Department of Technology, Mathematics and Computer Science

DEGREE PROJECT

2004:DS15

Plugin-Based Automated Testing Tool for .NET Assemblies

Olle Johansson

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

Plugin-Based Automated Testing Tool for .NET Assemblies

Olle Johansson

Summary

This work explains some of the problem when designing a plugin based testing tool for .NET assemblies. The reason for using modules to test is that they can easily be replaced. It is also simpler to write one new module rather then rewriting the whole testing tool. The different types of modules are explained and motivated. The modules are; Loader, Test generator, Value Generator, Tester, Test result viewer and Result saver. The work resulted in a testing tool prototype that was able to find simple errors in tested assemblies.

Publisher:	University of Trollhättan/Uddevalla, Department of Technology, Mathematics and Computer Science, Box 957, S-461 29 Trollhättan, SWEDEN Phone: + 46 520 47 50 00 Fax: + 46 520 47 50 99 Web: www.htu.se		
Examiner:	Robert Feldt		
Advisor:	Richard Torkar		
Subject:	Computer Science	**Language**:	English
Level:	Advanced	**Credits**:	10 Swedish, 15 ECTS credits
Number:	2004:DS15	**Date**:	May 25, 2004
Keywords	Software engineering, Application testing, Random testing, Black-box testing, Dot NET assembly testing,		

Preface

I want to thank my examiner Dr. Robert Feldt for his feedback and guidance. He has helped me in many path directing choices.

I would also like to thank my supervisor Richard Torkar who has helped me in many moments of uncertainty and giving me great feedback.

Contents

List of symbols

Plugin -	Plugin will in this work consist of a dynamic loaded library (DLL) file that contains all the information to solve one specific task involving testing.
Assembly -	Assemblies is how the compiled files in .NET are packaged. An assembly can be an EXE file or a DLL file.
Software Engineer -	A software engineer is a person that is working with the development of software and is trying to improve the quality.

1 Introduction

Building good software involves verification and validation (V & V). Since many software companies have strict deadlines testing is often something that is neglected, witch in turn results in bad software for customers. If it would be possible to make the computer do some of the work it would save time, witch could result in better tested software. To let both an automated tool and a software engineer test the software is a good approach, since they use different techniques. There will be a deeper discussion of the advantages and disadvantages of automated testing in chapter 4. By pushing for easy-to-use-tools, for testing software, there will hopefully exist software of higher quality in the future.

Validation: Are we building the right product?
Verification: Are we building the product right?

Figure 1.1: Verification & Validation

It is not easy to auto generate validation tests since it involves what the customer wants. The only way to know what the costumer want is to read the documentation and have a dialog with the customer, witch is not easy to automate. Verification is however possible to automate to some extent. In chapter 3.1 and 3.2 there will be a discussion about white box and black box testing and what parts that can be tested by a computer, or at least with good help from the computer.

The intention of this report is to explain how to make it possible to write testing applications that will ease the testing work for the developer. The testing was done on .NET assemblies and it focused on automating black box testing, but a lot of the content in this work can be applied on white box testing as well. We wanted to see if it was possible to get the information from an assembly needed to generate tests. The goal is to not be dependent on a single testing technique, but rather using plugin to test in different ways. We wanted to se if it was possible to bind plugins at runtime, which could be used to perform different testing methodologies. We are using .NET and C# to build the tool, since it is a modern programming language that is commonly used and is likely to be used even more in the future.

2 Method

To get the knowledge of testing methods and past work in the field of automated testing, there have been literature studies. A design was created that allowed the testing process to be done with different plugins. That design was used to build a prototype of the product. The prototype was built using Microsoft Visual Studio .NET. There has also been some evaluation on the testing tool. It has been tested for the ability to use different plugins. It has also been tested for the ability to detect errors in the tested software. There is also a discussion about how some testing metrologies do not work in this design.

3 Background

A lot of software projects today are poorly tested. Studies have shown that testing is one of the first things that are neglected then the time schedule is too limited [15]. There are even developers that think that it is easier to make the users test the software for them [15]. Most people involved in software engineering agree that the easiest and cheapest way of developing software is to find the defects early [14, 4]. For large companies with many customers a late detection of a fault can cost several million dollars and make serious harm to the company's trustworthiness [4]. Since the software has to be ready by deadline, tested or not, we believe that fast and easy-to-use tools would be of good help. Testing should not stand in the way of getting the work done; rather make the developer create more robust software.

One commonly used software for testing is NUnit [7] and other similar unit testing software. Unit testing is a simple way of writing tests and are known to detect faults early in the development process. It however depends on the developer's ability to write relevant and valid tests.

One commercial product from Parasoft [8] can make both static and dynamic analysis (se chapter 2.1 for more information) and auto generate unit tests for the .NET platform. This can be a good candidate to solve the problem specified in this work, but because of its price it has not been tested. One other problem is that it is not open source and there is no technical documentation available, so it is very hard to compare with other tools.

QuickCheck [9] is another tool that is used to test Haskell applications. QuickCheck is dependent of a specification of the tested application written in Haskell code. When doing tests, QuickCheck will generate random numbers that are used as arguments. If the output values follow the specification the test result is valid.

3.1 White Box vs. Black Box Testing

There are two main approaches when testing software, black box and white box. When using white box the source code are used to generate good tests. In white box testing there are two main branches; static and dynamic analysis. When using static analysis the source code is analyzed without running the application to find errors [14]. This method is not used in this work. When using dynamic analysis the source code is used to generate tests and then used to test the application in runtime [14]. Dynamic analysis could possible be used in the test application described in this work, but it has however, not been tested.

Black box testing does not involve the source code. The idea is to only look at the interface that interacts with the surrounding classes. Input values are generated and compared with the output values. One problem that often appears with black box testing is the “oracle” problem [3]. The oracle is the one that knows when the output value matches the input value. In many cases this has to be a software engineer.

Since it often is easier to generate tests through black box methodology, this is where the focus will be in this work. The reader should however keep in mind that the application that is described in this report is not limited to black box testing; it is used to keep the application simple.

3.2 Black Box Testing

Different test methodologies are suitable in different situations. I will explain some of the most common black box techniques and when they are suitable. Later in this work there will be examples of implementations of them so it is important to understand what they mean for the testing. One common problem for many of the black box techniques is that it is hard to generate input values when the input type is a user defined class. This would involve making objects of the argument types, and use as argument to the tested method. To do this it is necessary to call the argument class constructor and possible call methods on it to get it into an expected state. This problem is not taken care of in this work; we assume input values are of simple types like integers and strings.

3.2.1 Random Testing

Random testing is a simple method to generate input values that can be used in the testing. In its simplest form it generates random values through the entire input domain i.e. all valid input values. It does not take into consideration that errors are more likely to appear at a cretin range of the input domain. This is also one of its strength, since it is a good tool to use to gather statistical information about how many errors is left in the tested application [3]. One large drawback when using random testing is the need for an oracle that knows all the answers and can decide if the output values correspond to the input values [3].

3.2.2 Anti-Random Testing

Anti-random testing shares many similarities with random testing. The difference is that anti-random testing guarantees that the random values are spread through the whole input domain by giving every new value the maximum distance to the previous values [5]. This has shown to be more effective than random testing, especially when the number of tests is limited [13].

3.2.3 Partitioning Testing

Since the input domain is very large in most cases, it is useful to divide it into smaller sections called partitions. Each partition must be tested, but to test the same partition with different values are not considered useful. This can be a very effective way of testing software, but the problem is that it is hard to find the partitions with automated tools [2]. It is shown that when partition testing is done properly, it is proven to perform at least as well as random testing [1, 6].

3.2.4 Boundary Value Analysis

Boundary value analysis (BVA) is a subsection of partition testing. It concentrates the testing at the boundaries of the partition. The test should if possible be above, below and on the boundary. BVA have shown in experiment to be more effective than partition testing [12].

3.2.5 Cause-Effect Graphing

By reading the natural language in the specification it can be formed a Boolean graph that will give good test cases. The main problem with this approach is that since reading the specification is needed so it is not very suitable for automated testing [15].

3.2.6 Exhaustive Testing

Exhaustive is an easy but time consuming way of testing software. Every value in the input domain is used to test the software. This is not possible except in very small test cases or in very critical parts of an application [15].

4 Automated Testing

This chapter will introduce how testing can be automated in general, with focus on how it can be done with the .NET framework. Deeper information of how it was done in the implementation proposed in this work will be presented in chapter 5, 6 and 7.

By using automated tests the software engineer will hopefully test the software more often since it involves less work. The fact that the testing mechanism does not act like a human being can be an advantage. The software engineer does in many cases not see all the possibilities the application can be used in. An automated testing tool tests all the usage that is possible, even those that not was expected. This can also be a disadvantage for the automated test. It does not understand the purpose and does not test it in the way it will be used. This is especially true when it comes to random testing. Since the test values are random it does not correspond to how a real user would act. One other problem with random testing (and other techniques) is that an oracle is needed [3] to make the decision if the output value is corresponding to the input values. In this work a software engineer is needed to review the test results and decide if they were a success or not. Since automated tests do not test the software the same way as a software engineer would do, to do both will result in better tested software.

4.1 Automated Testing Tool for .NET

The main reason to build a testing tool for .NET is that many different applications can be compiled to a .NET assembly. No matter if the application that should be tested is written in C++, C#, COBOL (COBOL .NET) or Smalltalk (Smaltalk#), it can be tested with a testing tool like that. Many other testing tools can only be used to test application of one specific type. Since the test is done on the assembly there is no need for the source code. This makes it easier to trust commercial of the shelf (COTS) components that usually not give away the source code for testing or reviewing.

4.1.1 Extracting Information from Assemblies

The .NET framework contains a system called reflection that is used to extract metadata from an assembly. Figure 4.1 contains examples of information that can be extracted from an assembly by using reflection. This is everything that is needed to do most types of black-box testing.

```
Namespaces in assembly.
    Classes is assembly.
        Interfaces class is instance of.
        If class is abstract.
        Properties in classes.
           Name of property.
           Property type.
           Read and write protection of property.
        Methods in classes.
           Name if method.
           Method is abstract, final or virtual.
           Method is public, private or protected.
           List of argument name and type.
           List of return type.
```

Figure 4.1: Examples of information that can be extracted with reflection.

Another way to get the information could be to analyze the IL (Intermediate Language) code inside the assembly. IL is the language used inside the assembly and is quite readable for humans. In Figure 4.2 is the IL code for the classic HelloWorld application.

Analyzing IL code would not only give the interface information but also give the inner algorithms of the classes. This would make it possible to make white-box tests. The complicity of this method would however be much higher than using reflection. One way of getting information from the assembly into the application is to use Runtime Assembly Instrumentation Library (RAIL) [10]. RAIL has methods to read and change the implementation of an assembly. It is not even necessary to analyze and understand the IL code, RAIL got tools to analyze the assembly anyway. This is not tested in this work.

```
.method public hidebysig specialname rtspecialname
     instance void  .ctor() cil managed
{
     // Code size       17 (0x11)
     .maxstack  1
     IL_0000:  ldarg.0
     IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
     IL_0006:  ldstr      "HelloWorld"
     IL_000b:  call       void [mscorlib]System.Console::WriteLine(string)
     IL_0010:  ret
} // end of method HelloWorld::.ctor
```

Figure 4.2: Example of a HelloWorld application with IL code.

There also exist tools to reverse engineer the assembly to C# code again. One example is Reflector for .NET [11]. Some information will be lost like names on private variables, but the source code is readable. When C# code are available many white box techniques and tools are available, but this is not covered in this work.

4.1.2 Running Tests

To be able to run unit tests, a method is analyzed to see what arguments it uses. The argument types are used to generate input values as shown in example figure 4.3. That example ends with the oracle deciding if the test was a success or not.

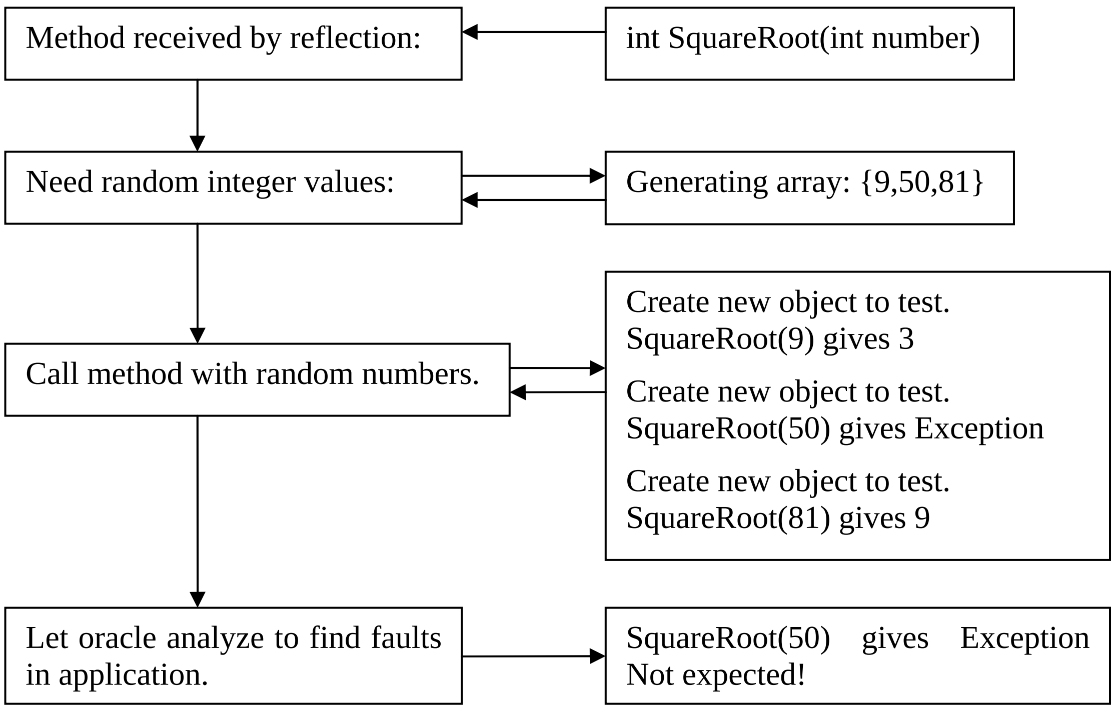

Method received by reflection:
int SquareRoot(int number)
Need random integer values:
Generating array: {9,50,81}
Call method with random numbers.
Create new object to test.
SquareRoot(9) gives 3
Create new object to test.
SquareRoot(50) gives Exception
Create new object to test.
SquareRoot(81) gives 9
Let oracle analyze to find faults in application.
SquareRoot(50) gives Exception Not expected!

Figure 4.3: A simple example of how random testing can be implemented.

In the example only one method call are done on each object. We believe that it can be useful to make more than one method call on the same object. This is needed to se if combinations of method calls can set the tested object in an unexpected state.

It can also be a good idea to allow the tester to set together sequences of method calls that is a high probability to be called. By doing this the testing application can do more complex tests, since the tester have already pointed in the right direction. The tester will in this case be used as an oracle of how the application will be used. An example of that will be shown in the demonstration of the implementation in chapter 7.

One way around the oracle problem can be to use the approach QuickCheck uses. In QuickCheck a specification is written in a programming language. In the case of QuickCheck Haskell is used, but in this case C# would probably be used and compiled into IL code. The specification is used to compare the input values against the output value. If the output value is not according to the specification, a fault is found. This methodology is not used in the implementation in this work; it could however give some advantages.

5 Analysis

There is a need to compare different testing methodologies. When comparing for example Random and Anti-Random testing it would be simple to only write the value generators. Everything else like loading a test case, presenting the test is out of interest in this case. That is why it is good to modularize the testing process and make plugins that do a specific task. This will make it easier to consecrate things that are important. Figure 5.1 shows an example of the plugins and how they cooperate. After the application start the user has the opportunity to select what type of modules are needed. Each plugin can be interactive and can have its own GUI. The test generator can for example ask the tester for advice to make better tests and the result viewer can ask the software engineer for valid tests.

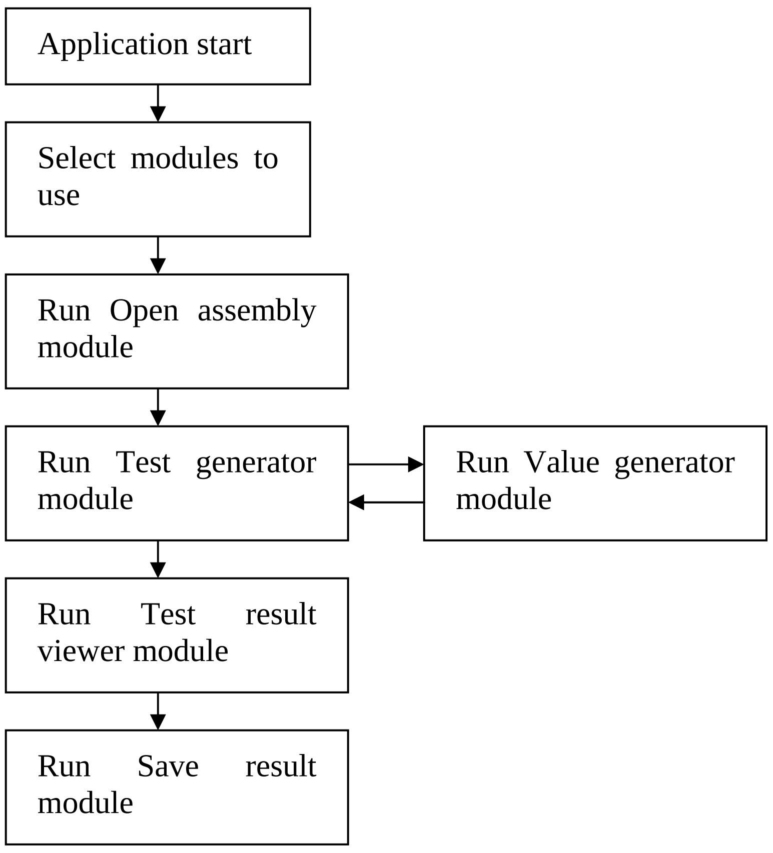


Figure 5.1: Model of how plugins can be used.

Each of the plugins can be replaced by other plugins that do the work in a different way. The main application takes care of the data flow between the plugins.

6 Design

A simplified UML diagram can be seen in Figure 6.1. This shows the interfaces for the modules and an empty implementation of a class. The events are put in the class that raises them to make the diagram easy to read. Other classes that contain the data that are transported between the plugins are not shown on the diagram.

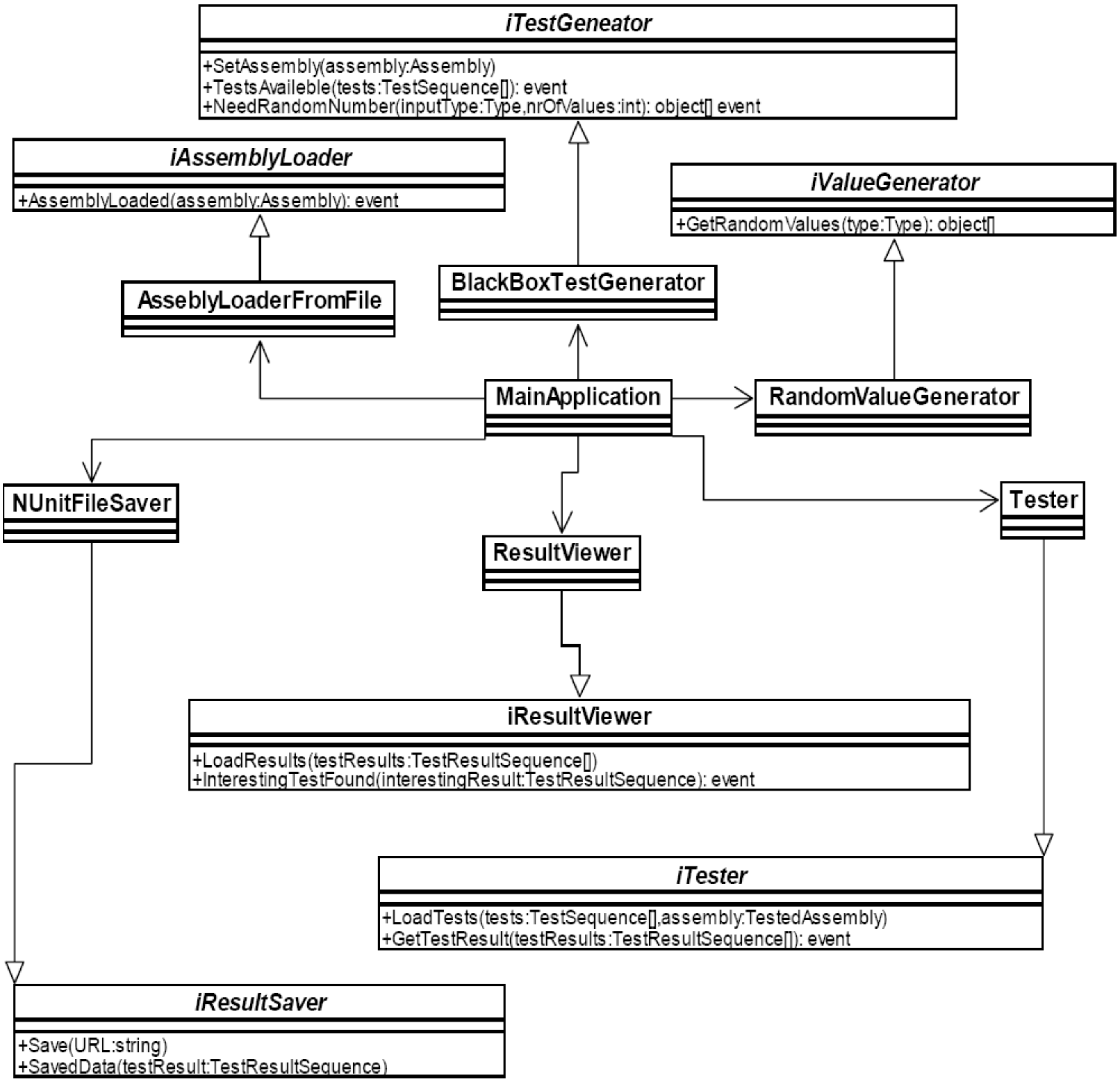

iTestGeneator
+SetAssembly(assembly:Assembly)
+TestsAvaileble(tests:TestSequence[]): event
+NeedRandomNumber(inputType:Type,nrOfValues:int): object[] event
iAssemblyLoader
+AssemblyLoaded(assembly:Assembly): event
iValueGenerator
+GetRandomValues(type:Type): object[]
AsseblyLoaderFromFile
BlackBoxTestGenerator
MainApplication
RandomValueGenerator
NUnitFileSaver
Tester
ResultViewer
iResultViewer
+LoadResults(testResults:TestResultSequence[])
+InterestingTestFound(interestingResult:TestResultSequence): event
iTester
+LoadTests(tests:TestSequence[],assembly:TestedAssembly)
+GetTestResult(testResults:TestResultSequence[]): event
iResultSaver
+Save(URL:string)
+SavedData(testResult:TestResultSequence)

Figure 6.1: Simplified UML diagram to show an overview of the design.

The centre of the application will be a MainApplication class. This will call other replaceable plugins that will do their specific job. One very important idea in this design is to this to use interfaces. By limit a plugin to a specific interface it will be easy to replace them. Figure 6.2 has an example of how the value generator is replaced by different black box test value generators. They are all limited by the value generator interface that makes sure that they are all compatible with each other. The plugins are hidden behind the interface so the real implementation of random values can be a small class or a large component consisting of several classes. The main application will never know the difference. The plugins are loaded at runtime and as dynamic linked library (DLL) files so they are very easy to replace. I will go into more details later of what the interfaces for the different modules contains.

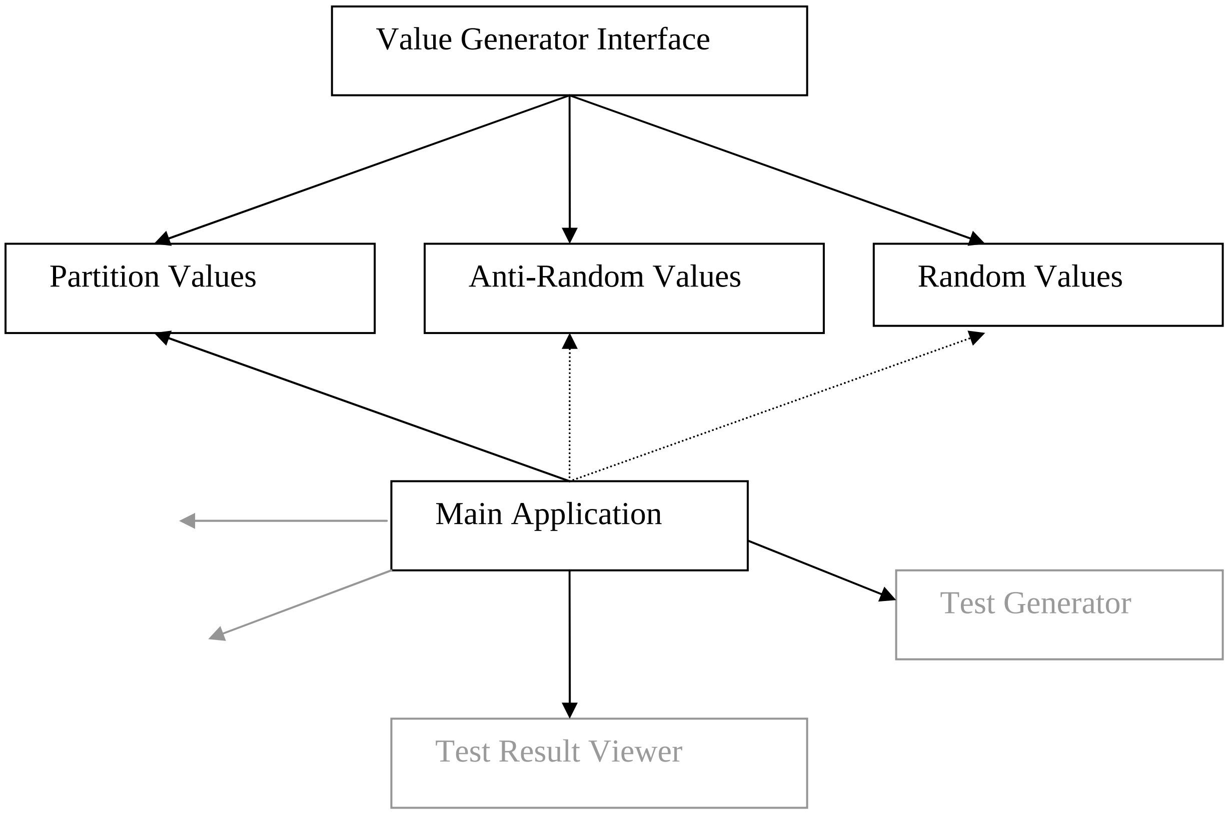


Figure 6.2: The ability to replace plugins that share the same interface.

6.1 Interfaces

Interfaces are a very central part of the suggested solution. This makes it easy to work with the modules as ordinary classes. When a plugin are used in the application it is referenced by its interface type

Ex. 1.

```
iValueGenerator valueGen = new RandomValueGen()
```

Ex. 2.

```
iValueGenerator valueGen = new AntiRandomValueGen()
```

Figure 6.3: Example of how interfaces can be used to reference plugins.

When looking at figure 6.3 we can see how similar the different value generators will appear. They will all get the same methods and properties. In the real application the modules will however not be called by the *new* keyword but rather using the reflection framework since they are not bound to the application until runtime.

6.2 Modules

Here comes a listing of the plugin types and how they are designed.

6.2.1 AssemblyLoader

When the loading is done in a plugin it is easy to change how the assembly is loaded. One simple way is to load the assembly directly from the hard drive, but I could also be useful to load it from CVS or from some kind of database. It would also be possible to write a loader that is pointed to a source file and then compile the source and load the assembly.

All different kinds of assembly loaders should inherit from the interface iAssemblyLoader. This interface is shown in figure 6.4. The only thing this interface contains is that the class must be able to raise an AssemblyLoaded event that returns the assembly that should be tested. The delegate that makes this possible is also put in the same place as the interface. This module is very likely to have a GUI and the GUI should be visible as soon as the class is loaded since there is no show method on the interface. The reason for designing the interface without a show method is that it does dot have to be a GUI in the plugin.

```
public interface iAssemblyLoader
{
    event AssemblyLoaded assemblyLoaded;
}

public delegate void AssemblyLoaded (Assembly asm);
```

Figure 6.4: Interface of the assembly loader plugin.

6.2.2 Test generator

One other part that is good to have in a plugin is the part that make the actually test, a test generator. To split the test generator and the value generator into two different modules gave some advantages. The test generator is responsible for decide what methods will be called and in what order. The value generator is responsible for generating values like random values or anti-random values. When these tasks are in different plugins it is easy to compare for example random testing and anti-random testing results since the test is generated in the same way and only the input values is different. It should be possible to write white box tests by looking at the logic in the assembly. One easier approach is to use black box testing since a black box test generator does not need to know the inner workings of the tested assembly. This work has focused at black box testing to keep it simple.

The interface of the test generator is a little more complex and it is shown in Figure 6.5. The generation starts when the method SetAssembly is called. This makes it possible to analyze the assembly to find interesting test cases. When the test generator needs an input value for a method it raises the NeedInputValue event. This event tells to the main class that it needs an array of values and that it should be in the specified type. By using an array to return values the value generator get the possibility to spread the values. There will be a deeper discussion of this later in the section about the value generator.

When the generation of the tests is done a new event is raised. This is the TestsAvaileble event and it will return an array of TestSequence. Each TestSequence is a sequence of method calls that will be run on the same object. This makes it possible to get the object in an unexpected state by combining different method calls.

```
public interface iTestGenerator
{
   void SetAssembly(Assembly testedAssembly);

   event TestsAvaileble testAvaileble;

   event NeedInputValue getInputValue;
}

public delegate void TestsAvaileble(TestSequence[] tests);

public delegate object[] NeedInputValue(Type inputType,
                                        int numberOfValues);
```

Figure 6.5: Interface of the test generator plugin.

If a white box test generator would be created it could have used the same interface. The main difference would be that is does not use the NeedInputValue event. The white box test generator will in that case use the tested algorithm to decide witch values will be used. This is needed if for example the white box generator wants to test all statements in the tested application. The lack of usage of the NeedInputValue event is however not a problem in.

6.2.3 Value Generator

As mentioned previous in the text the value generator is in a different module and apart from the test generator. The test generator will ask the value generator for test values but it should not care if the values come from random, anti-random, BVA et cetera. This is to generate almost identical tests that only alter the input values.

The test generator is mostly used by the black box test generators. The input values in white box techniques often is based on internal structures of the assembly and is therefore hard to predict by an outside module.

The figure 6.6 shows the interface for the value generator. The only method takes the type used to generate values and how many values that are wanted as arguments. The method returns an array of objects and those objects should be valid instances of the type asked for. The size of the array should be the same as the number of values asked for. The reason why it returns many values at the time is that the values can be related. When using an anti-random generator there should be maximum distance between each value and when using BVA all borders should be tested. If one value at the time is asked for there is no guarantee that the same value is not returned each time.

```
public interface iValueGenerator
{
   object[] GetRandomValues(
     Type randomType,
     int numberOfValues);
}
```

Figure 6.6: Interface of the value generator plugin.

6.2.4 Tester

One other part of the testing system is the actual tester plugin that executes the tests. In its simplest form it only creates a class from the assembly and executes the method calls. A more complex plugin variant of the tester could be one that adds the test to a cluster and received the result back to get decentralized running of the tests. The ability to change how the tests are run without to change the test algorithm can be very useful when the size of the tests grows.

The tester module use reflection to create an object of the type that should be tested. This is done for every instance of TestSequence that is iterating through to ensure that the tests are not affecting each other.

Figure 6.7 shows how the interface for the tester is built up. The tester loads a TestSequence array with the LoadTest method. This starts the testing of all the test cases. When all the tests are tested it raises the ReturnTestResult event to return the

TestResultSequence array, by raising the ReturnTestResult event. The TestResultSequence array is very similar to TestSequence array but this also includes the result of the test.

```
public interface iTester
{
   void LoadTests(TestSequence[] tests,
                  Assembly testedAssembly);

   event ReturnTestResult returnTestResult;
}

public delegate void ReturnTestResult
     (TestResultSequence[] testResults);
```

Figure 6.7: Interface of the tester plugin.

6.2.5 Test Result Viewer

There is many ways to visualize the test results, so this task is of course in a separate plugin. Each plugin to view the test should work with every test and value generator. This part is very likely to use a GUI. The test result viewer plugin does not only show the test result, it also makes the decision of what test result is more useful than others. It tries to find unique results. The interface of the module is shown in figure 6.8. When the software engineer have found a test result that is useful the plugin raises the InterestingTestFound event. This sends the TestResultSequence to the result saver module.

```
public interface iResultViewer
{
   void LoadResults (TestResultSequence[] testResults);

   event InterestingTestFound interestingTestFound;
}

public delegate void InterestingTestFound(
          TestResultSequence interestingResult);
```

Figure 6.8: Interface of the test result viewer plugin.

6.2.6 ResultSaver

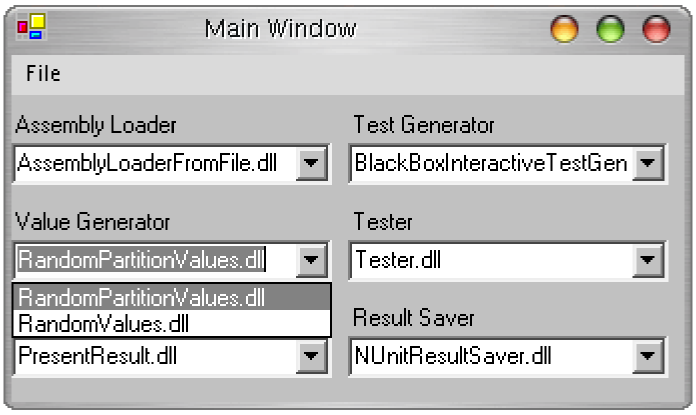
The result saver module is the module there the result is saved. How it is saved and where is up to the module. The module could ask for a path to save a XML file, or perhaps the result is saved to a NUnit DLL so the test could be run again in later with the NUnit tool. As shown in figure # the interface takes one TestResultSequence and save it.

```
public interface iResultSaver
{
   void SaveData ( TestResultSequence testResult);
}
```

Figure 6.9: Interface of the result saver plugin.

7 Implementation

A test implementation of the application was done to see if the theory works. The first thing of interests is the main application. The main application is responsible for the selection of the right modules as seen at picture 7.1. The modules exist within subdirectories relative to the main application executable file.

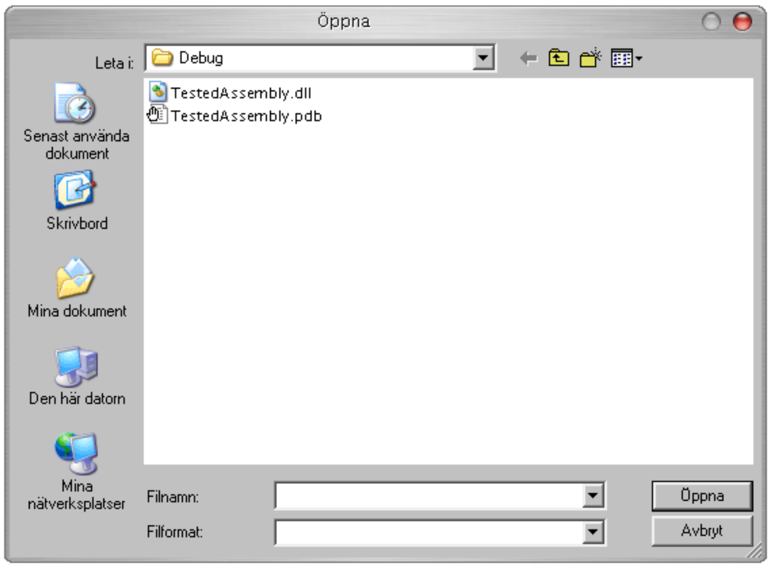


Assembly Loader: AssemblyLoaderFromFile.dll
Test Generator: BlackBoxInteractiveTestGen
Value Generator: RandomPartitionValues.dll
RandomPartitionValues.dll
RandomValues.dll
Tester: Tester.dll
Result Saver: NUnitResultSaver.dll
PresentResult.dll

Picture 7.1: Main Window

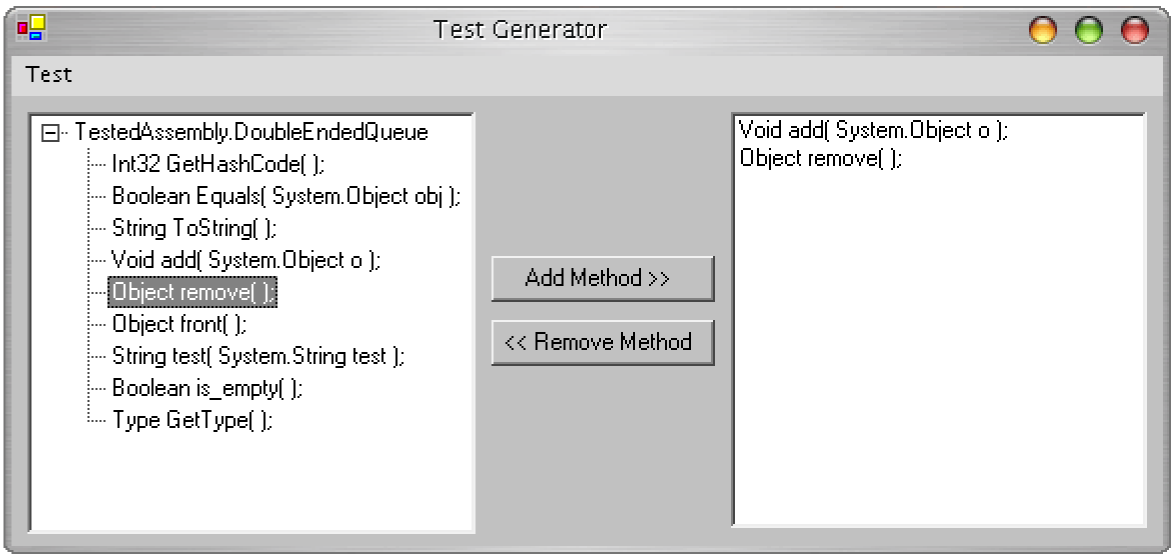
After the right modules is selected the user is suppose to use open an assembly to be tested from the file menu. This will start the open assembly plugin shown in picture 7.2[1]. The open assembly plugin does in this implementation show an open file dialog, so the user can choose any assembly that should be tested.

[1] A computer with the Swedish language installed was used to create the screen dumps. This is why some of the text will be Swedish.



Picture 7.2: Open assembly window

After the assembly is opened the test generator plugin starts. In this implementation a black box plugin is used. This plugin was implemented to be interactive, because it was useful to select what methods will be called during debugging of the application. As seen in picture 7.2 the user can specify what methods should be tested and in what order. In this example a double ended queue will be tested, so we chose to test the "add" method and the "remove" method, to see if it works to first add something to the queue and then remove it again.



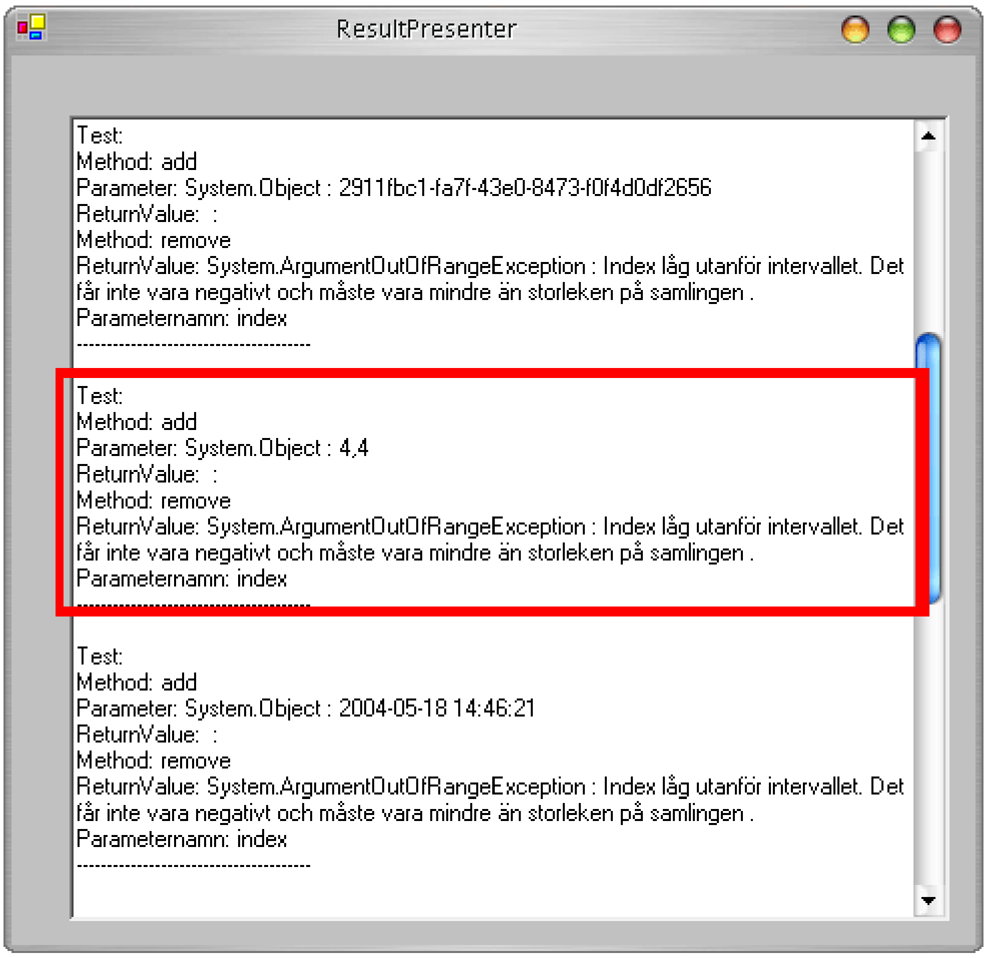
TestedAssembly.DoubleEndedQueue
Int32 GetHashCode( );
Boolean Equals(System.Object obj);
String ToString( );
Void add(System.Object o);
Object remove( );
Object front( );
String test(System.String test);
Boolean is_empty( );
Type GetType( );

Void add(System.Object o);
Object remove( );

Picture 7.3: An interactive black box plugin is used.

When the user have selected the methods that will be included in the test the user should select generate test on the test menu. This will start the generation of the tests; witch most likely will involve the value generator module to get input values to the methods. In this cast the value generator module will be used to get random objects for the add method. The implementation of the value generators used in this prototype is limited to generate values of the types: integer, double, string and object. This works to prove that this concept works, but to get good results better value generator modules must be developed. When the generation of the tests is done the testing itself is started. This is done in the tester module. In this implementation the tester module does not have any user interface.

When the test is done the result viewer module will start and show the GUI. This implementation will first show the method name, then the argument variables and last the return value. This is done in an ordinary text box and it does not sort the tests after how relevant whey is to keep it simple. As shown on picture 7.4 this test generated an ArgumentOutOfRangeException. This was not expected and we have uncovered a fault in the tested assembly.

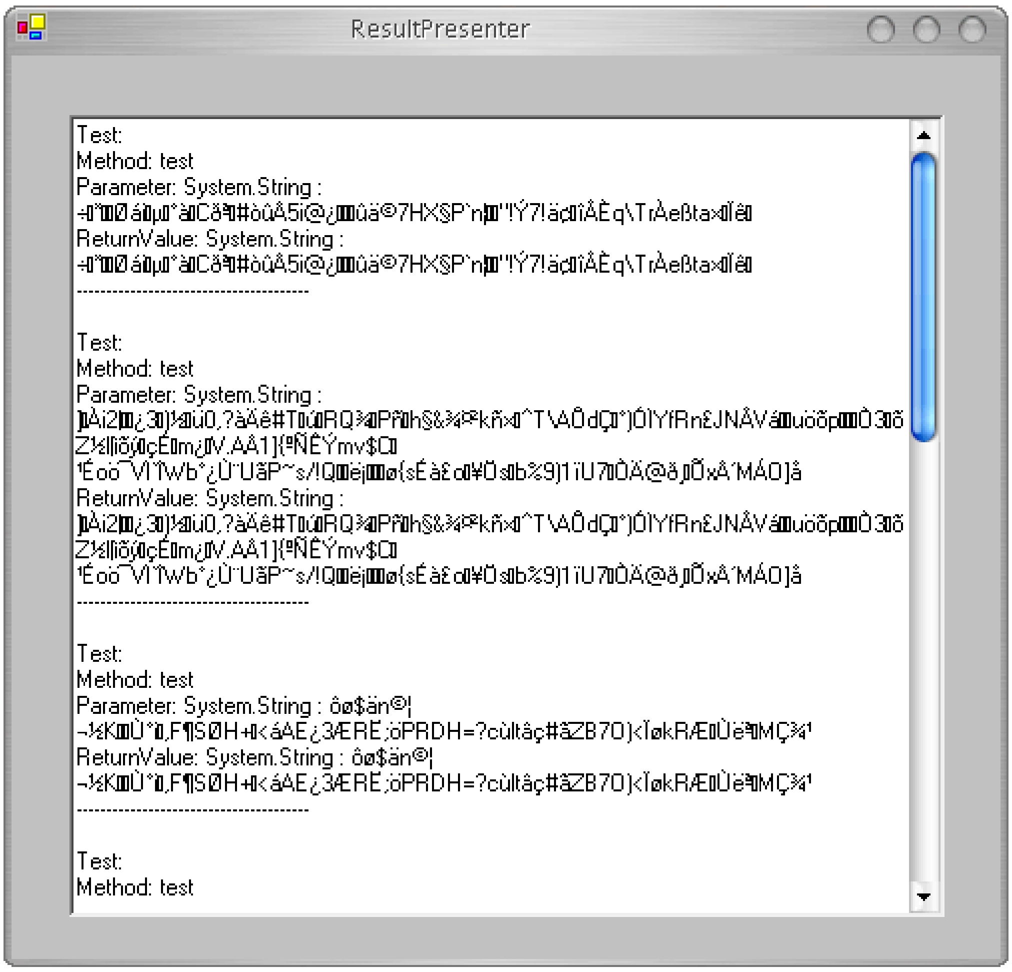


```
Test:
Method: add
Parameter: System.Object : 2911fbc1-fa7f-43e0-8473-f0f4d0df2656
ReturnValue:  :
Method: remove
ReturnValue: System.ArgumentOutOfRangeException : Index låg utanför intervallet. Det
får inte vara negativt och måste vara mindre än storleken på samlingen .
Parameternamn: index
--------------------------------------

Test:
Method: add
Parameter: System.Object : 4,4
ReturnValue:  :
Method: remove
ReturnValue: System.ArgumentOutOfRangeException : Index låg utanför intervallet. Det
får inte vara negativt och måste vara mindre än storleken på samlingen .
Parameternamn: index
--------------------------------------

Test:
Method: add
Parameter: System.Object : 2004-05-18 14:46:21
ReturnValue:  :
Method: remove
ReturnValue: System.ArgumentOutOfRangeException : Index låg utanför intervallet. Det
får inte vara negativt och måste vara mindre än storleken på samlingen .
Parameternamn: index
--------------------------------------
```

Picture 7.4: The result viewer has uncovered a fault.

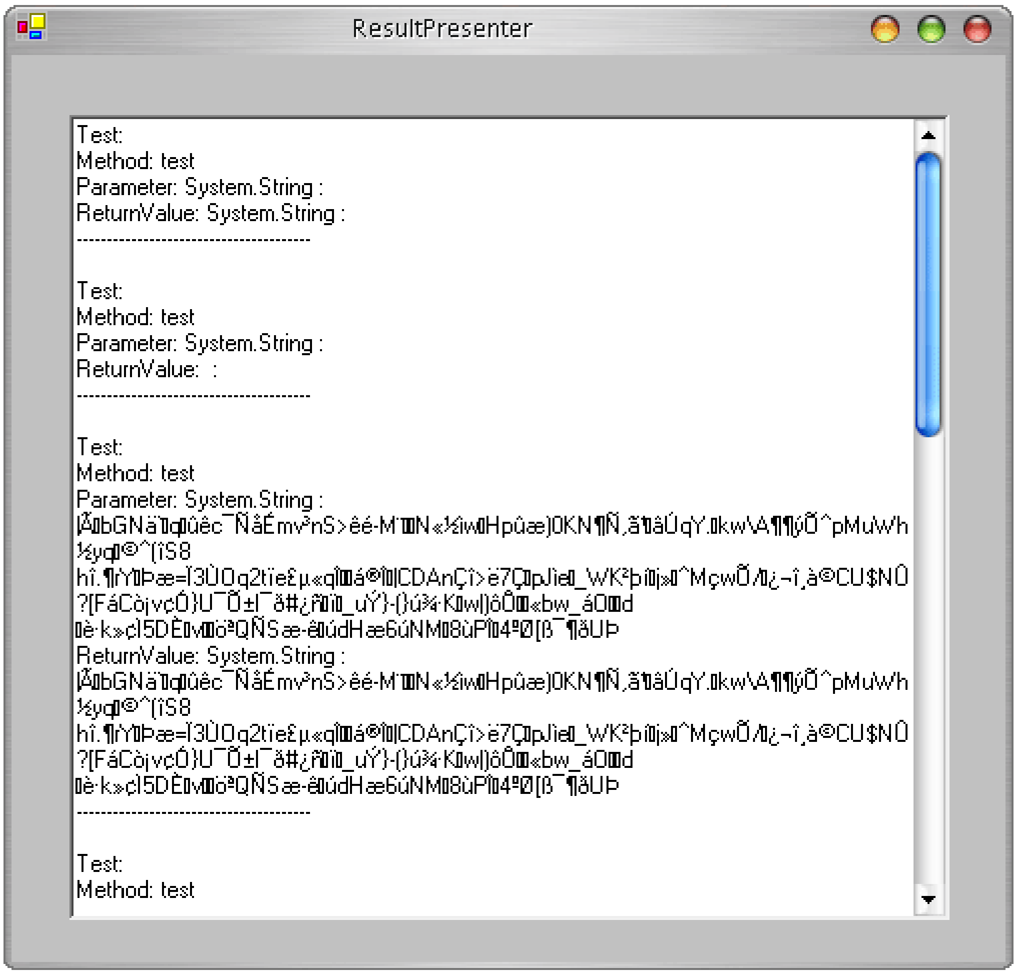
This was a very simple demonstration of how the application woks, but it will give an idea of how the application will look like. All plugin of the application are replaceable and the only thing needed to install a new plugin is to copy the module DLL file to the right directory.

To demonstrate that the modules are replaceable there will be another example that uses a random value generator and a value generator that combining random and BVA values. On picture 7.5 it is shown random strings. On picture 7.6 on the other hand the testing starts with an empty string, the next test uses null as string value and the third test uses a random string. By selecting different types of value generators in the menu the test result will be different.



```
Test:
Method: test
Parameter: System.String :
÷□°□□Ø á□µ□˚à□Cå¶#òûÅ5i@¿□□ûä©7HX§P`n□"!Ý7!äc□îÅÈq\TrÀeßtax□Ïê□
ReturnValue: System.String :
÷□°□□Ø á□µ□˚à□Cå¶#òûÅ5i@¿□□ûä©7HX§P`n□"!Ý7!äc□îÅÈq\TrÀeßtax□Ïê□
----------------------------------------


Test:
Method: test
Parameter: System.String :
]□Ài2□□¿3□)½□ü0,?àÄê#T□ú□RQ¾□Pñ□h§&¾ºkñx□^T\AÔdÇ□°)ÓÌYfRn£JNÅVá□□uöõp□□□Ò3□õ
Z½||iõý□çÉ□m¿□V.AÅ1]{ºÑÊÝmv$C□
¹Éoö¯VÌ˚ÎWb˚¿Ù¨UãP~s/!Q□□ëj□□□ø{sÉà£c□¥Ö s□b%9)1ïU7□ÒÄ@ð¸□ÕxÂ´MÁO]å
ReturnValue: System.String :
]□Ài2□□¿3□)½□ü0,?àÄê#T□ú□RQ¾□Pñ□h§&¾ºkñx□^T\AÔdÇ□°)ÓÌYfRn£JNÅVá□□uöõp□□□Ò3□õ
Z½||iõý□çÉ□m¿□V.AÅ1]{ºÑÊÝmv$C□
¹Éoö¯VÌ˚ÎWb˚¿Ù¨UãP~s/!Q□□ëj□□□ø{sÉà£c□¥Ö s□b%9)1ïU7□ÒÄ@ð¸□ÕxÂ´MÁO]å
----------------------------------------


Test:
Method: test
Parameter: System.String : ôø$än©¦
¬½K□□Ù°□,F¶SØH+□<áAE¿3ÆRË;öPRDH=?cùltâç#ãZB7O)<ÏøkRÆ□Ùë¶□MÇ¾¹
ReturnValue: System.String : ôø$än©¦
¬½K□□Ù°□,F¶SØH+□<áAE¿3ÆRË;öPRDH=?cùltâç#ãZB7O)<ÏøkRÆ□Ùë¶□MÇ¾¹
----------------------------------------


Test:
Method: test
```

Picture 7.5: Presenting random results.



```
Test:
Method: test
Parameter: System.String :
ReturnValue: System.String :
----------------------------------------

Test:
Method: test
Parameter: System.String :
ReturnValue:  :
----------------------------------------

Test:
Method: test
Parameter: System.String :
|Ã▯bGNä▯q▯ûêc¯ÑåÉmv²nS>êé-M▯▯N«½iw▯Hpûæ)0KN¶Ñ¸ã▯âÚqY.▯kw\A¶¶ýÕ^pMuWh
½yq▯©^[îS8
hî.¶rY▯Þæ=Ï3ÙOq2tïe£µ«qÎ▯á®Î▯|CDAnÇî>ë7Ç▯pJie▯_WK²þi▯j»▯^MçwÕ/▯¿¬î¸à©CU$NÒ
?[FáCòjv¢Ó}U¯Õ±I¯ð#¿ñ▯i▯_uÝ}-{}ú¾·K▯w|)ôÔ▯«bw_áO▯d
▯è·k»¢l5DÈ▯v▯ö²QÑSæ-ê▯údHæ6úNM▯8ùPÎ▯4º0[ß¯¶ðUÞ
ReturnValue: System.String :
|Ã▯bGNä▯q▯ûêc¯ÑåÉmv²nS>êé-M▯▯N«½iw▯Hpûæ)0KN¶Ñ¸ã▯âÚqY.▯kw\A¶¶ýÕ^pMuWh
½yq▯©^[îS8
hî.¶rY▯Þæ=Ï3ÙOq2tïe£µ«qÎ▯á®Î▯|CDAnÇî>ë7Ç▯pJie▯_WK²þi▯j»▯^MçwÕ/▯¿¬î¸à©CU$NÒ
?[FáCòjv¢Ó}U¯Õ±I¯ð#¿ñ▯i▯_uÝ}-{}ú¾·K▯w|)ôÔ▯«bw_áO▯d
▯è·k»¢l5DÈ▯v▯ö²QÑSæ-ê▯údHæ6úNM▯8ùPÎ▯4º0[ß¯¶ðUÞ
----------------------------------------

Test:
Method: test
```

Picture 7.6: Presenting a combination of BVA and random results.

8 Discussion

By using a modularized application to test software it will hopefully be easy to test new ideas. If a software engineer wants to test a new idea that involves testing, the only thing that person have to do is to write one or perhaps two modules instead of writing the whole testing application. By using the modularized approach it will be easy to exchange plugins with each other. If the application are good enough to make scientists use it, were would soon be a library of modules to chose from. The fact that it is open source gives it the possibility to grow.

Reflection was very easy to use and provided all information needed about the assembly to generate black box test. To make white box testing possible without knowing the source code, the IL code must be analysed. Third party software like RAIL can be used for that. RAIL does not only provide the information in the IL code, it also gives tools for analysing the algorithm without our application have to parse the IL code. This have however not been tested.

The main reason why it is hard to detect faults with the implemented solution is that it lacks the ability to rank test results. If the test result viewer was able to detect unique results, it would have been much easier to find the faults. In this implementation the software engineer have to scroll down a long list to find the faults. It was however not the key issue in this work to uncover faults. This work was about writing a system that could handle modules that supported the testing.

One problem that occurred was to generate values to be tested. The basic types like integer and double are not hard, but when generating complex types it will be more difficult. This work did not take this into consideration since it would make the value generator module a lot more complex.

One other thing that could lead to a problem is if a test generator needed a random value that does not exactly match any type. One example of this could be that it wants to generate an array of random length. If the generator asks for an integer it could end up with an array with millions of values. Sometimes it would be good to specify a range of valid values. This is not possible in this design.

If a test generator plugin would be created that are using a similar test approach as QuickCheck [9], this could be a problem. It could use the value generator to get input values, but there is no good way of loading specification for the application. On solution could be that the test generator plugin includes support for loading files that contains the specification. Another way would be to make a coding convention that the specification should be included in the assembly, but this would not work for COTS components. A third way is that the software engineer is allowed to enter specification each time the component is tested, in a text box or similar. This would not be good if the test is repeated several times, since the software engineer would get tired of entering the specification. One larger problem is how the correct response should be transported to the result viewer module. The present design does not support that, and the result viewer module is not written to take advantage of knowing the correct response.

9 Conclusion

We were able to build a plugin-based automated testing tool. It was however not fully automated, since it used the software engineer to make decisions. As an example was random testing implemented. It was not good enough to be used to uncover many faults, only very simple. The main reason for this is that the plugin that presents the tests is not yet able to rank the results after how unique they are.

We were able to change testing methodology in the application by changing what plugins was used. The plugins was DLL files, what was bound to the application at runtime using reflection. It was easy to change what plugin the application was using. The application is however limited in what types of methodology is used. There is no support to let the test generator predict valid output values form the method calls.

.NET got good support for generating and running tests. The reflection classes gave all the information needed to work with precompiled assemblies. Reflection does also make it possible to run any methods to see how it responds.

10 Future Work

One future work will be to write plugins that will improve the functionality. It would also be interesting to see if a solution close to QuickCheck can be used in this design. It would involve redesigning large parts of the application, but the result could be good. It would also be interesting to se how RAIL would improve the testing. RAIL has not been tested in practice in this work, but it could improve the selection of good input values to use white box testing.

References

[1] T. Y. Chen and Y. T. Yu. On the Relationship between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 20(12):977-980, 1994.

[2] Domain testing. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 444-448. Wiley, 2002.

[3] D. Hamlet. Random testing. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1095-1104. Wiley, 2002.

[4] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2002.

[5] Y. K. Malaiya. Anti-Random Testing: Getting the Most Out of Black-Box Testing. In *Proc. International Symposium On Software Engineering*, pages 86-95, October 1995.

[6] S. C. Ntafos. On Comparations of Random, Partition, and Proportional Partition Testing. *IEEE Transactions of Software Engineering*, 27(10):949-960, 2001.

[7] NUnit. *http://www.nunit.org/*, April 2004.

[8] Parasoft. *http://www.parasoft.com/*, May 2004.

[9] QuickCheck. http://www.cs.chalmers.se/~rjmh/QuickCheck/, May 2004.

[10] RAIL. *http://rail.dei.uc.pt/*, May 2004.

[11] Reflector for .NET. *http://www.aisto.com/roeder/dotnet/, May 2004.*

[12] S. C. Reid. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. In *4th International Software Metrics Symposium*, pages 64-73 IEEE Computer Press, 1997.

[13] W. ShenHui. Antirandom vs. pseudorandom testing. In *ICCD '98. Proceedings*, pages 221-223, October 1998.

[14] I. Sommerville. *Software Engineering*. John Wiley & Sons, Inc., 6th edition, 2001.

[15] R. Torkar. Empirical Studies of Software Black Box Testing Techniques: Evaluation and Comparation. Karlskrona, 2004.