2006

Europa: a framework for writing reusable automated tests for C# components

Christian Castillo

Rochester Institute of Technology
MS Project Report

Europa: A Framework for Writing Reusable Automated Tests for C# Components

By Christian Castillo
cac2763@cs.rit.edu

Version 1.0
January 28, 2006

Committee:

Chairperson: Prof. Trudy Howles _____

Reader: Dr. Fereydoun Kazemian _____

Observer: Gary Passero _____

Abstract

One of the main goals of software construction is reusability. Component-based (CB) development is a new paradigm that promises to facilitate the reuse of software elements. The component-based model attempts to create an analogy with computer hardware. The idea is that software applications should be developed by assembling existing components rather than reinventing solutions to problems that have already been solved. In this paradigm, a component consumer is faced with the problem of determining if a given component meets his expectations. Testing the component can help answer this question. The component consumer may have to apply the same test cases to several different implementations of a given component in order to choose the one that best meets his needs. In traditional automated test frameworks, the test cases are tightly coupled with the unit under test, which reduces the opportunities for reuse. Under such a framework, the component consumer would need to write custom code to test each candidate component implementation, which may dramatically increase the cost of the development effort. The goal of this project is to develop an automated test framework for C# components that allows for the reuse of tests on different implementations of a given component.

Contents

1 Introduction

Software testing is an integral part of the software development process. Generally speaking, software testing is the execution of code using a combination of inputs and state in order to reveal defects in the software. One common practice among beginner programmers is to insert output statements in certain key locations in their code to help diagnose problems. When the software is believed to be correct, these statements are removed or commented out. Another common practice is to insert assertions in the code. These assertions are checks on the state of the computation. If an assertion is not satisfied, the program execution is aborted.

As the need for higher quality control in the software development process increases, the need for better testing approaches becomes evident. Using a test driver to apply a set of test cases to a component is an improvement over the methods mentioned above. The test driver is a utility program whose only purpose is to verify the correct operation of some other piece of software. Writing a test driver provides a mechanism to automatically execute a set of test cases on a given software component. The disadvantage of using this approach is that a custom test driver is required for each software component. Managing and executing test cases using this approach becomes harder as the number of test drivers involved increases. One of the reasons for this is the lack of consistency in the way that correctness is verified and test results are reported.

Automated test frameworks overcome many of the deficiencies of the test-driver approach. An automated test framework provides a common foundation on which test suites (sets of test cases) can be written and executed in a uniform way. Most of the automated test frameworks available today are object-oriented. Usually, a class represents a test suite, and methods within that class represent individual test cases. Automated test frameworks typically provide a construct for asserting on a pass/fail condition of a test case, and a component for managing the execution of the test cases. Some of the most widely used automated test frameworks include JUnit (for Java), and CppUnit (for C++). These are both freely available.

1.1 Component-based Software Engineering

Component-based software engineering (CBSE) is a discipline concerned with the development of software systems by assembling reusable components, the development of components as reusable assets, and the maintenance of systems built from components. The component producer is the individual or organization that develops a software component. The component consumer is the individual or organization that builds a software system by assembling such components. The components that make up a system may come from in-house collections or from third-party vendors, including commercial-off-the-shelf (COTS) components.

The main motivation for CB development is reducing the cost of producing software. The idea of building applications from reusable components is not new: as early as 1968 [1] proposed a software industry based on reusable components. With the emergence of component-oriented platforms and products over the last few years, the possibility of adopting this new software development paradigm is now greater than ever.

Development models for building applications from software components is a topic of on-going research [2]. The process of choosing a particular component implementation to be used as part of an application must include, at a minimum, the following phases:

1. A specification phase, in which the features of the required component are identified.
2. A searching phase, in which several candidate implementations of the required component are identified.
3. A selection phase, in which the component implementation that best meets the application's needs is chosen.

The specification phase takes place after the component consumer has defined the architecture of the application to be developed, and identified the components needed to achieve the desired functionality. The application developer will then define the functional and non-functional requirements for each one of these components.
The searching phase involves locating the needed components in some sort of component repository [3]. The Software Engineering Institute at Carnegie Mellon University is currently developing a prototype of one such repository. Agora is an automatically generated, indexed, worldwide database of software components [4]. Agora is designed to make it easier for application developers to find components that meet their requirements.

This project focuses on the selection phase. The main criterion that must be met by a component implementation in order to be selected to become part of the application being developed is that it must satisfy the functional requirements identified by the component consumer during the specification phase. Therefore, the component consumer will apply, as a first step, a set of test cases to each candidate implementation in order to exclude from further consideration those implementations that fail to meet expectations.

The component selection phase can be completed more efficiently if the same set of test cases is used to validate all the candidate component implementations. The advantage of reusing the test suite during the component selection phase can be easily understood by considering the alternative approach: building a separate test suite for each component implementation being evaluated. Using this approach, a worst-case scenario will yield *n* different test suites for testing *n* different component implementations. In applications that use a large number of components, the cost of creating custom test suites to evaluate different implementations of each component can be quite large. Moreover, once a component implementation has been selected, the test suites that were built for evaluating the other component implementations are no longer needed and are likely to be discarded. In order to reduce cost, project managers may be inclined to set limits on the number of components to be evaluated during the selection phase. As a result of using this approach, not only does the cost of building the application increase, but the quality of the application may suffer, since cost-reducing decisions might dictate that component implementations (possibly including the best one) be ruled out.

In addition to reusing the test suite during this initial component evaluation, the test suite can also be reused during the maintenance phase for testing the replacement of an existing component in the application. This replacement can be a new version of the component released by the same component producer, or a different implementation altogether. In either case, it is important to perform regression testing on the new

component implementation in order to verify that the new implementation (or new version) of the component does not break the functionality of the application.

Integrating reuse processes and activities into the software lifecycle enables significant software productivity, quality, and cost improvements [5]. The test suite is an asset that should be designed for reuse. Europa is a test framework that allows a component consumer to run the same test cases on different implementations of C# components, thus reaping the benefits of asset reuse.

1.2 Motivation

Traditional automated test frameworks (e.g. JUnit, CppUnit, csUnit) do not fit well in the CB paradigm. The main reason for this is the fact that, in these frameworks, the test cases are tightly coupled with the unit under test. As a result, a component consumer would have to rewrite the test cases in order to test a different implementation of the same component. Figure 1.1 illustrates this situation.

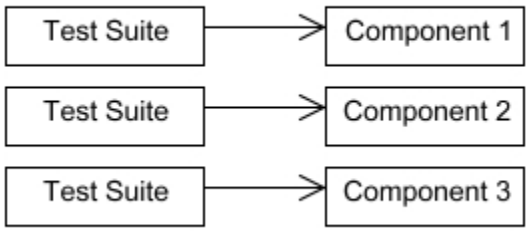

Figure 1.1: Test cases have to be re-written to adapt to each component implementation's interface.

The need to rewrite the test cases in order to test each component implementation is mainly due to differences in class names and method signatures. Rewriting test cases for each candidate component implementation will inevitably lead to an increase in the total cost of the application development.

The key to reusing the test cases is decoupling them from the unit under test. This can be achieved by writing the test cases to target a virtual component, and adapting the function calls invoked by the individual test cases to what the specific component implementation expects. This is illustrated in Figure 1.2.

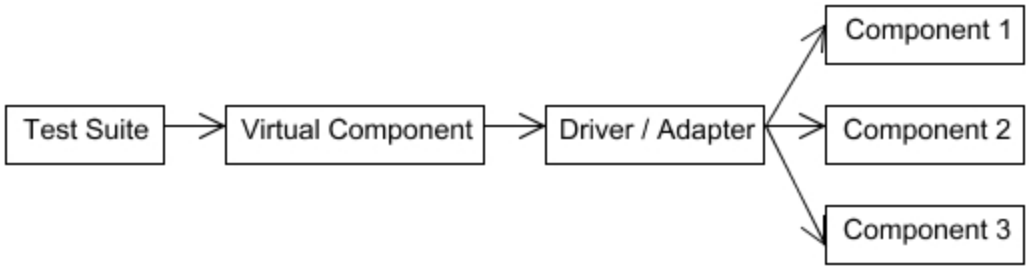

Figure 1.2: Test cases are decoupled from component implementation.

The architecture of this framework was first described by [6]. In this architecture, individual test cases are written against a virtual component. At run time, the virtual component forwards all function calls to the driver, which adapts each function call to match what is expected by the real component. The return value of each function call makes its way back to the test case that issued the call, which then reports on the outcome of that particular test case. The results of all the test cases are collected by the driver and displayed to the user.

2 Technical Background

The .NET Framework introduces a layer of abstraction between software and the hardware that it runs on. In .NET, code is compiled into an intermediate form, Microsoft Intermediate Language (MSIL), which is then executed by the Common Language Runtime (CLR).

MSIL is a combination of code and metadata (information about the constructs in the program). One of the advantages of the .NET architecture is the ability to read and manipulate the metadata in an assembly. The ability to access, at run-time, the metadata in an assembly is called reflection.

Reflection plays a central role in the design of Europa. The framework uses the mechanism of reflection in order to present to the user a view containing the interfaces of the virtual component and the real component. Through this view, users are able to define the correspondence among the methods in the virtual component and the real component. When executing the test cases, reflection is once again used in order to find and invoke the appropriate method in the real component. The figure below illustrates how reflection can be used.

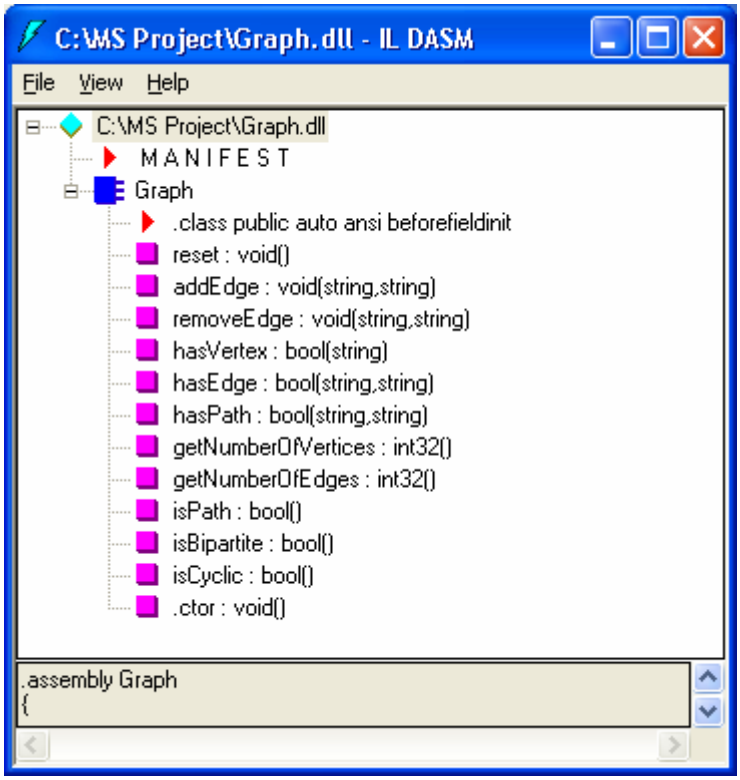

Figure 2.1: Viewing metadata in an assembly.

Figure 2.1 demonstrates how reflection is used by Microsoft Intermediate Language Disassembler (ildasm) to access the metadata of the Graph component. ildasm is a tool that is distributed with the .NET Framework SDK. It allows users to load an executable (dll or exe) through the File menu, and displays information associated with the types in the assembly.

3 Case Study

The purpose of the case study is to make the discussion more concrete. This case study considers a component for discovering basic properties of an undirected graph. The interface of one possible implementation of such component is given below.

```
public class Graph
{
        public void reset();
        public void addEdge(string source, string destination);
        public void removeEdge(string source, string destination);
        public bool hasVertex(string name);
        public bool hasEdge(string source, string destination);
        public bool hasPath(string source, string destination);
        public int getNumberOfVertices();
```

```
        public int getNumberOfEdges();
        public bool isPath();
        public bool isBipartite();
        public bool isCyclic();
}
```

Clients of this component describe the structure of the graph by specifying its edges. For instance, if g is a component of the type above, the following sequence of statements draws K_3:

```
g.reset();                  // erase previous graph, if any
g. addEdge("1", "2");       // draw K_3 (a complete graph with 3 vertices)
g. addEdge("2", "3");
g. addEdge("3", "1");
```

Once a graph has been specified, clients of the component can query the component about basic properties of the graph. The interface supported by the Graph component is by no means complete. Its main purpose is to help illustrate some of the concepts that will be introduced in the next sections.

4 Design

Europa was developed using the techniques of object-oriented analysis and design. The steps followed in designing the framework are:

- Determine use cases
- Study specific scenarios
- Construct interaction diagrams
- Construct detailed class diagram

4.1 Use Cases

Users interact with the test framework in three major ways: when writing the automated test cases to be executed on the component implementations, when adapting the test cases to match the interface of a specific component implementation, and when selecting and executing the test cases. These use cases are illustrated and discussed below.

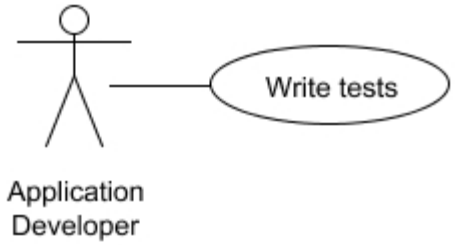

Figure 4.1 Use case diagram for writing the automated tests.

Figure 4.1 is the UML representation of the use case for writing the test cases to be executed on the component implementations. These test cases are written once the requirements of the desired component have been identified. Writing the test cases also involves creating the virtual component. The functional requirements of the desired component are expressed as methods of the virtual component. The application developer writes the test cases against the virtual component.

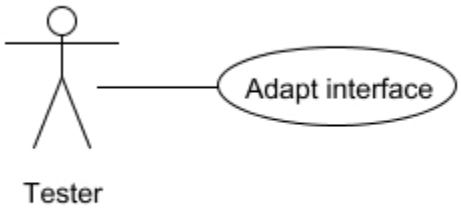

Figure 4.2 Use case diagram for adapting to target component's interface.

The framework allows users to specify the mapping between the methods in the virtual component and the real component being tested. This use case is illustrated in Figure 4.2.

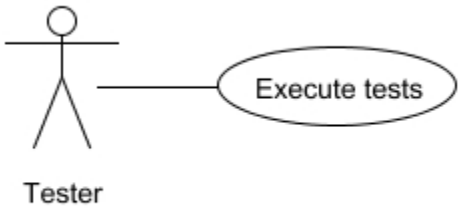

Figure 4.3 Use case diagram for executing test cases.

Figure 4.3 is the use case diagram for executing the test cases on a specific component implementation. The framework allows users to choose which test cases to execute.

4.2 Scenarios

Use Case: AdaptInterface
ID: UC1
Actors: Tester
Assumptions: The virtual component and the test suite have been written and loaded.
Flow of Events: 1. The user launches the Method Mapping Wizard (MMW). 2. The MMW presents a view of the methods found in the test suite. 3. For each method in the test suite, the user right-clicks on the method name and chooses 'Adapt' from the context-menu. 4. The MMW presents a dialog with controls populated with information specific to the methods found in the real component. 5. The user selects the method in the real component that the method being processed should be mapped to. 6. When all methods have been mapped, the user clicks File->Save. 7. The MMW presents a file dialog box. 8. The user types a file name, and chooses the location where the interface adapting information should be saved. 9. The MMW saves the interface adapting information in the file specified by the user.

Use Case: ExecuteTests
ID: UC2
Actors: Tester
Assumptions: - The virtual component and the test suite have been written. - The interface adapting file for the real component being tested has been created.

Flow of Events:

1. The user loads the dll containing the test suite.
2. The user loads the file containing the interface adapting information.
3. The user loads the dll containing the real component being tested.
4. The framework presents a view containing all the test cases found in the test suite.
5. The user selects the test cases to execute by placing a check mark next to those test cases to be executed and clicks on 'Run Tests'.
6. The framework executes the test cases that were selected by the user and presents a Pass/Fail indication when test execution is complete.

4.3 Interaction Diagrams

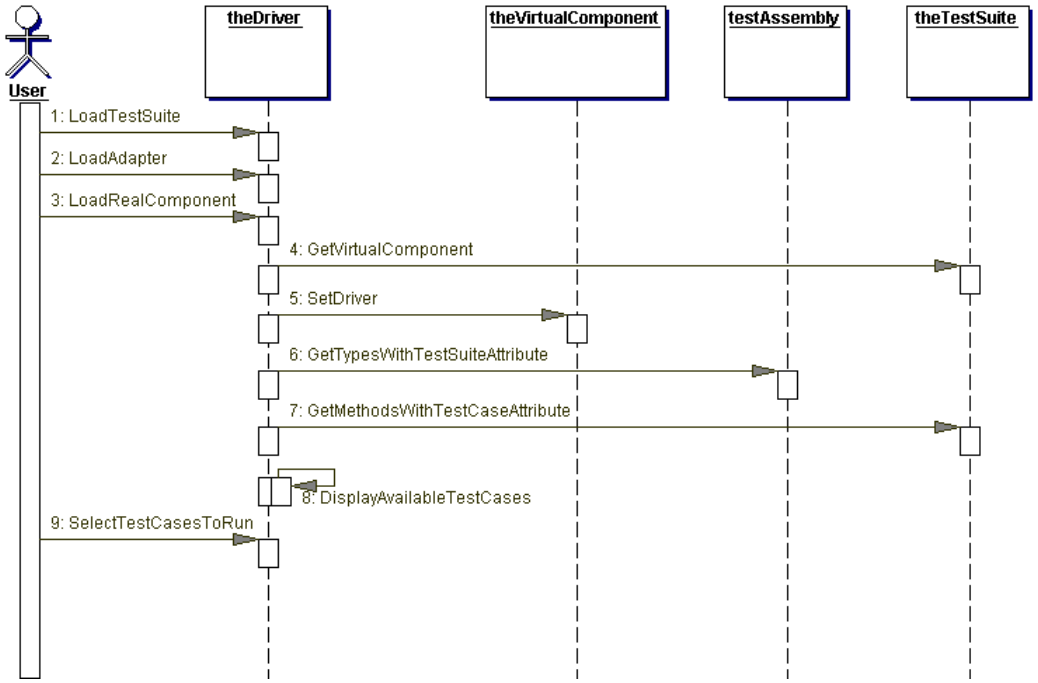

Figure 4.4 Sequence diagram for selecting test cases to execute.

Figure 4.4 is a sequence diagram showing the interactions that take place when the user selects which test cases to execute. The user starts by loading the dlls containing the adapting information necessary to execute the test cases. This is done through the main user interface of the framework's driver and corresponds to messages (1) to (3) in the sequence diagram.

The test cases in the test suite are written against a virtual component. This virtual component codifies the desired functionality in the form of public methods. The test suite and the virtual component are housed in the same dll. Furthermore, the test suite holds a reference to the virtual component it uses. In messages (4) and (5), the driver retrieves that reference and informs the virtual component that it should forward method calls to the driver (through an interface).

In messages (6) and (7) the driver identifies the test suite and the test cases contained within it. In message (8) the available test cases are displayed to the user. Finally, in message (9) the user selects the test cases to be executed.

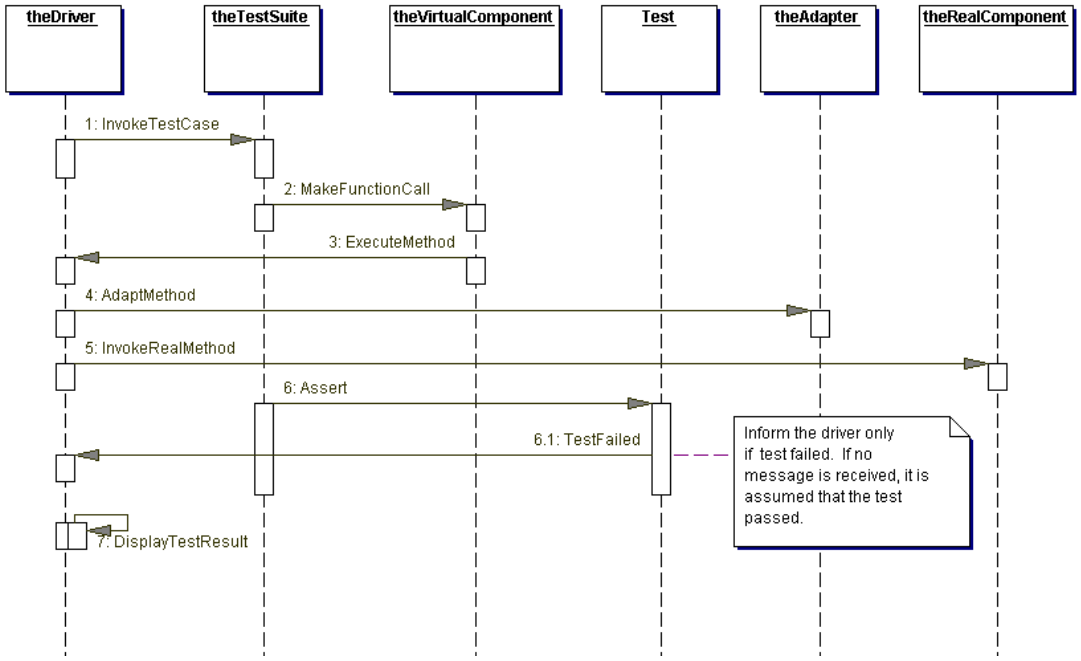

Figure 4.5 Sequence diagram for adapting and executing a test case.

Figure 4.5 is a sequence diagram for adapting a method call on the virtual component to what is expected by the real component during the execution of a test case. In message (1) the framework driver invokes a test case that has been selected by the user for execution. Message (2) shows the test case being executed making a function call on the virtual component (recall that test cases are written against a virtual component). The virtual component forwards the function call to the driver (message 3). In message (4) the method call is adapted to what is expected by the real component. Once the adapting information has been retrieved, the corresponding real method is invoked in message (5). In message (6), an assertion is checked to determine if the test case passed or failed. The driver is informed only if the assertion failed –no notification is sent to the driver if the test case passed. This is shown in message (6.1). Finally, in message (7) the driver displays the result of the execution of the test case to the user.

4.4 Detailed Class Diagrams

This section presents class diagrams showing the attributes and the public interfaces of the types that make up the framework. The diagrams also show the relationship between these types. The classes have been grouped into two clusters: the test-writing cluster and the test-execution cluster. The test-writing cluster contains the types that are relevant when writing the test suite, the test cases that make up the test suite, and the checks that determine whether test passed or failed. The test-execution cluster contains the classes related to the invocation, adaptation and execution of the test cases.

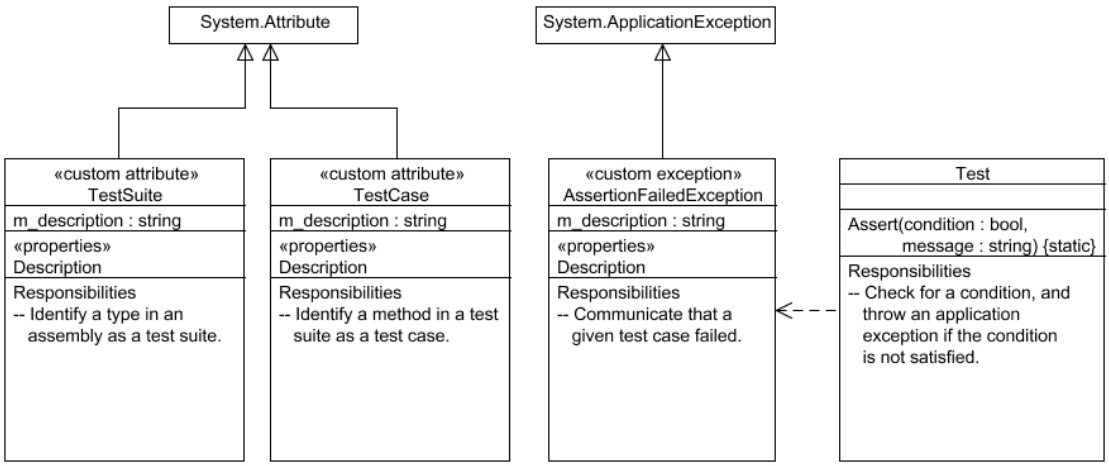

Figure 4.6 Class diagram for the test-writing cluster.

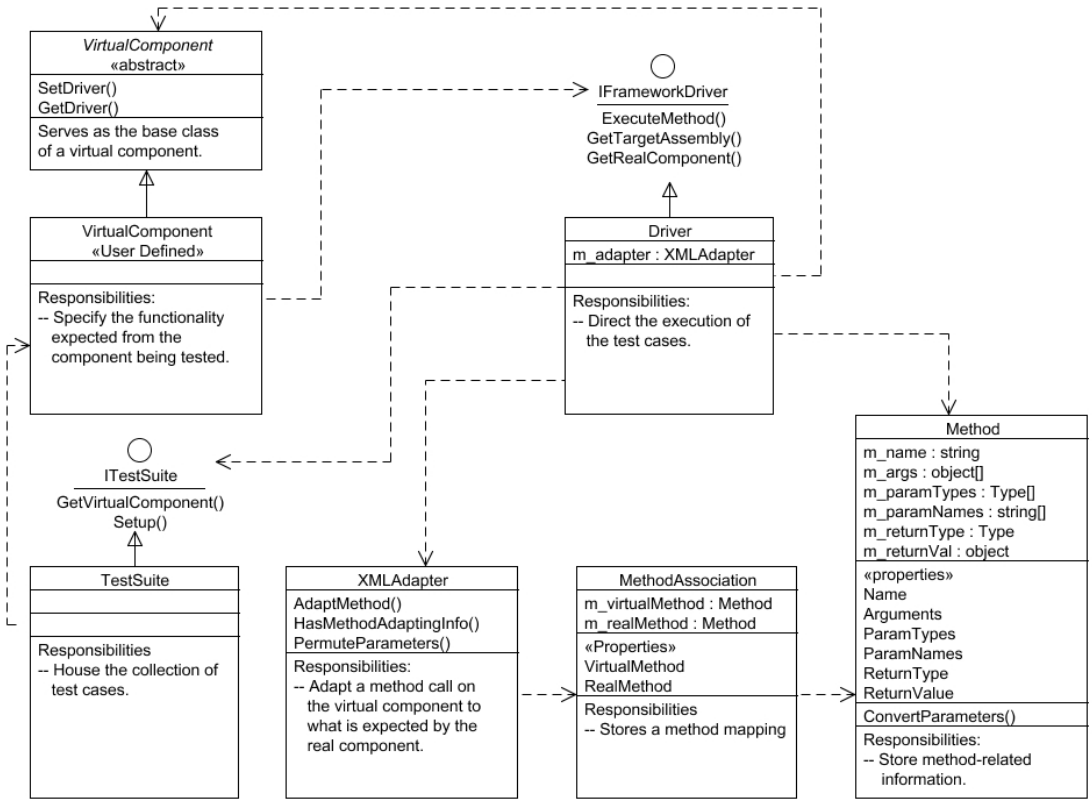

Figure 4.7 Class diagram for the test-execution cluster.

5 User Manual

5.1 Introduction

This manual contains detailed instructions on how to use Europa. Using the test framework involves: writing the test cases, and executing the test cases. In order to execute the test cases, a file containing method adapting information for each real component implementation must be provided to the framework. Europa includes a tool for creating and storing the method adapting information. Section 2.1.1 covers the topic of writing test cases. Section 2.1.2 discusses test execution.

A sample test package is provided with Europa. This test package consists of three different implementations of a Graph component, a test suite, and method adapting information for each component. The sample test package is used throughout this document to help illustrate how the framework is used.

5.1.1 Software Requirements

Europa requires that the .NET Framework version 2.0 be installed. The .NET Framework v2.0 is available as a free download at:

http://msdn.microsoft.com/netframework/downloads/updates

5.2 Getting Started

Europa is made up of two assemblies: twEuropa.dll and Europa.exe. twEuropa.dll is the test writing component. It provides classes that are used for writing the test cases. Europa.exe is the test execution component. It provides the user interface for loading and executing the test cases.

5.2.1 Test Writing Component

The test writing component provides a number of classes that are used to create the test suite. In order to gain access to these classes, add an assembly reference to twEuropa.dll in the Visual Studio project containing the test suite. The classes used to write the test cases reside in the Europa namespace. They are discussed in detail in sections 5.3.1 and 5.3.2.

5.2.2 Test Execution Component

The test execution component provides a graphical user interface for loading, selecting and executing the test cases. A screenshot of this GUI appears below.

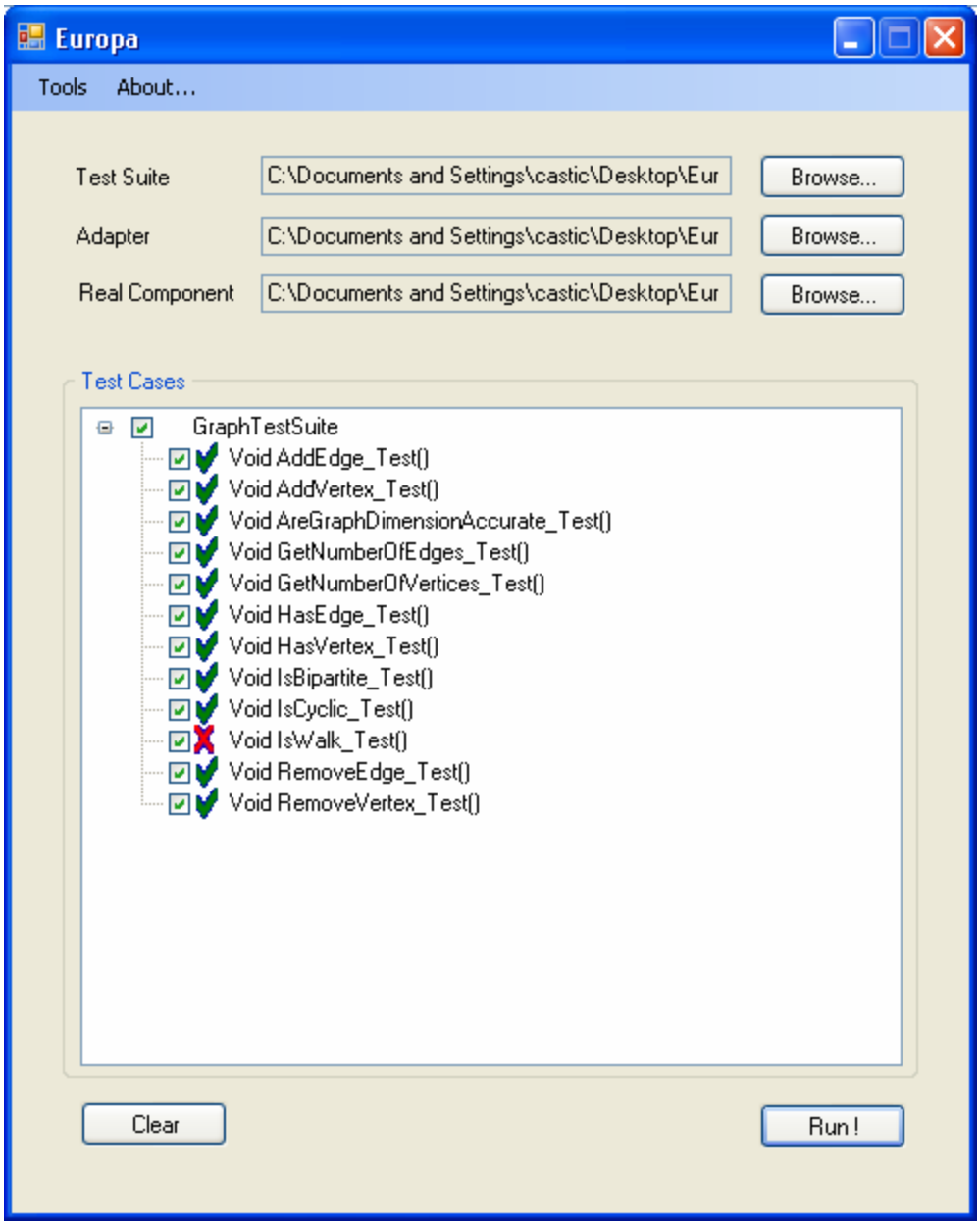

Figure 5.1: Test Execution Component User Interface

Users load the test suite, the adapting information file and the real component to be tested by browsing to the location containing each item. Once these items have been loaded, the framework displays a tree-view of the test cases available in the loaded test suite. The user selects the test cases to be executed and clicks Run!. The framework executes the selected test cases and reports on the outcome of each executed test case. A green checkmark indicates that the test case passed. A red 'X' indicates that the test case failed. In the event that an exception is thrown by the real component during the execution of a test case, an 'E' is assigned to the test case that encountered the problem. In all cases, placing the mouse over a test case causes the framework to

display a tooltip offering more information about the test case, the reason for failure or exception.

5.3 Writing Test Cases

Generally speaking, writing a test case boils down to manipulating the state of the component being tested, and/or executing an algorithm implemented by the component. This is accomplished by calling methods implemented by the component. The return value of the method being targeted by the test case is then compared to the known expected result. The following sections explain how test cases are written in Europa.

5.3.1 Creating a Test Suite

Creating a test suite involves creating two classes: a virtual component and the actual set of test cases. The virtual component is an abstraction of the real component. Test cases in the test suite are written against the virtual component.

The framework provides an abstract base class (Europa.VirtualComponent) that a virtual component must inherit from. This base class provides a mechanism for the framework to hand a driver reference to the virtual component at test execution time. The methods in this class (Europa.VirtualComponent.SetDriver and Europa.VirtualComponent.GetDriver) provide the necessary implementation and cannot be overridden.

The methods implemented by a virtual component must forward the method calls to the driver for execution. Methods are forwarded to the framework driver by calling the method IFrameworkDriver.ExecuteMethod on the inherited member VirtualComponent.driver. This method has the following signature:

```
object ExecuteMethod(string methodName, params object[ ] parameters)
```

The first parameter is a string that is used to uniquely identify the virtual method. It is important to ensure that this string exactly matches the name of the virtual method being executed because this parameter is used to map the virtual method to the real method during test execution. The second parameter is a variable-length list of parameters to be passed to the real component. In the case in which the method takes no parameters, this argument should be set to null or simply be left out. Notice that this method's return value is an object, which must then be casted down to the expected value in the case when the return value of the virtual method is other than void.

Below is a partial declaration of our VirtualGraph.

```
using Europa;
public class VirtualGraph : VirtualComponent
{
    // clear the graph.
    public void Clear()
    {
        driver.ExecuteMethod("Clear", null);
    }

    // add an edge.
    public void Add_Edge(string v, string w)
    {
        driver.ExecuteMethod("Add_Edge", v, w);
    }

    // determine if the graph is bipartite.
    public bool Is_Bipartite()
    {
        object retValue = driver.ExecuteMethod("Is_Bipartite", null);
        return (bool) retValue;
    }
    // class declaration continues...
}
```

The next step is writing the test suite. Creating a test suite involves creating a class that implements the ITestSuite interface. In addition, a test suite may have the optional [TestSuite] custom attribute, which can be used to provide a description of the test suite to be displayed to the user at test execution time. Below is a code snippet of the GraphTestSuite, which shows the implementation of both methods of the ITestSuite interface.

```
[TestSuite("test cases for a graph component")]
public class GraphTestSuite : ITestSuite
{
    // called by the framework prior to test case execution
    // so that any needed initialization can be performed.
    public void Setup()
    {
         IFrameworkDriver driver = m_vc.GetDriver();
         string target = driver.GetTargetAssembly();
         if (target.Equals("GraphLib.dll"))
         {
             m_isCustom = true;
         }
    }

    // retrieve a reference to the virtual component.
    public VirtualComponent GetVirtualComponent()
    {
        return m_vc;
    }
}
```

In our sample test suite, method ITestSuite.Setup is being used to help determine, at test-execution time, whether or not the test suite contains a custom test case. We discuss custom test cases in section 5.3.2.1.

5.3.2 Creating Test Cases

A test case is created by adding a method to a test suite with the custom attribute [TestCase]. This custom attribute takes an optional description as a parameter. Test cases do not have any parameters and must not return a value. Test cases are written against a virtual component. The general idea is to perform some manipulations on the component, and then assert on a condition. The framework provides Europa.Test to perform this assertion. This class contains one static method whose signature is as follows:

static void Assert(bool condition, string message)

The first parameter is the condition that must be met, and the second parameter is a message to be displayed by the framework in the case when the assertion fails. This message provides information to the user regarding the reason why the test case failed. Below is a code snippet showing one of the test cases in our sample test suite.

```
[TestCase("determine if graph is bipartite")]
public void IsBipartite_Test()
{
   m_vc.Clear();
   m_vc.Add_Edge("1", "2");
   m_vc.Add_Edge("2", "3");
   m_vc.Add_Edge("3", "4");

   Test.Assert(m_vc.Is_Bipartite(), "failed: graph is bipartite");

   m_vc.Add_Edge("1", "3");
   Test.Assert(!m_vc.Is_Bipartite(),
               "failed: graph is not bipartite");

}
```

Member GraphTestSuite.m_vc is of type VirtualGraph and represents our virtual component. In our example, it is defined with class scope. The test case begins by clearing the graph to bring it to a known state. Edges are then added to it, and the algorithm to determine if a graph is bipartite is executed. The return value of the method call is compared to the expected value. If the test case fails, the message provided to the assertion that failed will be displayed to the user.

5.3.2.1 Custom vs. Generic Test Cases

By providing a virtual component as an abstraction of a real component to be tested, Europa achieves a complete separation between the process of writing the test cases and the process of executing those test cases on the real component. Since the actual component to be tested may not have been identified at the time of writing the test cases, there are, of course, limitations on what can be adapted and what cannot be. In particular, it would not be possible to write a test case for a method in a real component that takes a custom type as one of its parameters because this custom type (or even its existence) may be unknown at the time of creating the test suite. The framework provides a mechanism so that custom test cases can be written to target real component directly, thus bypassing the virtual component layer. Doing so, of course, requires that the real component for which the custom test case is being written be available at the time of writing the test case so that the test suite can be linked to the custom type that prompted the need for the custom test case.

The graph component ASTOR.GraphLib provided with the sample package is one component that requires a custom test case. Method ASTOR.GraphLib.InsertVertex has the following signature:

void InsertVertex(ASTOR.Vertex v);

where ASTOR.Vertex is a custom type defined in the real component's assembly. The code snippet below is responsible for determining if a test case needs to be run as a custom test case.

```
// constructor.
public GraphTestSuite()
{
   m_vc = new VirtualGraph();
   m_customTargets = new List<string>();
   m_customTargets.Add("GraphLib.dll");
}

// performs needed initialization.
public void Setup()
{
   IFrameworkDriver driver = m_vc.GetDriver();
   string target = driver.GetTargetAssembly();
   if (m_customTargets.Contains(target))
   {
      m_isCustom = true;
   }
}
```

In its constructor, the GraphTestSuite class creates a list of all the real component assemblies for which the test suite contains one or more custom test cases. In our sample package, there is one such component. When method GraphTestSuite.Setup is

called, the test suite queries the framework driver for the name of the assembly containing the real component currently being tested. The test suite then searches in its list of real component names for which one or more test cases must be run in custom mode to determine if the current component being tested is in that list. The relevant test cases will then be run in custom mode if needed, depending on the outcome of this search. Below is a code snipped showing how method ASTOR.GraphLib.InsertVertex was written to execute in custom mode when appropriate.

```
// test case for adding a vertex.
public void AddVertex_Test()
{
   m_vc.Clear();
   AddVertex("1", m_isCustom);

   Test.Assert(m_vc.Has_Vertex("1"),
               "failed: vertex was not added");
}

//helper funciton used to add a vertex to the graph.
private void AddVertex(string v, bool custom)
{
   if (!custom)
   {
       m_vc.Add_Vertex(v);
   }
   else
   {
       IFrameworkDriver driver = m_vc.GetDriver();
       string target = driver.GetTargetAssembly();
       if (target.Equals("GraphLib.dll"))
       {
          ASTOR.GraphLib graph =
              (ASTOR.GraphLib)driver.GetRealComponent();
          ASTOR.Vertex newVertex = new ASTOR.Vertex(v);
          graph.InsertVertex(newVertex);
       }
   }
}
```

If running in custom mode, the test to add a vertex bypasses the virtual component and operates directly on the real component.

5.4 Executing Test Cases

Executing the test cases involves launching Europa's test execution component (Europa.exe). Loading the required items and selecting the test cases to run is covered in section 5.2.2. One aspect that requires further discussion is the creation of the Adapter. This feature of the framework is described in the next section.

5.4.1 Performing Method Adaptation

Europa derives its knowledge on how to map method calls on the virtual component to methods on the real component from a file (the Adapter). This file is read at test execution time, and contains all method mapping information. One such file exists for each real component being tested. The chosen format for the contents of this file is XML. It is possible for users to construct this file manually. It is also possible for users of the framework to create their own application to help create this file. Europa provides a tool that can be used to create the Adapter. In order to use the tool, users must first load the test suite and the real component for which the file is to be constructed. At that point, users can launch the Method Mapping Wizard, which is found under the Tools menu option. Below is a screenshot of the Method Mapping Wizard being used to create the Adapter for the ACME.Graph component of our sample package.

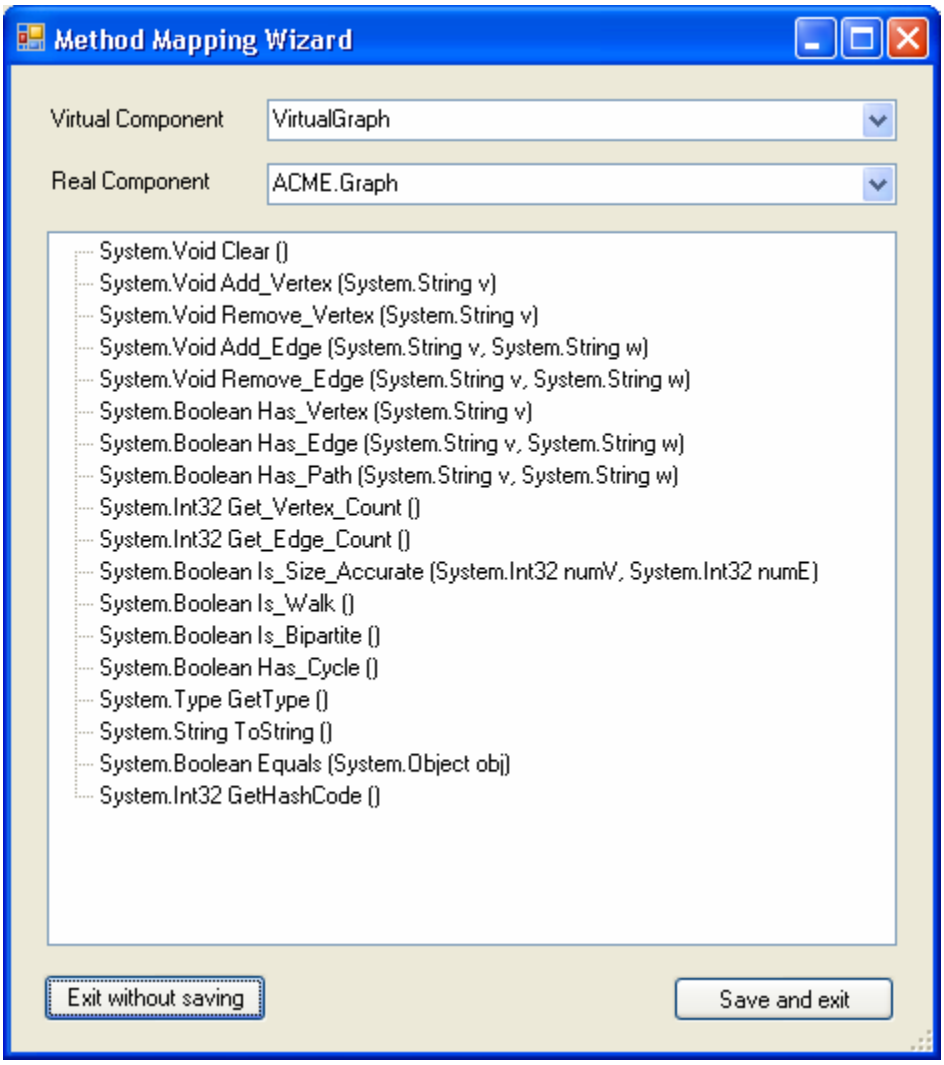

Figure 5.2: Method Mapping Wizard (MMW) User Interface

The user starts by selecting the virtual component, real component pair for which the Adapter is to be created. Once these selections have been made, the Method Mapping Wizard (MMW) presents a tree view of all the virtual methods found in the virtual component. In order to define a method mapping, the user must right-click on the name of the virtual method for which the mapping is to be defined, and select "Add method mapping" from the context menu. Removing a previously defined method mapping works in a similar way, except that the "Delete method mapping" option is chosen instead. Below is a screenshot of the dialog for defining a method mapping.

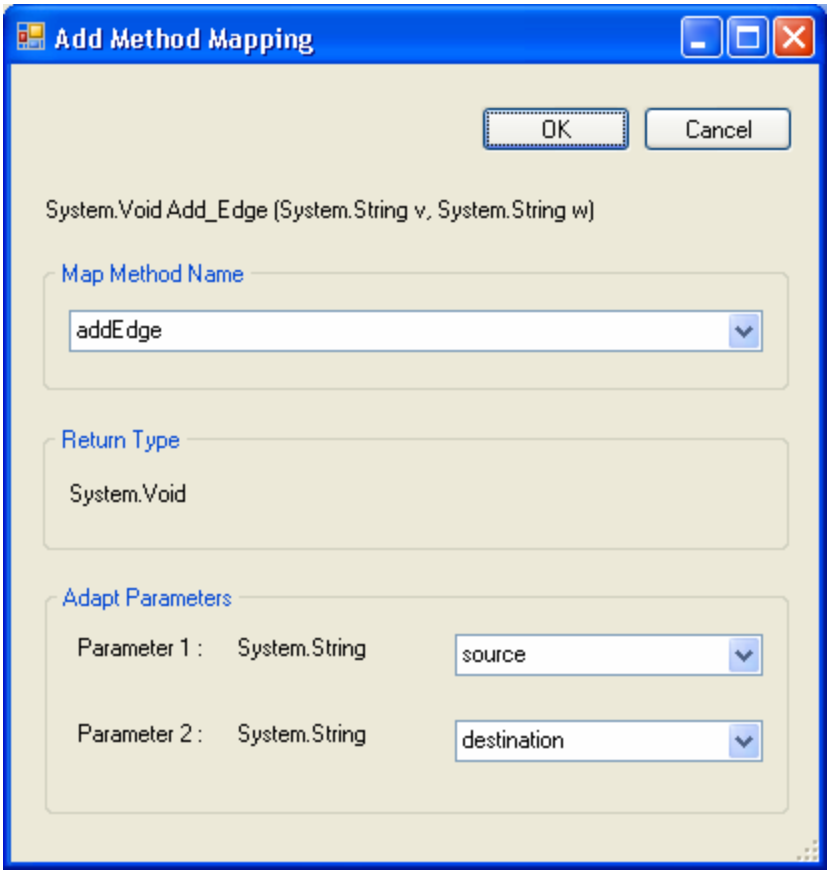

Figure 5.3: Defining a Method Mapping

The "Map Method Name" combo box contains the names of all the public methods in the real component. Once the user selects the appropriate method that the virtual method should be mapped to, the return type and parameter list information is presented to the user. Below is a screenshot of the MMW once all relevant mappings in our example have been defined.

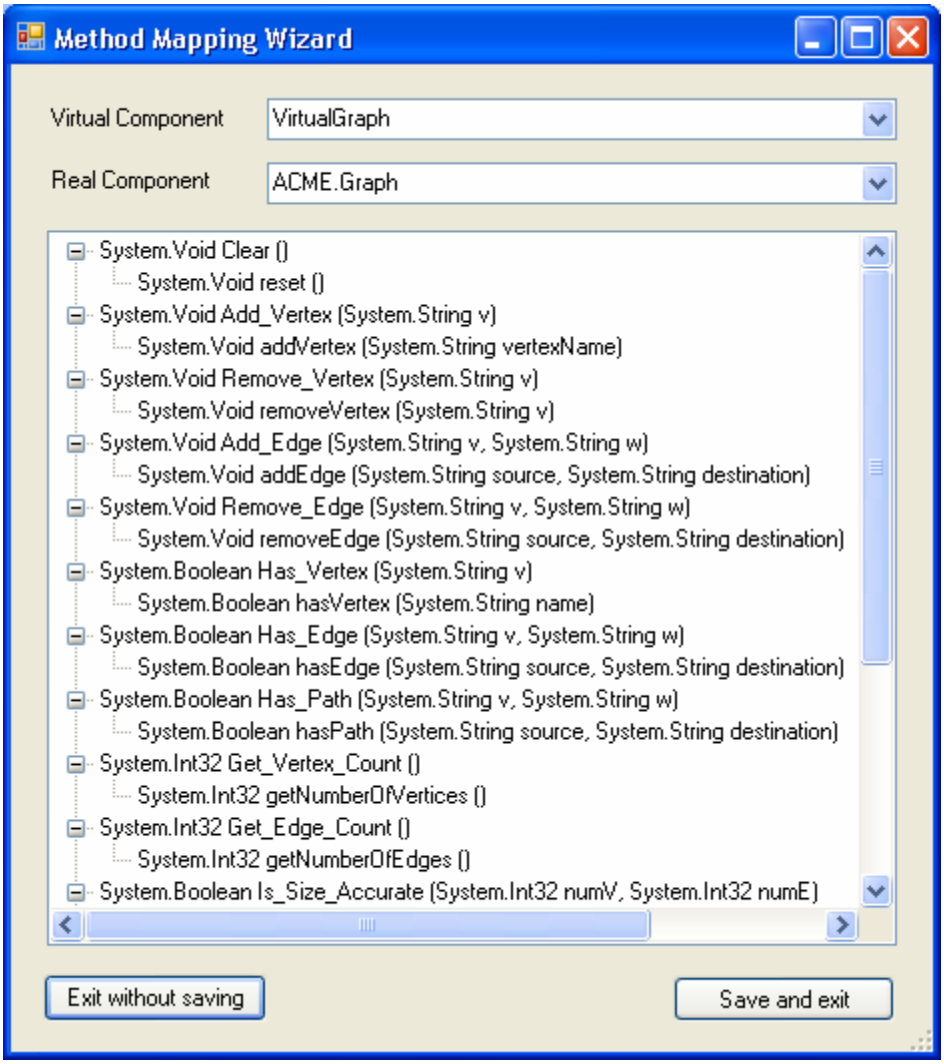

Figure 5.4: View of All Method Mappings

The final step in creating the Adapter for the ACME.Graph component is to click ‘Save and exit’ to store the method mapping information.

5.4.2 Running the Test Cases

Refer to section 5.2.2 for details on executing the test cases.

5.5 Troubleshooting

This section presents some issues that might arise when using the framework.

PP-1 When loading the virtual component or the real component, the framework displays an error message saying that it could not file the assembly or one of its dependencies.

Solution Make sure that both the virtual component and the real component are located in the same directory as the framework test execution component (Europa.exe).

PP-2 When loading the Adapter file, the framework displays an error message saying that it could not find the file “XMLAdapterSchema.xsd”.

Solution This file is used to validate the format of the Adapter file being loaded. Make sure that the Adapter file being loaded and file “XMLAdapterSchema.xsd” are located in the same directory.

PP-3 When loading the test suite assembly, the test suite and the test cases in it are not displayed in the tree view.

Solution The test suite must implement the Europa.ITestSuiteInterface. This is used by the framework to determine if a type defined in an assembly is a test suite or not.

6 Conclusion

This project demonstrates the feasibility of using the mechanism of reflection to design and implement an automated test framework suitable for unit testing in the component-based software development paradigm. In particular, this project shows that Microsoft’s .NET platform meets the technology requirements for the development of such framework.

The ability to adapt method invocations from a virtual component to a real component is the main innovation in this new framework. A key assumption in our framework is that the process of writing the test cases (which includes the creation of the virtual component) is completely decoupled from the real component search phase. In other words, users of the framework write the test cases with an understanding of the desired functionality of the real component (coded as test cases against a virtual component), but without any knowledge about the interfaces provided by real component implementations. This necessarily imposes some limitations on how much interface adaptation can be achieved. In particular, method adaptation is limited to methods in the real component whose parameters are .NET built-in types. The .NET built-in types constitute the “common language” among all .NET components.

For cases involving real components containing one or more methods with parameters of a custom type, the framework provides a mechanism for bypassing method adaptation, so that test cases operate directly on a particular real component. In this case, the operation of the framework is equivalent to how traditional automated test tools work.

7 Future Work

Components are highly specialized units of software. For this reason, it is expected that the degree of similarity among the interfaces of different implementations of the same component will be high. One area of improvement of our framework is allowing for interface adaptation on real components where the differences between the interfaces are greater. This includes the following:

- Allowing for method adaptation when the number of parameters between the virtual component and the real component is different. Currently, Europa requires that corresponding methods have the same number of parameters, although they can be rearranged in any order. The case when the number of parameters in the virtual method is greater than the number of parameters in the real method can be easily handled. When the number of parameters in the virtual components is less than what is expected in the real component, a mechanism for the user to enter definitions for the missing parameters at the moment of defining the method mapping can be provided. The current version of Europa requires the use of a custom test case for dealing with differing number of parameters.

- Allowing for mapping one virtual method call to multiple real method calls. The current version of Europa allows one-to-one method mappings only. Adding support for one-to-many method mappings involves adding a mechanism for retaining intermediate results between calls, and possibly pass an intermediate result from a method call as a parameter to a subsequent call. Users of the current version of Europa can use custom test cases to deal with real components that require one-to-many method mappings.

- Allowing for testing on real components that do not provide a default constructor. Europa requires the ability to create instances of the real component via its default constructor. The .NET reflection API supports the creation of types using any of the constructors available. This capability can be accomplished by allowing users to pick the constructor to be used at the time of creating the method mappings. This requires the implementation of the inline parameter definition feature discussed in the first bullet above.

8 Appendices

8.1 Adapter XML Schema

This is the schema of the XML adapter. When loading an adapter file, the framework first checks to make sure that the file being loaded conforms to this schema.

```
<?xml version="1.0" encoding="utf-8" ?>

<!--
   Christian Castillo
   cac2763@ca.rit.etu
   Rochester Institute of Technology
```

```
   This XML schema is used to validate the XML file containing
   the method adapting information used by Europa's XMLAdapter.
-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://europa.org/XMLAdapterSchema.xsd"
  targetNamespace="http://europa.org/XMLAdapterSchema.xsd"
  xmlns:target="http://europa.org/XMLAdapterSchema.xsd"
  elementFormDefault="qualified">

    <xs:element name="AdapterInfo" type="target:AdapterInfoType"/>

      <xs:complexType name="AdapterInfoType">
            <xs:sequence>
                  <xs:element name="MethodMappings"
                   type="target:MethodMappingsType"
                   minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="VirtualComponent" type="xs:string"
                          use="required"/>
            <xs:attribute name="RealComponent" type="xs:string"
                          use="required"/>
      </xs:complexType>

      <xs:complexType name="MethodMappingsType">
            <xs:sequence>
                  <xs:element name="MethodAssociation"
                   type="target:MethodAssociationType"
                   minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>

      <xs:complexType name="MethodAssociationType">
            <xs:all>
              <xs:element name="VirtualMethod"
                  type="target:VirtualMethodType"
                  minOccurs="1" maxOccurs="1"/>

              <xs:element name="RealMethod"
                  type="target:RealMethodType"
                  minOccurs="1" maxOccurs="1"/>
            </xs:all>
      </xs:complexType>

      <xs:complexType name="VirtualMethodType">
            <xs:sequence>
              <xs:element name="Parameters"
               type="target:ParametersType"
               minOccurs="1" maxOccurs="1"/>

               <xs:element name="Return" type="target:ReturnType"
                minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="Name" use="required"/>
      </xs:complexType>

      <xs:complexType name="RealMethodType">
```

```
            <xs:sequence>
              <xs:element name="Parameters"
               type="target:ParametersType"
               minOccurs="1" maxOccurs="1"/>

              <xs:element name="Return" type="target:ReturnType"
               minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="Name" use="required"/>
      </xs:complexType>

      <xs:complexType name="ParametersType">
            <xs:sequence>
              <xs:element name="Param" type="target:ParamType"
               minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>

      <xs:complexType name="ParamType">
            <xs:attribute name="Name" type="xs:string" use="required"/>
            <xs:attribute name="Type" type="target:DataType"
                          use="required"/>
      </xs:complexType>

      <xs:complexType name="ReturnType">
            <xs:attribute name="Type" type="target:DataType"
                          use="required"/>
      </xs:complexType>

      <xs:simpleType name="DataType">
            <xs:restriction base="xs:string">
                  <xs:enumeration value="System.Boolean"/>
                  <xs:enumeration value="Systen.Byte"/>
                  <xs:enumeration value="System.SByte"/>
                  <xs:enumeration value="System.Char"/>
                  <xs:enumeration value="System.Decimal"/>
                  <xs:enumeration value="System.Double"/>
                  <xs:enumeration value="System.Single"/>
                  <xs:enumeration value="System.Int32"/>
                  <xs:enumeration value="System.UInt32"/>
                  <xs:enumeration value="System.Int64"/>
                  <xs:enumeration value="System.UInt64"/>
                  <xs:enumeration value="System.Object"/>
                  <xs:enumeration value="System.Int16"/>
                  <xs:enumeration value="System.UInt16"/>
                  <xs:enumeration value="System.String"/>
                  <xs:enumeration value="System.Void"/>
            </xs:restriction>
      </xs:simpleType>

</xs:schema>
```

8.2 Defects Found During the Test Phase

This section contains a list of software defects found during the testing phase of the development effort. Each trouble report (TR) is assigned a unique ID.

TR-01 The Run! button is enabled, even though the user has not loaded an adapter file.

Resolution Tests cannot be executed without the adapting information. The code was modified so that the Run! button is enabled only after the user has loaded the test suite, the adapter file, and the real component.

TR-02 After defining the method mappings in the Method Mapping Wizard, the MMW is dialog is not dismissed after saving the adapting information.

Resolution This is an omission. The Close() method was added to the handler to dismiss the dialog after the adapting information is saved.

TR-03 Methods of the base class of a virtual component (i.e. methods of class Europa.VirtualComponent) show up in the “Add Method Mapping Dialog”.

Resolution This is an inconvenience. The methods in the base class interface are not meant to be adapted. Code was added to prevent these methods from appearing in the “Add Method Mapping Dialog”.

TR-04 After selecting and executing a set of test cases, the results for those test cases remain in the UI after the user unselects those test cases and executes a different set of test cases.

Resolution The code was modified so that test cases not being executed during the current run are reset to a default state, which involves clearing out any results from previous runs.

TR-05 Rearranging the order of the parameters when defining a method mapping seems to have no effect. At test execution time, the method is executed on the real component with the parameters in the same order as defined in the virtual component, ignoring the reordering done when defining the method mapping.

Resolution The reordering was incorrectly being done on the parameter names instead of on the values defined in the call coming out of the virtual method. The problem was corrected.

TR-06 Users are allowed to launch the Method Mapping Wizard even though they have not loaded the required components.

Resolution Changes were made so that the user is allowed to launch the Method Mapping Wizard only after loading the virtual and real components.

9 References

[1] M. D. McIlroy. Mass Produced Software Components. Proceedings of the NATO Software Engineering Conference, Garmisch, Germany (1968) 138-155.

[2] A. Bertolino, A. Polini. Re-thinking the Development Process of Component-based Software. Proceedings of the 9th IEEE Conference on Engineering of Computer-Based Systems, pages 7-10, Lund, Sweden, April 2002. IEEE Computer Society.

[3] P. Vitharana. Risks and Challenges of Component-Based Software Development. Communications of the ACM, Vol. 46, No. 8, August 2003, pages 67-72.

[4] R. C. Seacord, S. A. Hissam, K. C. Wallnau. Agora: A Search Engine for Software Components. Technical Report CMU/SEI-98-TR-011.

[5] Reuse Processes. Software Engineering Standards Committee of the IEEE Computer Society. June 1999.

[6] A. Bertolino and A. Polini. A Framework for Component Deployment Testing. Proceedings of the 25th International Conference of Software Engineering, pages 221-231, Portland, Oregon, 2003. IEEE Computer Society.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[8] M. J. Harrod. Testing: A Roadmap. Proceedings of the Conference on The Future of Software Engineering, pages 61-72, Limerick, Ireland, 2000. ACM Press.