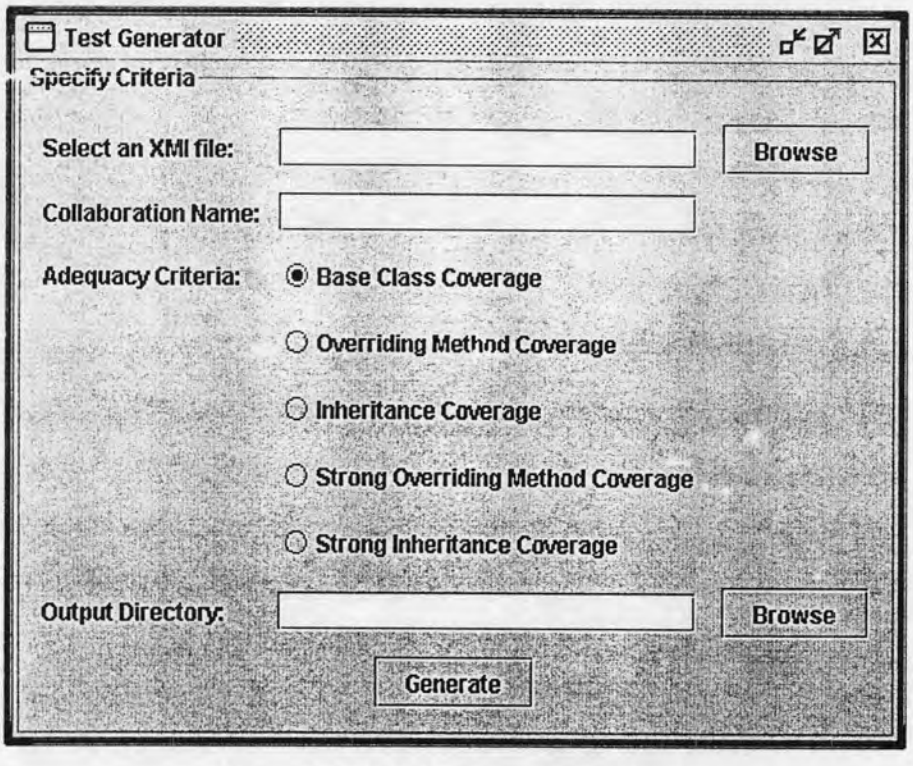


Figure 8.3 UI of the test case generator

8.1.2 Instrumentation Tool

The tool also provides instrumentation support for using with a Java program. AspectJ [41], an AOP implementation for Java, is utilized for Java program alteration. In AspectJ, BCEL (Bytecode Engineering Library) [43], which is capable of manipulating Java class in binary format, is used. In the tool, an aspect is provided for weaving into any Java program. The output of the weaving process is an instrumented program. During execution, an instrumented program generates a message sending sequence, which reflects the actual object interaction under the current execution. After the execution of the instrumented program completes, the generated message sending sequence is written to an XML file using JavaBeans XML serialization mechanism, which is the same mechanism used in writing the expected message sending sequence from test case generation.

However, an instrumented program needs to be informed when to start capturing the message sending sequence and when to stop so that the captured message sending sequence is written to a file. Moreover, the name and the location of the file must be specified. To solve this issue, a test harness is implemented as a part of the tool. The test harness is a Java program in command line mode, which takes two arguments: the name of test case class and the name of output file. A test case class is a class which contains an implementation of a test case. It must implement the test case interface, th.ac.chula.testgen.mss.instru.TestCase, which is introduced in this tool. The test harness instantiates an object of the test case class and runs the test operation. Before the test case is run, the test harness starts message sending sequence capturing. After the test case is complete, the test harness stops the capturing and writes the captured message sending sequence to an XML file, using the file name specified from the command line. The test harness is activated using a command at command prompt as shown in figure 8.4. Figure 8.5 shows a UML component diagram illustrating components of the instrumentation tool and test harness framework.

```
java th.ac.chula.testgen.mss.instru.TestHarness <test-case-class>
<output-file>

Example
>java th.ac.chula.testgen.mss.instru.TestHarness
test.PrintFileInfoTest01 PrintFileInfoTest01.xml
```

Figure 8.4 Usage of the test harness

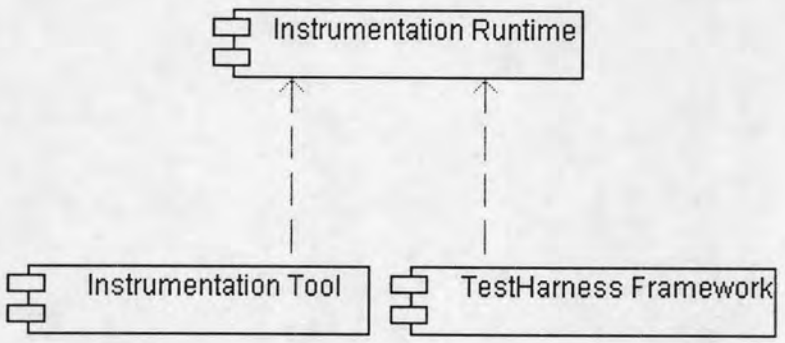


Figure 8.5 UML component diagram of the instrumentation tool and test harness framework

8.1.3 Message Sending Sequence Verification Tool

After test cases are generated, the program under test is instrumented, and then the instrumented program is tested with the test cases. After test execution, the actual message sending sequence captured from the execution is compared against the expected message sending sequence from the test case with the message sending sequence evaluator. The evaluator loads both message sending sequences and analyzes them using the procedure presented in chapter 6. Since the actual message sending sequence is captured from test execution of a particular test case, the analysis procedure demands that the classes of objects in both message sending sequences must exactly match. In other words, the expected message sending sequence already specifies which subclass is used for test for a polymorphic interaction. Figure 8.6 below shows the user interface of the message sending sequence evaluator.

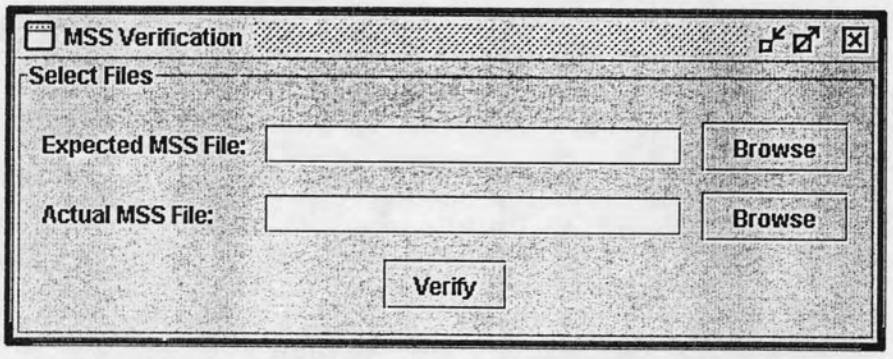


Figure 8.6 UI of the message sending sequence evaluator

8.2 Evaluation

The tool is evaluated with a number of UML sequence diagrams to show that the test approach is capable of generating test cases in all situations presented in this research.

8.2.1 Store Example

The store example from chapter 5 and 7 is used here to show that test cases can be generated for all 3 polymorphic assignment patterns. First the UML model is written in ArgoUML. The class diagram of the abstract class Product is shown in figure 8.7, and the sequence diagram of purchasing product process, which contains a Simple polymorphic assignment, is shown in figure 8.8.

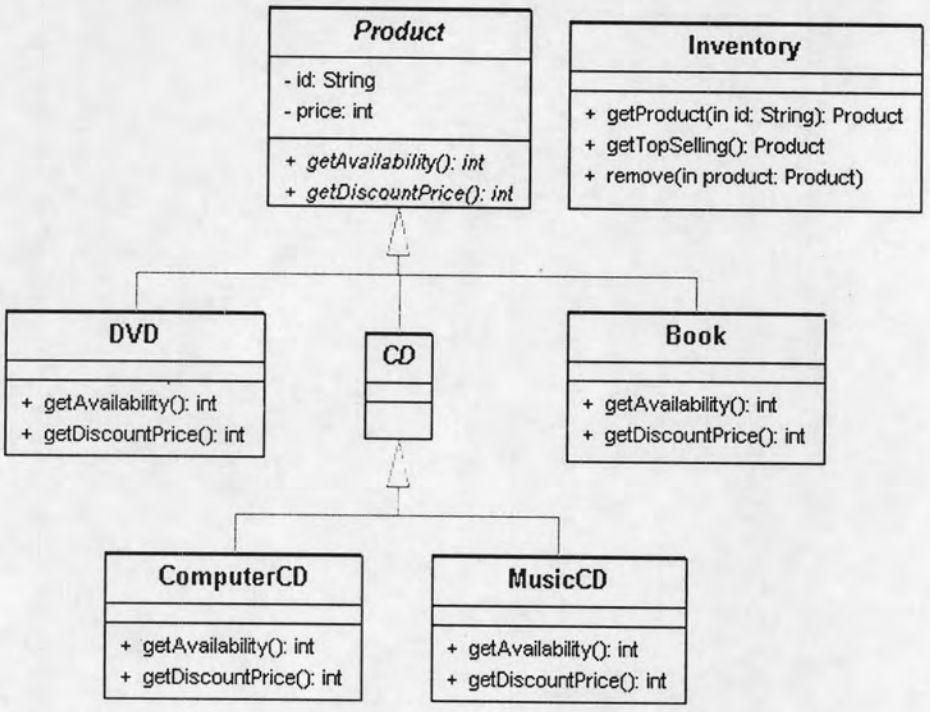


Figure 8.7 Product class diagram

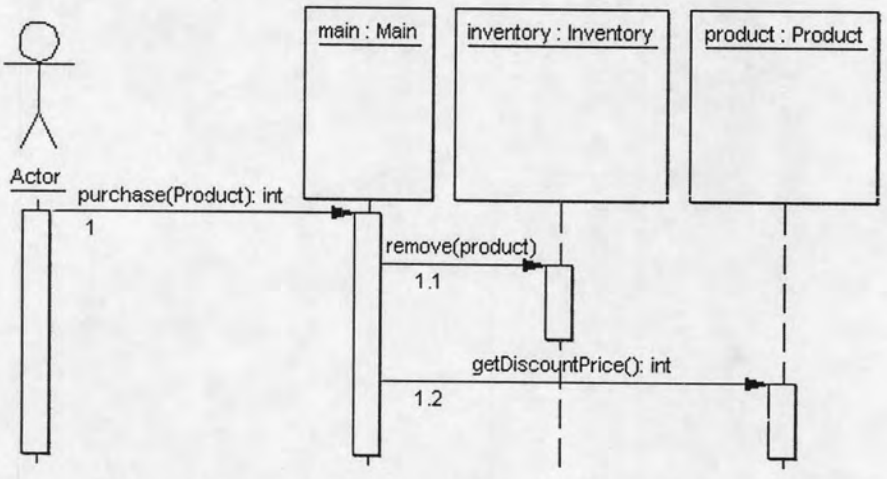


Figure 8.8 UML sequence diagram for purchasing product

In addition to the class hierarchy and the sequence diagram, it is necessary that particular tagged values are applied to corresponding UML elements to describe polymorphic behaviors. Tagged values are applied as discussed in chapter 7. Figure 8.9 shows how tagged values can be applied in ArgoUML.

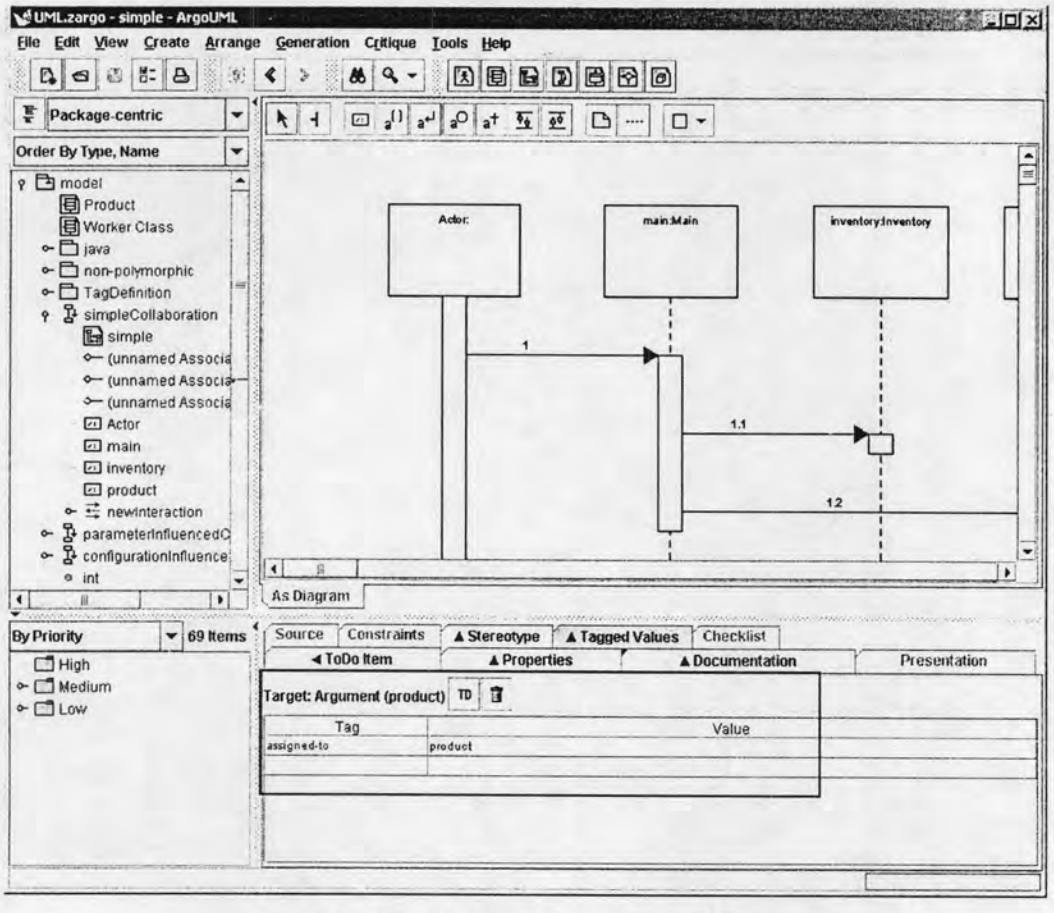


Figure 8.9 Tagged value in ArgoUML

The UML model is then exported to an XMI file. The XMI file is used for test case generation through the user interface as shown previously in figure 8.3. Since all concrete subclasses in the class hierarchy in figure 8.7 override the polymorphic methods in the store example, using overriding method coverage and using inheritance coverage criteria always result in the same test scenarios. Also, every sequence diagram in the store example has only one polymorphic assignment; therefore, there is no different between the two criteria and their strong versions. From this point on, it is assumed that one of the adequacy criteria, excluding base class coverage, is selected for test case generation for the store example.

The test case generator writes each generated test case and its expected message sending sequence to an HTML file and an XML file respectively. For each generation session, a test case summary file is also generated to an HTML file. Figure 8.10 shows an example of a list of files generated by the test case generator. Figure

8.11 and 8.12 shows the test case summary and an example of test case for purchasing product interaction respectively.

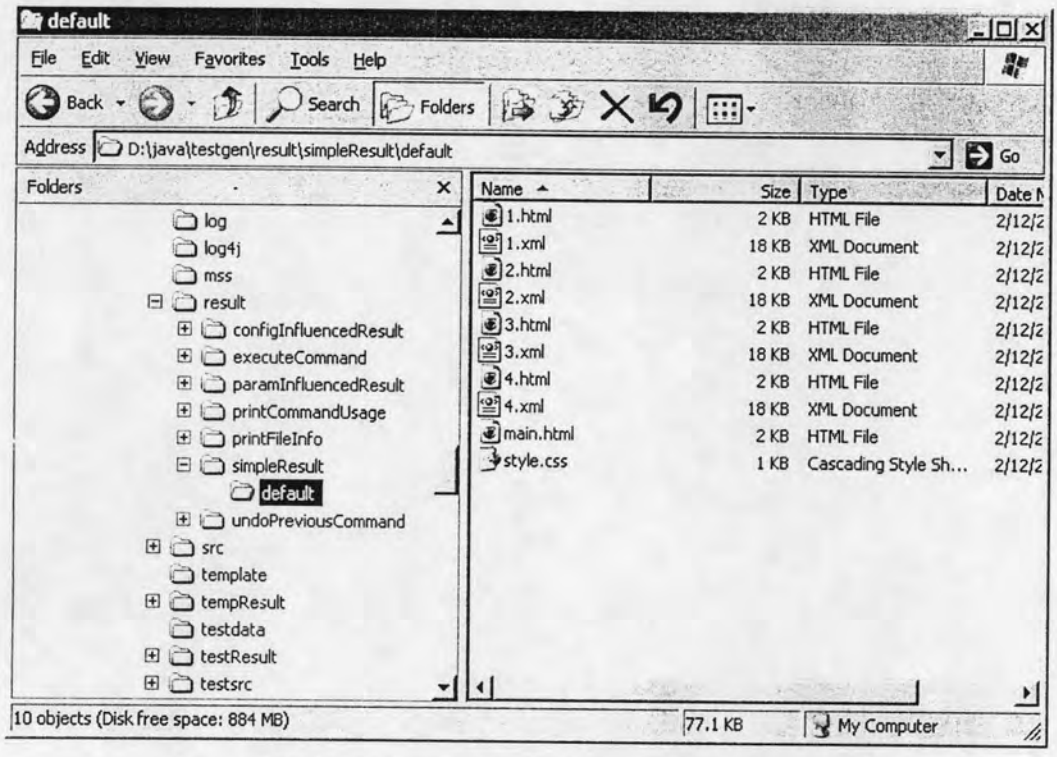
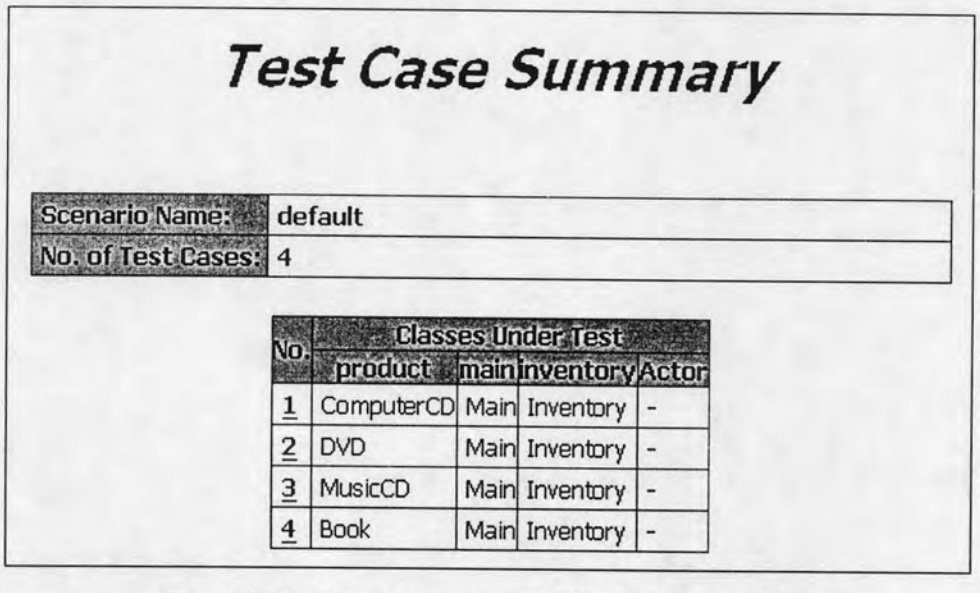


Figure 8.10 List of files generated by the test case generator



Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

No.	Classes Under Test			
	product	main	inventory	Actor
1	ComputerCD	Main	Inventory	-
2	DVD	Main	Inventory	-
3	MusicCD	Main	Inventory	-
4	Book	Main	Inventory	-

Figure 8.11 Test case summary for purchasing product

The test case summary shows the number of generated test cases along with the list of classes under test for certain roles in each test case. From figure 8.11, the test case no.1 has an instance of class ComputerCD for role 'product', which is the polymorphic entity in the interaction. The table shows classes under test for all roles, since it is possible that there is more than one polymorphic entity in an interaction.

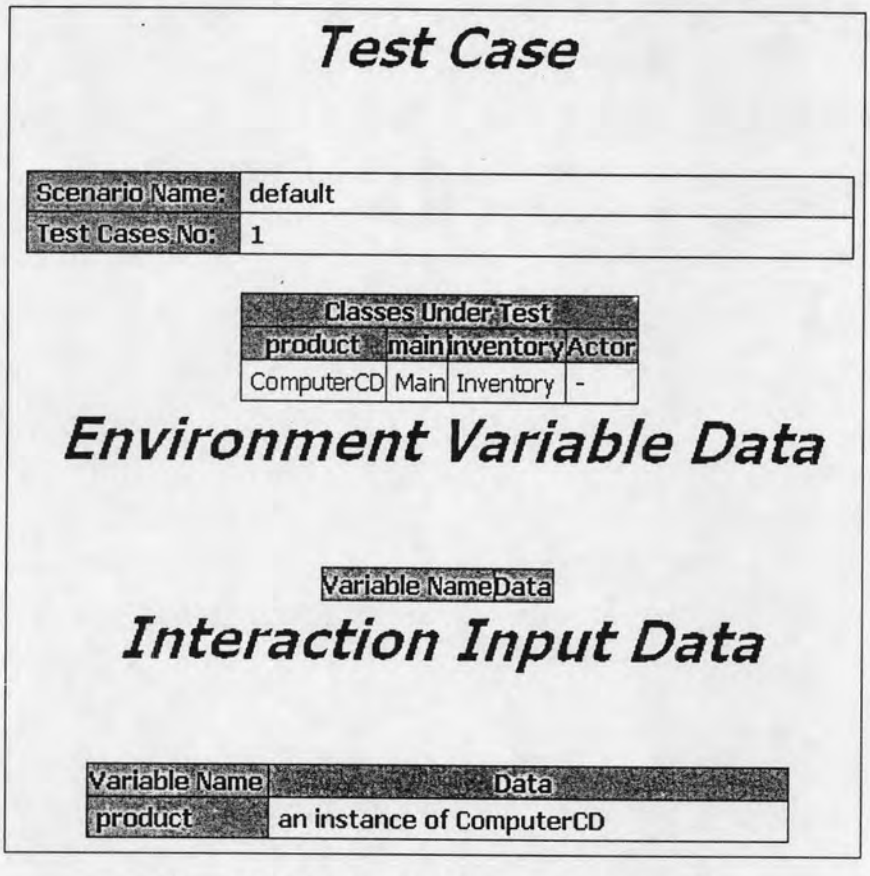


Figure 8.12 A test case for purchasing product

A test case HTML file consists of test data in two sections, environment variable data and interaction input data. For the purchasing product interaction only interaction input data section is presented with data. The test case shows that the interaction input named 'product' has a value as an instance of class ComputerCD.

Figure 8.13 shows a UML sequence diagram for checking product availability interaction, which contains a Parameter-influenced polymorphic assignment.

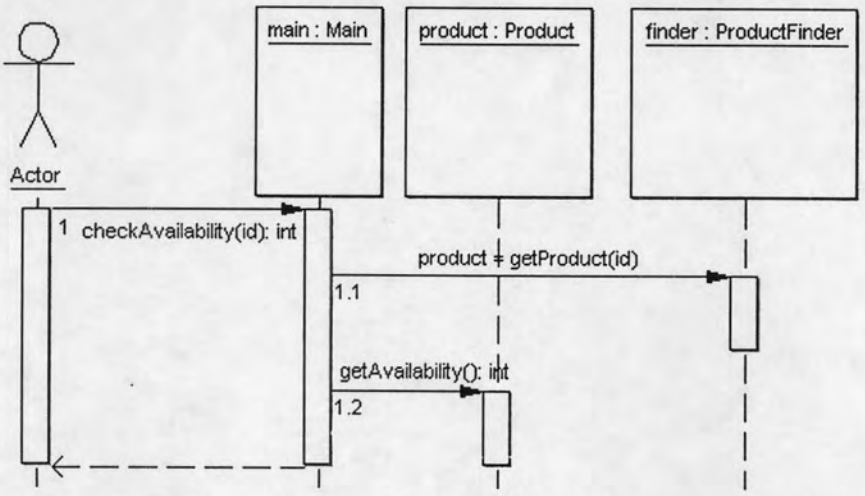


Figure 8.13 UML sequence diagram for checking product availability

Test case generation of the interaction also yields 4 test cases like the previous interaction. One of the generated test cases is shown in figure 8.14.

Test Case

Scenario Name:	default
Test Cases No:	1

Environment Variable Data

Variable Name	Data

Interaction Input Data

Variable Name	Data
id	19739911

Figure 8.14 A test case for checking product availability

In figure 8.15, a UML sequence diagram of checking availability of the top selling product, which contains a Configuration-influenced polymorphic assignment, is shown. One of the generated test cases from the interaction is shown in figure 8.16. Notice that the generated test data are in environment variable data section, since the interaction contains a Configuration-influenced polymorphic assignment.

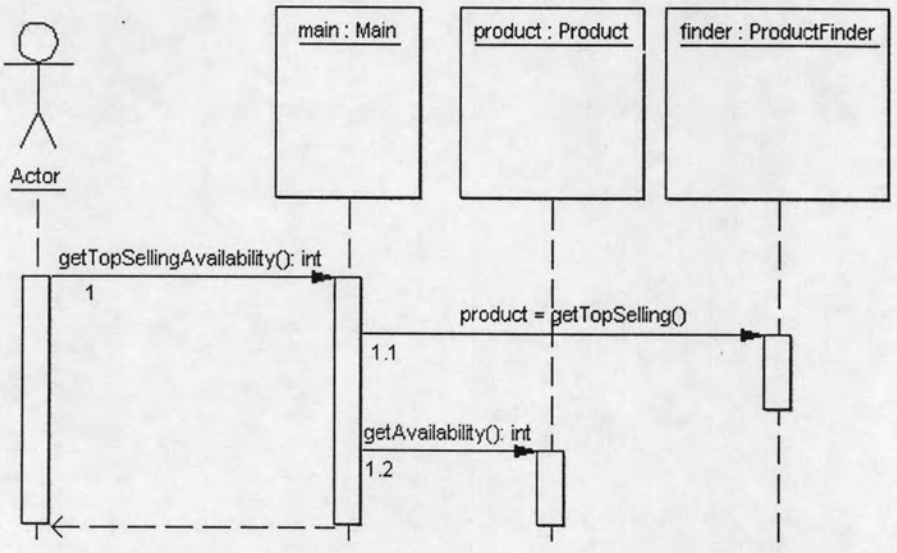


Figure 8.15 UML sequence diagram for checking availability of the top selling product

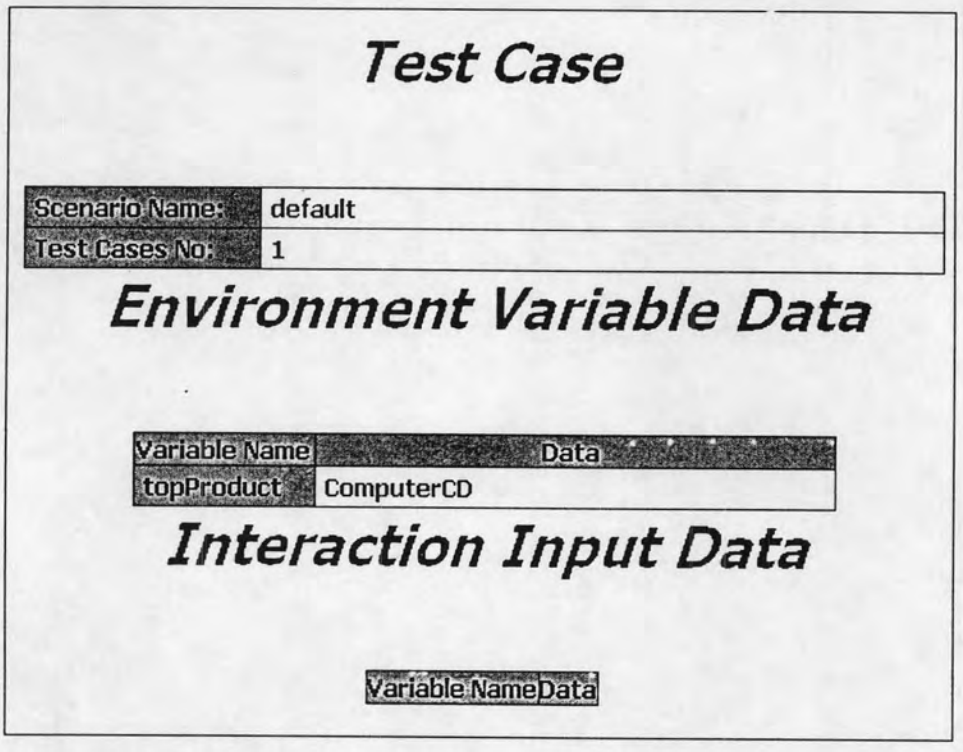


Figure 8.16 A test case for checking availability of the top selling product

8.2.2 File Management Example

This example is a design of several file management utilities. A file is modeled as an object as well as an operation on files. As a consequent, a general propose program, such as command execution, can be implemented to work with any command and any type of file. This ability relies on polymorphism of file and command objects.

Figure 8.17 and 8.18 shows class diagrams for the example. There are two inheritance hierarchies for the example. Notice that there are interfaces, abstract classes, and concrete classes in the diagrams. Using the tool to generate test cases for these diagrams would show that the tool can also handle interfaces and abstract classes as well as concrete classes.

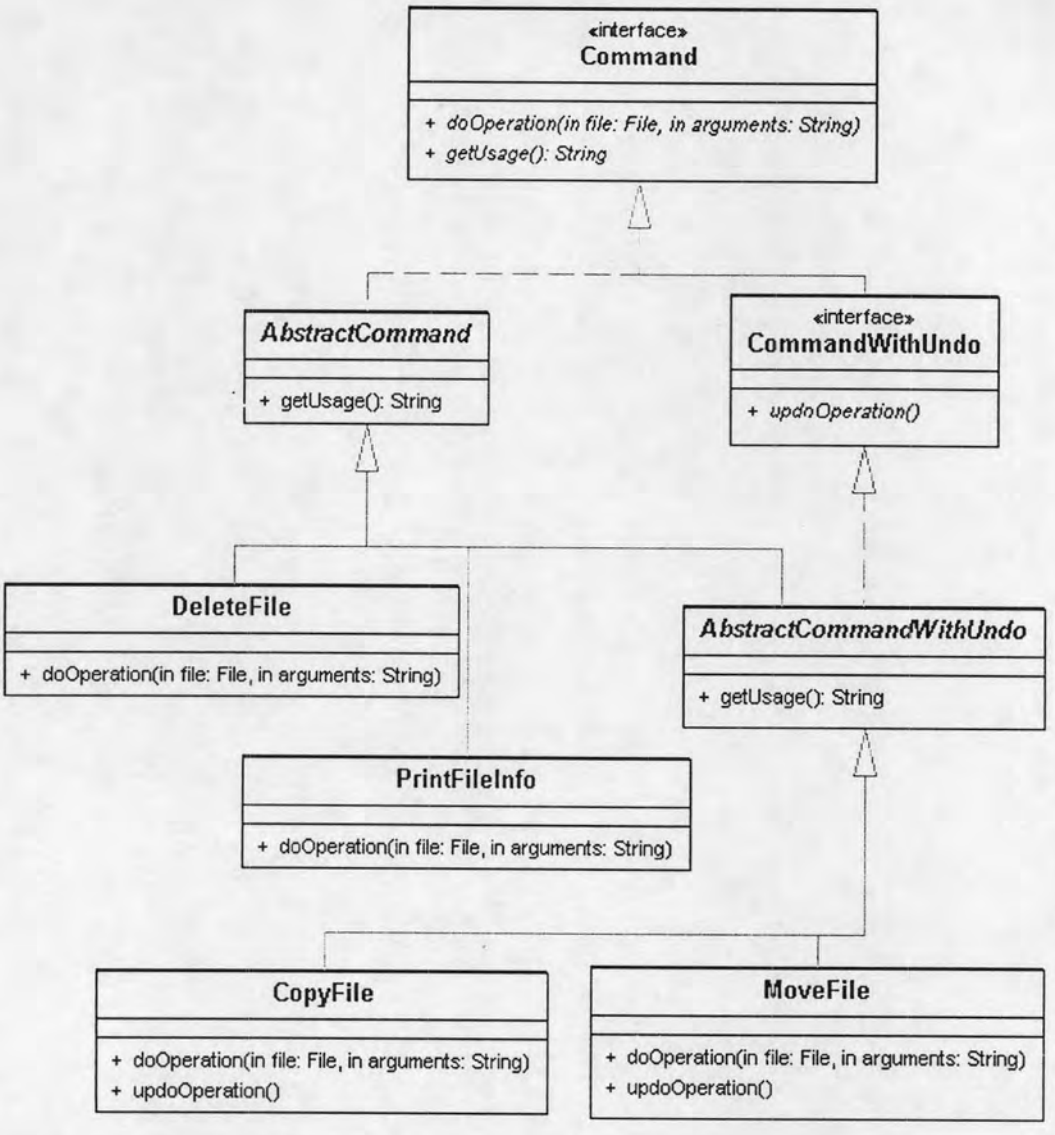


Figure 8.17 Class diagram for interface Command and its subtypes

From figure 8.17, a command is represented by interface named "Command". There is an abstract class "AbstractCommand" which implements a general method "getUsage" for all commands. There is an interface named "CommandWithUndo", which is a subinterface of interface "Command". The subinterface represents a type of command, which allows the action to be undone. Also there is an abstract class "AbstractCommandWithUndo", which extends "AbstractCommand" and implements "CommandWithUndo". The class overrides the method "getUsage" on class "AbstractCommand". There are four concrete classes in the diagram: class "DeleteFile"

and "PrintFileInfo" as subclasses of "AbstractCommand", and class "CopyFile" and "MoveFile" as subclasses of "AbstractCommandWithUndo".

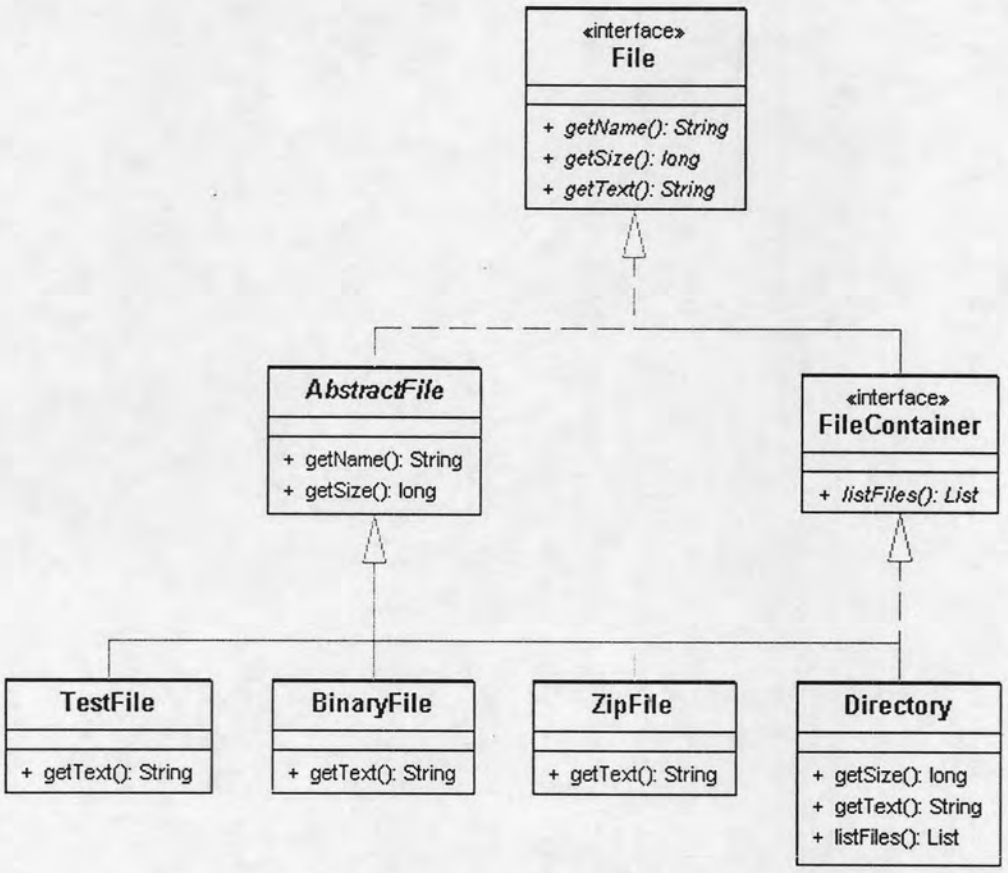


Figure 8.18 Class diagram of interface File and its subtypes

An Interface "File" in figure 8.18 is for representing a file on storage. There is a subinterface "FileContainer" for representing a container of files. An abstract class "AbstractFile" is an abstract implementation of a file. It contains implementation of two methods: "getName" and "getSize". There are four concrete classes in the diagram: class "TextFile", "BinaryFile", "ZipFile", and "Directory". The class "Directory" is also an implementation of interface "FileContainer".

Figure 8.19 shows a UML sequence diagram for print file information interaction. In the interaction, the actor passes an instance of interface "File" to a command, which is an instance of class "PrintFileInfo". The command then calls method "getName" and

"getSize" on the file instance. The interaction contains a Simple polymorphic assignment.

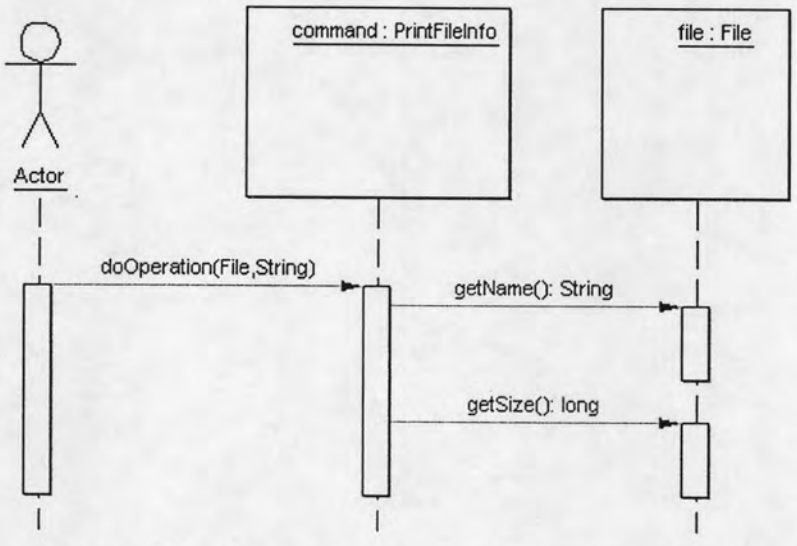
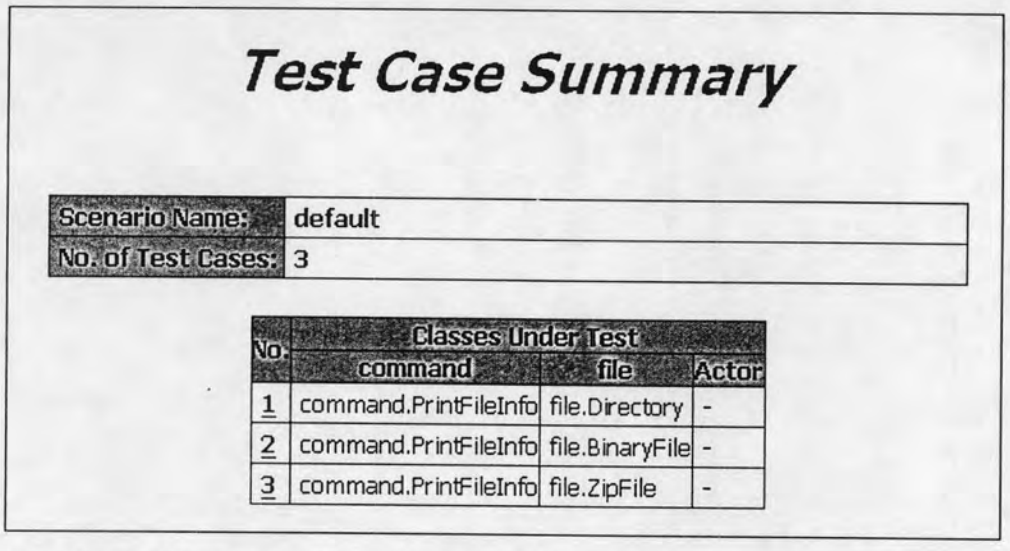


Figure 8.19 UML sequence diagram for print file info

The method "getName" and "getSize" is implemented on the abstract class "AbstractFile"; therefore, the concrete subclasses share the implementation of the methods. Using overriding method coverage and inheritance coverage criteria yield different test sets. Test summaries for both of them are shown in figure 8.20 and 8.21. The test case summary in figure 8.21 shows that all concrete subclasses are tested according to the inheritance coverage criterion. Figure 8.20 shows that only classes that overrides the method "getName" and "getSize" are required to be covered. However, class "AbstractFile" is abstract. It cannot be instantiated; therefore, one of its concrete subclasses must be selected for test instead. In this case, class "BinaryFile" is selected.

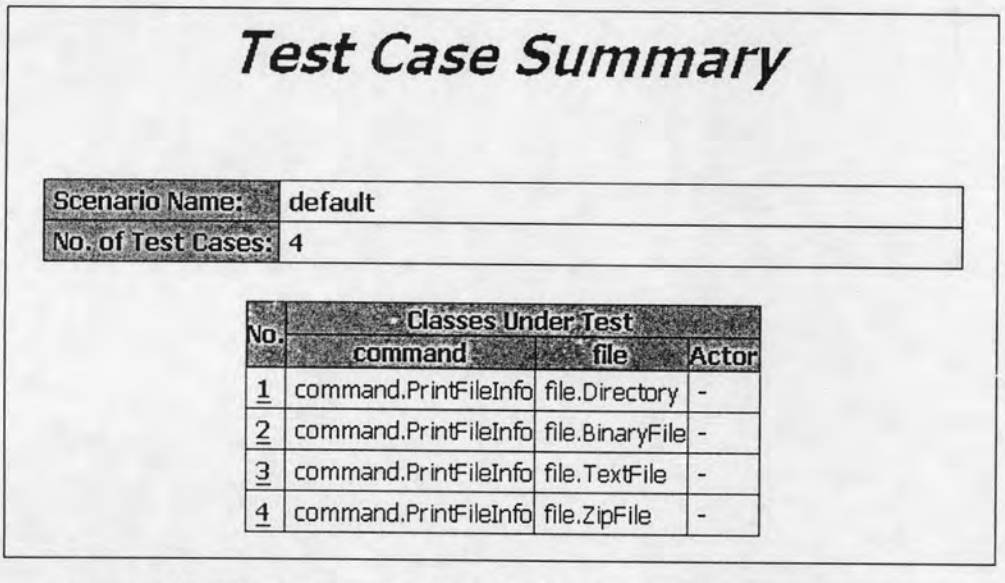


Test Case Summary

Scenario Name:	default
No. of Test Cases:	3

No.	Classes Under Test		
	command	file	Actor
1	command.PrintFileInfo	file.Directory	-
2	command.PrintFileInfo	file.BinaryFile	-
3	command.PrintFileInfo	file.ZipFile	-

Figure 8.20 Test case summary for print file info using overriding method coverage



Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

No.	Classes Under Test		
	command	file	Actor
1	command.PrintFileInfo	file.Directory	-
2	command.PrintFileInfo	file.BinaryFile	-
3	command.PrintFileInfo	file.TextFile	-
4	command.PrintFileInfo	file.ZipFile	-

Figure 8.21 Test case summary for print file info using inheritance coverage

Shown in figure 8.22 is a UML sequence diagram for print command usage interaction. The actor supplies a name of the command whose usage is to be printed to the main controller. The controller obtains an instance of interface Command from CommandFactory and then calls the method "getUsage" on the command. The interaction contains a Parameter-influenced polymorphic assignment.

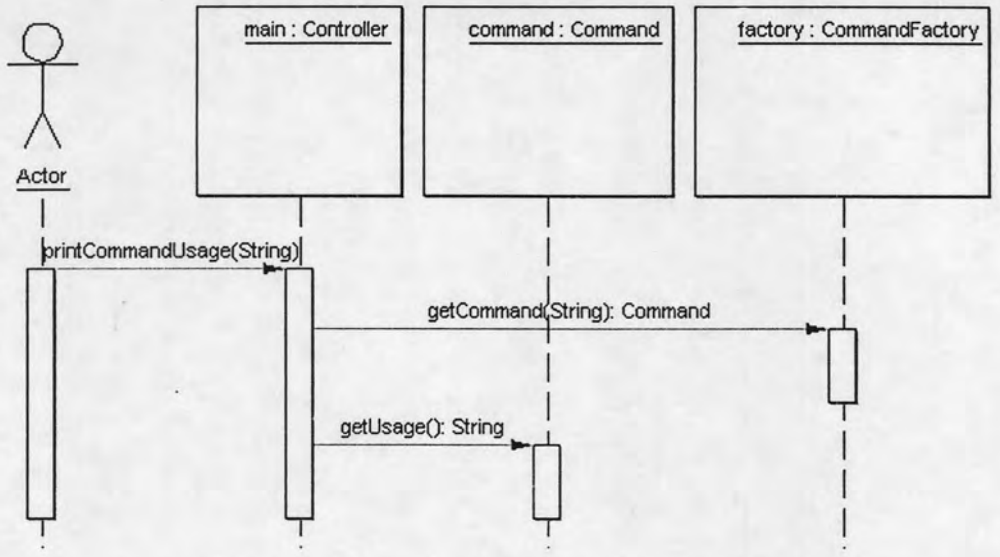
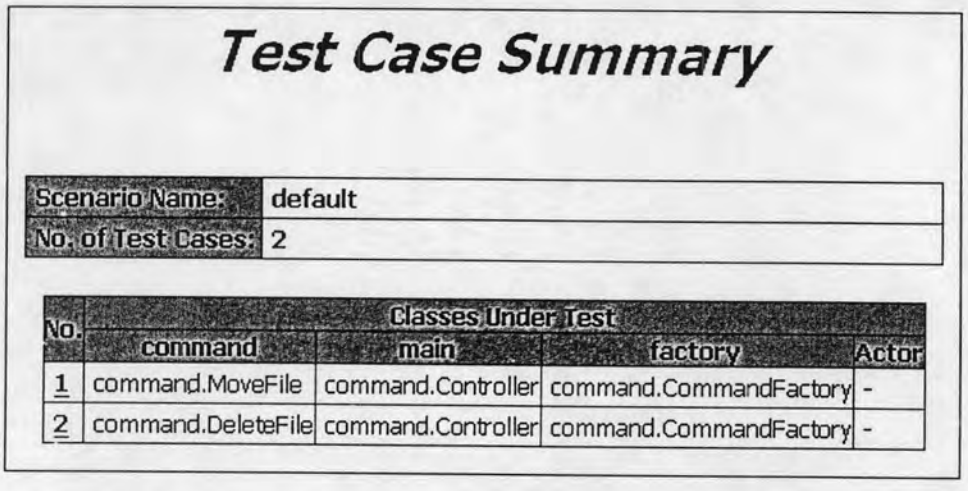


Figure 8.22 UML sequence diagram for print command usage

Using Overriding method coverage and Inheritance coverage result in different number of test cases according to the class diagram in figure 8.17. Similar to the previous interaction, this interaction involves interfaces and abstract classes. For overriding method coverage, one of concrete subclasses must be selected to be tested instead of its abstract super type. In this case, class "DeleteFile" is selected instead of "AbstractCommand", and class "MoveFile" is selected instead of "AbstractCommandWithUndo". Test case summaries for the example using each of both adequacy criteria are shown in figure 8.23 and 8.24 respectively.

This interaction and the previous interaction each have only one polymorphic assignment; as a result, using Overriding method coverage and Inheritance coverage are not different from using their strong versions.

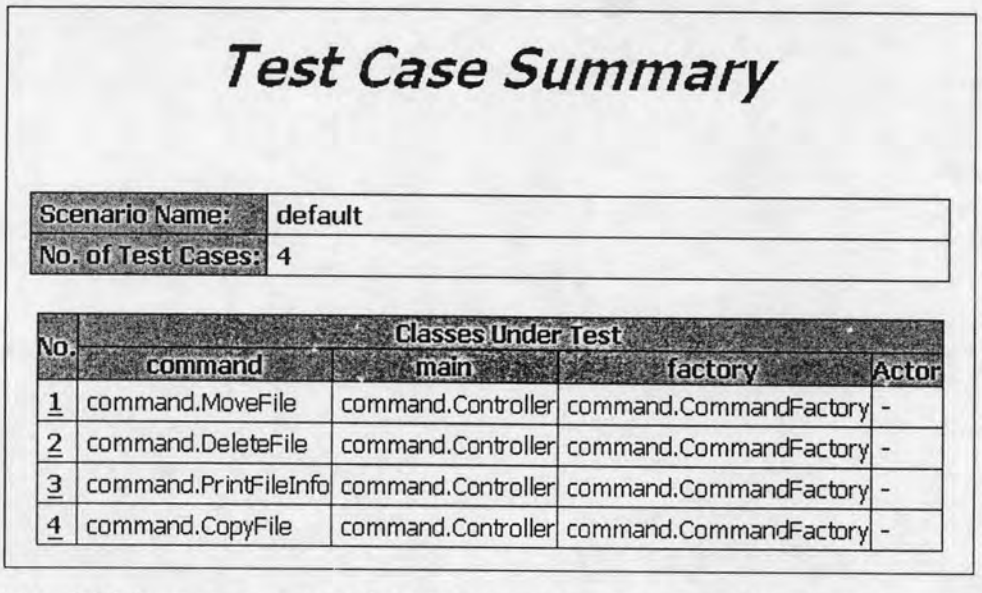


Test Case Summary

Scenario Name:	default
No. of Test Cases:	2

No.	Classes Under Test			
	command	main	factory	Actor
1	command.MoveFile	command.Controller	command.CommandFactory	-
2	command.DeleteFile	command.Controller	command.CommandFactory	-

Figure 8.23 Test case summary for print command usage using Overriding method coverage



Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

No.	Classes Under Test			
	command	main	factory	Actor
1	command.MoveFile	command.Controller	command.CommandFactory	-
2	command.DeleteFile	command.Controller	command.CommandFactory	-
3	command.PrintFileInfo	command.Controller	command.CommandFactory	-
4	command.CopyFile	command.Controller	command.CommandFactory	-

Figure 8.24 Test case summary for print command usage using Inheritance coverage

Figure 8.25 shows a UML sequence diagram for execute command interaction. In the interaction, the actor supplies a command name and an instance of interface File to be operated to the main controller. The main controller get an instance of interface Command from CommandFactory and passes an instance of interface File to the command. This interaction is different from the previous two interactions. There are two polymorphic interactions in the interaction: one for 'file' and one for 'command'. The one

for 'file' is a Simple polymorphic assignment, and the one for 'command' is a Parameter-influenced assignment.

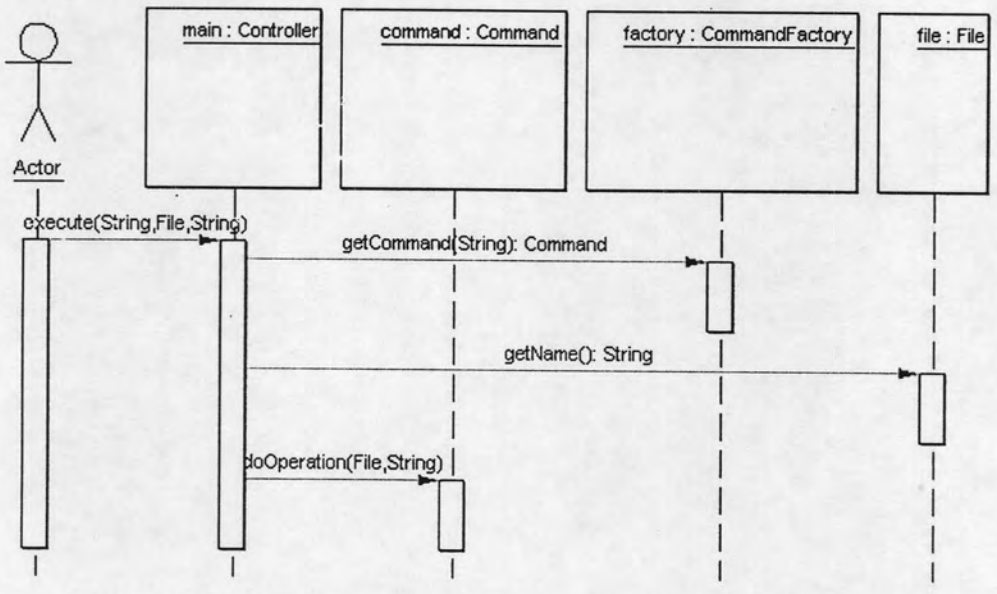
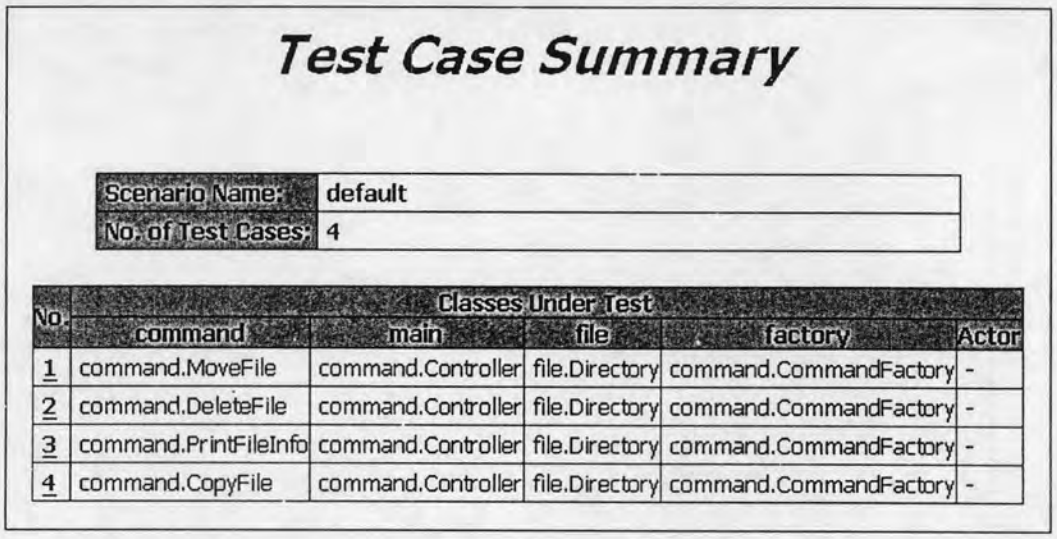


Figure 8.25 UML sequence diagram for execute command

Since there is more than one polymorphic assignment, Strong overriding method coverage and Strong inheritance coverage are different from their normal versions. Figure 8.26, 8.27, 8.28, and 8.29 show test case summaries for execute command interaction using Overriding method coverage, Inheritance coverage, Strong overriding method coverage, and Strong inheritance coverage respectively.

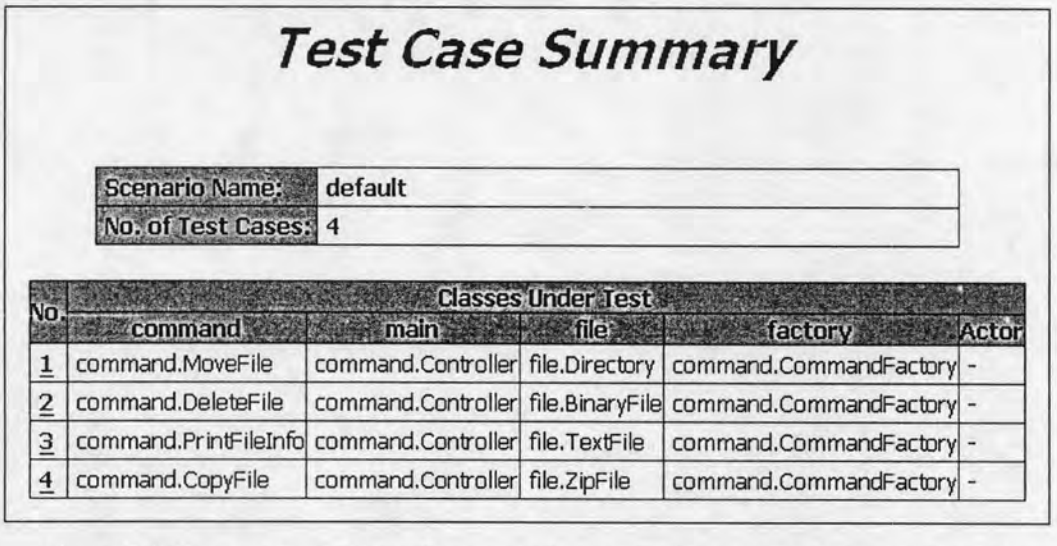


Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

No.	Classes Under Test				
	command	main	file	factory	Actor
1	command.MoveFile	command.Controller	file.Directory	command.CommandFactory	-
2	command.DeleteFile	command.Controller	file.Directory	command.CommandFactory	-
3	command.PrintFileInfo	command.Controller	file.Directory	command.CommandFactory	-
4	command.CopyFile	command.Controller	file.Directory	command.CommandFactory	-

Figure 8.26 Test case summary for execute command using Overriding method coverage



Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

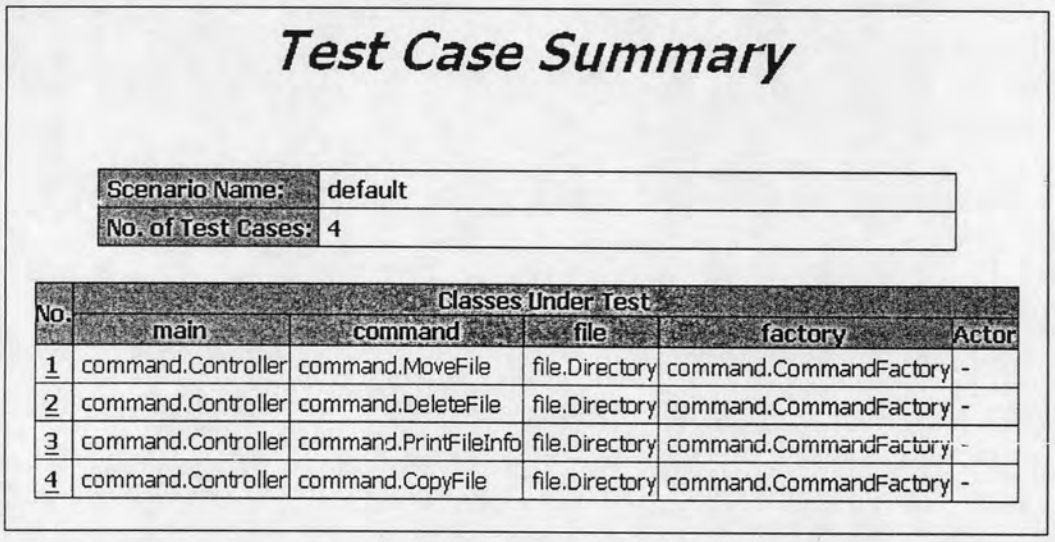
No.	Classes Under Test				
	command	main	file	factory	Actor
1	command.MoveFile	command.Controller	file.Directory	command.CommandFactory	-
2	command.DeleteFile	command.Controller	file.BinaryFile	command.CommandFactory	-
3	command.PrintFileInfo	command.Controller	file.TextFile	command.CommandFactory	-
4	command.CopyFile	command.Controller	file.ZipFile	command.CommandFactory	-

Figure 8.27 Test case summary for execute command using Inheritance coverage

Although the number of test cases for Overriding method coverage and Inheritance coverage are equal, they require different classes to be covered by test. The interaction has two polymorphic calls: the method "doOperation" on interface "Command" and method "getName" on interface "File". Since the abstract class "AbstractFile" provides the implementation of method "getName" for all concrete subclasses of "AbstractFile", only one of the concrete subclasses needs to be covered by test according to Overriding method coverage. The class "Directory" is selected in

this case as shown in figure 8.26. On the contrary, Inheritance coverage requires all subclasses to be covered. Consequently, all four concrete subclasses of "AbstractFile" must be covered by test. However, both adequacy criteria do not enforce testing these classes in combination. It is coincidentai that both adequacy criteria result in the same number of test cases.

Since there is only one class for role 'file' to be covered by test according to Overriding method coverage, the test cases satisfying the criterion also satisfy Strong overriding method coverage. This is only true when there is only one polymorphic entity in the interaction, which requires more than one subclass to be covered by test.



Test Case Summary

Scenario Name:	default
No. of Test Cases:	4

No.	Classes Under Test				
	main	command	file	factory	Actor
1	command.Controller	command.MoveFile	file.Directory	command.CommandFactory	-
2	command.Controller	command.DeleteFile	file.Directory	command.CommandFactory	-
3	command.Controller	command.PrintFileInfo	file.Directory	command.CommandFactory	-
4	command.Controller	command.CopyFile	file.Directory	command.CommandFactory	-

Figure 8.28 Test case summary for execute command using Strong overriding method coverage

Test Case Summary

Scenario Name:	default
No. of Test Cases:	16

No.	Classes Under Test				
	main	command	file	factory	Actor
1	command.Controller	command.MoveFile	file.Directory	command.CommandFactory	-
2	command.Controller	command.MoveFile	file.BinaryFile	command.CommandFactory	-
3	command.Controller	command.MoveFile	file.TextFile	command.CommandFactory	-
4	command.Controller	command.MoveFile	file.ZipFile	command.CommandFactory	-
5	command.Controller	command.DeleteFile	file.Directory	command.CommandFactory	-
6	command.Controller	command.DeleteFile	file.BinaryFile	command.CommandFactory	-
7	command.Controller	command.DeleteFile	file.TextFile	command.CommandFactory	-
8	command.Controller	command.DeleteFile	file.ZipFile	command.CommandFactory	-
9	command.Controller	command.PrintFileInfo	file.Directory	command.CommandFactory	-
10	command.Controller	command.PrintFileInfo	file.BinaryFile	command.CommandFactory	-
11	command.Controller	command.PrintFileInfo	file.TextFile	command.CommandFactory	-
12	command.Controller	command.PrintFileInfo	file.ZipFile	command.CommandFactory	-
13	command.Controller	command.CopyFile	file.Directory	command.CommandFactory	-
14	command.Controller	command.CopyFile	file.BinaryFile	command.CommandFactory	-
15	command.Controller	command.CopyFile	file.TextFile	command.CommandFactory	-
16	command.Controller	command.CopyFile	file.ZipFile	command.CommandFactory	-

Figure 8.29 Test case summary for execute command using Strong inheritance coverage

Test cases generated using Strong inheritance coverage shown in figure 8.29 are the result of Cartesian product of all classes to be covered by each role in the interaction. Since there are four classes to be covered by role 'command' and four classes to be covered by role 'file', the total number of test cases are 16. Note that the test cases generated using this adequacy criterion cover the same set of classes covered by the test cases generated using Inheritance coverage. The difference is that inheritance coverage does not have any requirement about combination of classes, while this criterion requires all possible combination of classes to be tested.

In addition to test case generation, the example is implemented and tested against the generated test cases. The program is also instrumented for capturing the

actual message sending sequence, and after the test execution, it is compared to the expected message sending sequence using the tool. Figure 8.30 and 8.31 shows examples of using the tool for evaluating message sending sequence.

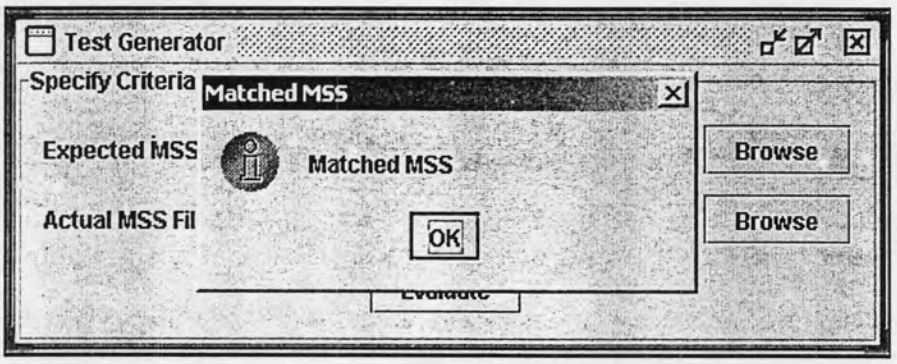


Figure 8.30 Message sending sequence evaluator – matched

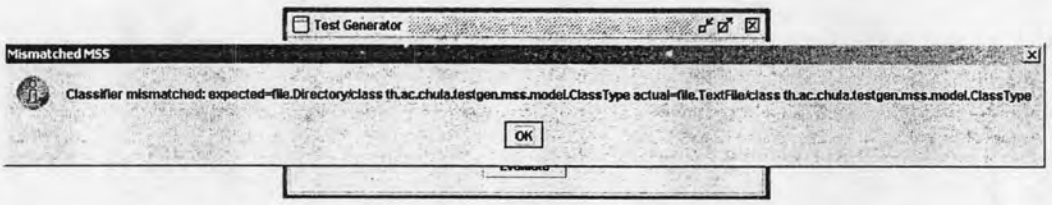


Figure 8.31 Message sending sequence evaluator – mismatched

8.2.3 Evaluation Summary

So far this section shows various situations where the tool can be applied. From the store example, the tool is demonstrated to show its capability of generating test case for interactions containing any of all three polymorphic assignment patterns presented in this research.

The file management example is used for evaluation here to show that the tool also works well with more sophisticated situations. The example shows that the tool can deal with class diagrams with interfaces and abstract classes. Both interfaces and abstract classes cannot be instantiated; therefore, they cannot be tested directly. One of concrete subclasses must be selected for test instead. The tool can identify if an

interface or an abstract class is selected to be covered by a particular adequacy criterion and can replace the selection with a proper concrete subclass.

Moreover, the file management example also shows that the tool can deal with more than one polymorphic assignment. This is where the adequacy criteria proposed in chapter 5 really count. The example shows that all the proposed criteria each yields different generated test cases, which cover different set of classes to be tested. In addition, the result also confirms the subsumption hierarchy of the adequacy criteria discussed in chapter 5.

Since the tool is implemented strictly based on the test approach presented in this research, it is possible to say that the tool represents the test approach. The major purpose of the evaluation of the tool is not to only show the correctness in the implementation of the tool, but also to show that the test approach is applicable and possible to be implemented. From the evaluation of the tool, the test approach, including the adequacy criteria, the message sending sequence model and its verification procedure, the polymorphic assignment patterns, the UML extension for test case generation, and the test case generation procedure, are evaluated through the use of the tool against the examples.