Technical white paper

VuGen .NET Protocol Recording Best Practices and Troubleshooting

HP LoadRunner Best Practices Series

Table of contents

Introduction to .NET Protocol Recording ... 2
  Importance of .NET Protocol Recording and Filters ... 2
  Configuring .NET Protocol Recording in VuGen ... 2
  Preparing for .NET Protocol Recording Troubleshooting ... 5
Problem Patterns ... 6
  Pattern: Included but Not Recorded ... 6
  Pattern: Undefined Object, Type Not Included ... 7
  Pattern: Undefined Object, Type Included ... 8
  Pattern: Undefined Object, Type and Constructor Included ... 9
  Pattern: Irretrievable Object and Duplicate Calls in Thread ... 11
.NET Recording Tips and Techniques ... 12
  Use Visual Studio to Edit your Script ... 12
  Manually Edit the Filter File to Record non-Public Types and Members ... 13
  Use .NET Reflection to Invoke Non-Public Constructors ... 13
  Use .NET Reflection to Invoke Non-Public Members ... 13
  Use .NET Reflection to Work With Non-Public Types ... 14
  Turn on Stack Tracing ... 14
  Discover How an Object is Constructed ... 15
  Solve undefined object error ... 15
  Address Business Layer ... 16
Methodologies for Recording .NET Applications ... 17
  Top-Down Approach ... 17
  Bottom-Up Approach ... 18

Welcome to this Document

Welcome to VuGen .NET Protocol Recording Best Practices and Troubleshooting. This document provides concepts, guidelines, and practical examples for recording and troubleshooting .NET application method calls

using the HP LoadRunner .NET protocol in LoadRunner's Virtual User Generator application (VuGen)

This document is intended for Performance Test Automation Engineers who are developing scenarios for load testing of .NET applications, and who need to record .NET calls made from their Application Under Test (AUT). It is applicable to HP LoadRunner 11.50 and later.

Introduction to .NET Protocol Recording

Importance of .NET Protocol Recording and Filters

The Microsoft .NET Framework provides a foundation for developers to build various types of applications such as ASP.NET, Windows Forms, Web Services, distributed applications, and applications that combine several of these models.

HP LoadRunner's VuGen supports .NET as an application level protocol. VuGen allows you to create Virtual User (Vuser) scripts that emulate users of Microsoft .NET client applications created with the .NET Framework. VuGen records all of the client actions through methods and classes, and creates Vuser scripts in C# or VB.NET.

Recording filters indicate which assemblies, interfaces, namespaces, classes, or methods will be included or excluded during the recording and script generation phases.

By default, the VuGen environment is configured for .NET Remoting, ADO.NET, Enterprise Services, and WCF (Windows Communication Foundation) applications, and provides built-in system filters with the relevant interfaces for each of these technologies.

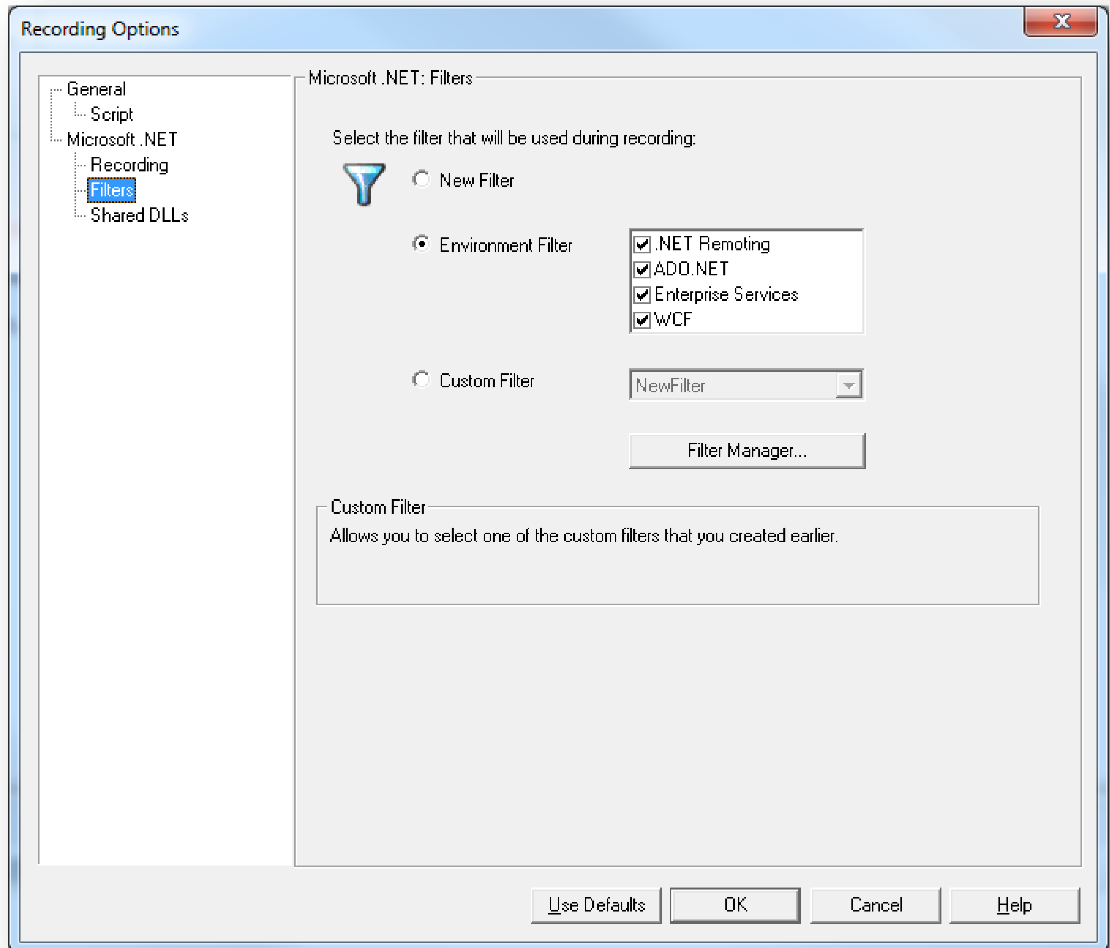
VuGen also allows you to design custom filters. Custom filters provide several benefits:

- Remoting. When working with .NET Remoting, you can include certain classes that allow you to record the arguments passed to the remote method.
- Missing Objects. If your recorded script did not record a specific object within your application, you can use a filter to include the missing interface, class or method.
- Debugging. If you receive an error, but you are unsure of its origin, you can use filters to exclude methods, classes, or interfaces in order to pin-point the problematic operation.
- Maintainability. You can record scripts at a higher level to make the script easier to maintain and to correlate.

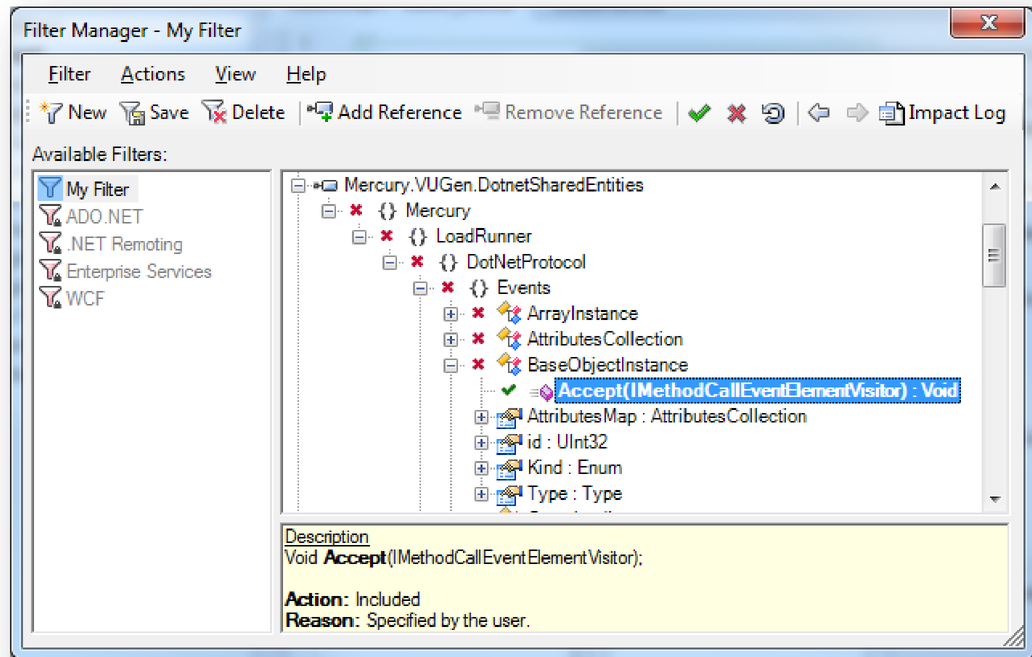
For more information, read the "*.NET Protocol*" section of the HP LoadRunner User Guide.

Configuring .NET Protocol Recording in VuGen

.NET Protocol recording is configured in the Recording Options dialog, accessible from the *Record > Recording Options…* menu. Select *Filters* under *Microsoft .NET* from the tree on the left:

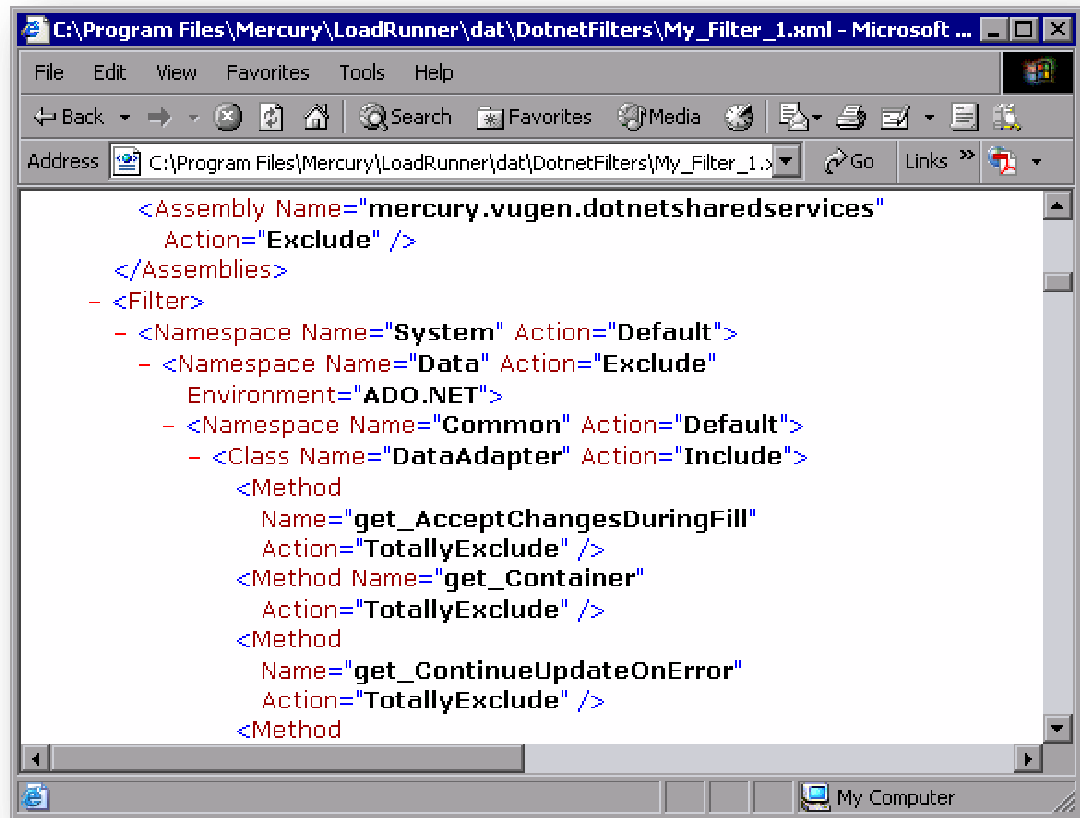


The '*Filter Manager…*' button opens a window that lets you create and manage custom filters. It displays the assemblies, namespaces, classes, methods, and properties in a color-coded tree hierarchy:



The Filter Manager's tree hierarchy only displays public classes and methods. It does not show non-public classes or delegates, but you can add classes or methods that are not public by manually entering them in the filter's definition file.

The filter definition files, *<filter_name>.xml* reside in the *dat\DotnetFilters* folder of your installation. The available Action properties for each element are: *Include*, *Exclude*, or *Totally Exclude*. A typical filter definition file might look something like this:



VuGen saves a backup copy of the filter as it was configured during the recording, *RecordingFilterFile.xml*, in the script's data folder. This is useful if you made changes to the filter since your last recording and you need to reconstruct the environment.

For a full description, see the "*Filter Manager [.NET Protocol]*" section in the HP LoadRunner User Guide.

Preparing for .NET Protocol Recording Troubleshooting

There are some tools that you should have available to make troubleshooting easier. This chapter describes some of these tools.

.NET Reflector

To be most effective, you should have some knowledge of the architecture of the .NET application you are trying to record. The best way to gain insight into the architecture is to use a.NET reflector tool, which will decompile your application's modules, and let you browse its assemblies.

You may encounter scenarios, such as the ones described in this guide, where for example, you need to find where an object is constructed, where it is stored after the construction, and how it is retrieved for later use. A reflector is highly recommended for finding out this information.

Examples of reflectors include:

- ILSpy – http://ilspy.net
- Red gate reflector - http://www.reflector.net/

XML Editor

As the filter manager (see the "*Configuring .NET Protocol Recording in VuGen*" section) currently only shows public types and members, you will need a way of including non-public items in the filter. Although you can use a simple text editor, such as Notepad, to edit the XML, the easiest way is to use an XML editor. Examples of XML editors include:

- Microsoft Visual Studio – http://www.microsoft.com/visualstudio/eng/visual-studio-update
- NotePad++ – http://notepad-plus-plus.org/
- XML Spy – http://www.altova.com/xml-editor/

Problem Patterns

When you define your filter and record your application, you may find that the results are not what you expected. This section describes some common patterns of the recorder's behavior, and explains how to work around them.

Each of the patterns will follow the following format:

- AUT Code – this is the actual code of the application being recorded. The examples here show the minimum code required to exhibit the problematic behavior, so there is no error handling etc.
- Initial Filter – this is how filter was initially configured.
- Generated Code – this is the VuGen code that is generated by the recorder, and which is typically used to drive the desired load testing scenario.
- Explanation – an explanation of the observed behavior.
- Recommended Fix – if necessary, a change to the filter will be recommended in order to achieve the desired results.
- Generated Code After Fix – this is the VuGen code that is generated by the recorder after the recommended fix has been applied to the filter.

Pattern: Included but Not Recorded

In this pattern, a method, MethodB, was included in the filter, but does not appear in the recording.

AUT Code

```
public class Program {
  public static void MethodA() {
    Console.WriteLine("MethodA");
    MethodB();
  }

  public static void MethodB() {
    Console.WriteLine("MethodB");
  }

  static void Main(string[] args) {
    MethodA();
  }
}
```

Initial Filter

```
<Filter>
  <Namespace Name="IncludedButNotRecorded" Action="Default">
    <Class Name="Program" Action="Default">
      <Method Name="MethodA()" Action="Include"/>
      <Method Name="MethodB()" Action="Include"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code

```
Public virtual int vuser_init() {
  lr.log("Event 1: Program.MethodA();");
  Program.MethodA();
```

```
  return 0;
}
```

Explanation

The generated code does not contain a call to MethodB. However, this is correct behavior. MethodB is called by the MethodA, so it does not need to be recorded.

Pattern: Undefined Object, Type Not Included

In this pattern, the generated code attempts to pass an object of an undefined type to a method.

AUT Code

```
public class NotIncludedType {
  public NotIncludedType(int i) {
  }
}

public class Program {
  public static void CallWithUndefinedObject(NotIncludedType obj) {
    Console.WriteLine(obj);
  }

  static void Main(string[] args) {
    CallWithUndefinedObject(new NotIncludedType(1));
  }
}
```

Initial Filter

```
<Filter>
  <Namespace Name="UndefinedObject" Action="Default">
    <Class Name="Program" Action="Default">
     <Method Name="CallWithUndefinedObject(UndefinedObject.NotIncludedType)"
Action="Include"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code

```
public virtual int vuser_init() {
  #warning:  Code Generation Error
  // Found an undefined object of type UndefinedObject.NotIncludedType. Assigning it the
name obj_1.
  // Suggested solution: adding both this type, in assembly UndefinedObject,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null, to the filter
  // and/or any other types that return instances of this one.
  // Note:  This script will not compile as is.
  lr.log("Event 1: Program.CallWithUndefinedObject(obj_1);");
  Program.CallWithUndefinedObject(obj_1);

  Return 0;
}
```

Explanation

Because we didn't record the constructor of the class `NotIncludedType`, VuGen doesn't know how to create an object of that type. To fix it, we need to include the constructor of `NotIncludedType` to the filter.

Recommended Fix

```
<Filter>
  <Namespace Name="UndefinedObject" Action="Default">
    <Class Name="Program" Action="Default">
          <Method Name="CallWithUndefinedObject(UndefinedObject.NotIncludedType)"
Action="Include"/>
```

```
    </Class>
    <Class Name="NotIncludedType" Action="Default">
      <Method Name=".ctor(System.Int32)" Action="Include"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code After Fix

```
public virtual int vuser_init() {

  lr.log("Event 1: new NotIncludedType(1);");
  NotIncludedType_1 = new NotIncludedType(1);

  lr.log("Event 2: Program.CallWithUndefinedObject(NotIncludedType_1);");
  Program.CallWithUndefinedObject(NotIncludedType_1);

  return 0;
}
```

Pattern: Undefined Object, Type Included

In this pattern, the generated code attempts to pass an object of an undefined type to a method, even though the type was marked as included in the filter.

AUT Code

```
public class IncludedType {
  internal IncludedType(int i) {
  }
}

public class Program
{
  public static void CallWithUndefinedObject(IncludedType obj) {
    Console.WriteLine(obj);
  }

  static void Main(string[] args) {
    CallWithUndefinedObject(new IncludedType(1));
  }
}
```

Initial Filter

```
<Filter>
  <Namespace Name="TypeIncludedUndefinedObject" Action="Default">
    <Class Name="IncludedType" Action="Include"/>
    <Class Name="Program" Action="Default">
      <Method Name="CallWithUndefinedObject(TypeIncludedUndefinedObject.IncludedType)"
Action="Include"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code

```
public virtual int vuser_init() {
  #warning:  Code Generation Error
  // Found an undefined object of type TypeIncludedUndefinedObject.IncludedType. Assigning
it the name obj_1.
  // Suggested solution: adding both this type, in assembly TypeIncludedUndefinedObject,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null, to the filter
  // and/or any other types that return instances of this one.
  // Note:  This script will not compile as is.
  lr.log("Event 1: Program.CallWithUndefinedObject(obj_1);");
  Program.CallWithUndefinedObject(obj_1);
```

```
  return 0;
}
```

Explanation

Even though the class `IncludedType` is marked as included in the filter, the constructor call is not recorded. This is because the constructor is marked as `internal`. VuGen will not record non-public types and members unless the type or member is explicitly added to the filter.

Recommended Fix

```
<Filter>
  <Namespace Name="TypeIncludedUndefinedObject" Action="Default">
    <Class Name="IncludedType" Action="Include">
      <Method Name=".ctor(System.Int32)" Action="Include"/>
    </Class>
    <Class Name="Program" Action="Default">
      <Method Name="CallWithUndefinedObject(TypeIncludedUndefinedObject.IncludedType)"
Action="Include"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code After Fix

```
public virtual int vuser_init() {
  lr.log("Event 1: new IncludedType(1);");
  IncludedType_1 = new IncludedType(1);

  lr.log("Event 2: Program.CallWithUndefinedObject(IncludedType_1);");
  Program.CallWithUndefinedObject(IncludedType_1);

  return 0;
}
```

However, note that this will still not compile, because the generated code is attempting to initialize an object with an internal constructor. See "*Use .NET Reflection to Invoke Non-Public Constructors*" for an explanation of how to work around this problem.

Pattern: Undefined Object, Type and Constructor Included

In this pattern, the filter includes the constructor explicitly, but there is an undefined object in the generated code.

AUT Code

```
public class IncludedType {
  internal IncludedType(int i) {
  }
}

public class Program {
  public static void CallWithUndefinedObject(IncludedType obj) {
    Console.WriteLine(obj);
  }

  public static void Initialize() {
    Instance = new IncludedType(1);
  }

  public static IncludedType Instance;

  static void Main(string[] args) {
    Initialize();
    CallWithUndefinedObject(Instance);
  }
}
```

Initial Filter

```
<Filter>
```

```
  <Namespace Action="Default" Name="TypeConstructorIncludedUndefinedObject">
    <Class Action="Include" Name="IncludedType">
      <Method Action="Include" Name=".ctor(System.Int32)"/>
    </Class>
    <Class Action="Default" Name="Program">
      <Method Action="Include" Name="Initialize()"/>
      <Method
Name="CallWithUndefinedObject(TypeConstructorIncludedUndefinedObject.IncludedType)"
Action="Include"/>
    </Class>
    <Class Action="Include" Name="IncludedType"/>
  </Namespace>
</Filter>
```

Generated Code

```
public virtual int vuser_init() {
  lr.log("Event 1: Program.Initialize();");
  Program.Initialize();

  #warning:  Code Generation Error
  // Found an undefined object of type
TypeConstructorIncludedUndefinedObject.IncludedType. Assigning it the name obj_1.
  // Suggested solution: adding both this type, in assembly
TypeConstructorIncludedUndefinedObject, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null, to the filter
  // and/or any other types that return instances of this one.
  // Note:  This script will not compile as is.
  lr.log("Event 2: Program.CallWithUndefinedObject(obj_1);");
  Program.CallWithUndefinedObject(obj_1);

  return 0;
}
```

Explanation

The filter explicitly includes the constructor, but there is still an undefined object. This is because the method `Initialize` in the AUT code performs the construction and initialization of the Instance member. The fully constructed and initialized Instance object is then passed into the `CallWithUndefinedObject` function. But the generated code is unaware of this, and attempts to pass an object it doesn't know about to the `CallWithUndefinedObject` function. There are two ways around this. Either modify the generated code to pass the Program's `Instance` member, or exclude the `Program.Initialize` method from the filter so that it isn't generated at all.

Recommended Fix

Either modify the code as follows:

```
public virtual int vuser_init() {
  lr.log("Event 1: Program.Initialize();");
  Program.Initialize();

  lr.log("Event 2: Program.CallWithUndefinedObject(Program.Instance);");
  Program.CallWithUndefinedObject(Program.Instance);

  return 0;
}
```

Or, change the filter as follows:

```
<Filter>
    <Namespace Action="Default" Name="TypeConstructorIncludedUndefinedObject">
    <Class Action="Include" Name="IncludedType">
      <Method Action="Include" Name=".ctor(System.Int32)"/>
    </Class>
    <Class Action="Default" Name="Program">
      <Method Action="Include" Name="Initialize()"/>
```

```
      <Method
Name="CallWithUndefinedObject(TypeConstructorIncludedUndefinedObject.IncludedType)"
Action="Include"/>
    </Class>
    <Class Action="Include" Name="IncludedType"/>
  </Namespace>
</Filter>
```

Generated Code After Fix

If the filter is changed, the generated code becomes:

```
public virtual int vuser_init() {
  lr.log("Event 1: new IncludedType(1) ");
  IncludedType_1 - new IncludedType(1);

  lr.log("Event 2: Program.CallWithUndefinedObject(IncludedType_1);");
  Program.CallWithUndefinedObject(IncludedType_1);

  return 0;
}
```

This will not compile, because `IncludedType(System.Int32)` is not a public constructor. See "*Use .NET Reflection to Invoke Non-Public Constructors*" for an explanation of how to work around this problem.

Note

The first solution of modifying the generated code to pass the initialized object can be challenging, since it may be difficult to retrieve the object that was created during the initialization procedure. However, it is ultimately possible, and the proof of this is the fact that the AUT itself must be able to find it in order to pass it. The best place to start is with the developers of the AUT, who may be able to help. If that is not possible, and you don't have access to the AUT's source code, you can use a .NET reflector to inspect the AUT's logic, and determine how to retrieve the instance.

Pattern: Irretrievable Object and Duplicate Calls in Thread

In this pattern, a worker thread is responsible for a work item, but the generated code attempts to do the work in the main thread.

AUT Code

```
public class NotIncludedType {
  public class Program
  {
    public static void Worker(object state) {
      WorkItem item = state as WorkItem;
      Console.WriteLine(item.Inc());
  }

  public class WorkItem {
    int seed = 0;
    public WorkItem(int seed) {
      this.seed = seed;
    }

    public int Inc() {
      return ++seed;
    }
  }

  public static void DoWork() {
    ThreadPool.QueueUserWorkItem(new WaitCallback(Worker), new WorkItem(11));
  }

  public static void Main(string[] args) {
    DoWork();
```

```
    Thread.Sleep(1000);
  }
}
```

Initial Filter

```
<Filter>
  <Namespace Action="Default" Name="IrretrievableObject">
    <Class Action="Default" Name="Program">
      <Method Action="Include" Name="DoWork()"/>
    <Class Action="Include" Name="WorkItem"/>
    </Class>
  </Namespace>
</Filter>
```

Generated Code

```
public virtual int vuser_init() {
  lr.log("Event 1: Program.DoWork();");
  Program.DoWork();

  #warning:  Code Generation Error
  // Found an undefined object of type IrretrievableObject.Program+WorkItem. Assigning it
the name Program_WorkItem_1.
  // Suggested solution: adding both this type, in assembly IrretrievableObject,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null, to the filter
  // and/or any other types that return instances of this one.
  // Note:  This script will not compile as is.
  lr.log("Event 2: Program_WorkItem_1.Inc();");
  Int32RetVal = Program_WorkItem_1.Inc();

  return 0;
}
```

Explanation

There is an obvious error in the generated code. The `Program.WorkItem` is undefined, and is in fact not actually retrievable, since objects of this type are not stored anywhere, and are transferred to new threads directly as state objects. However, there is a second problem which is more serious. If we were able to work around the first problem, the generated code would then behave differently to the AUT, because it would call `Program.DoWork`, and then call `Program.WorkItem.Inc`. This would cause the code to call `Inc` twice – once as a result of the `Worker` function, and then again as a direct call.

The first problem cannot be solved, as there is no way to retrieve the undefined object. The fix for the second problem is to exclude either `Program.DoWork` or `Program.WorkItem.Inc` from the filter.

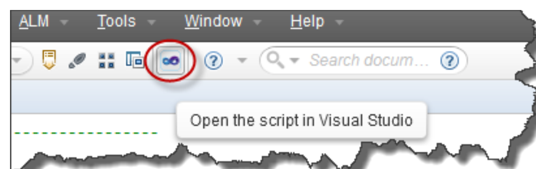
Problems like this are difficult to distinguish from patterns such as "*Pattern: Undefined Object, Type and Constructor Included*" without diving down into the AUT code.

.NET Recording Tips and Techniques

This section details a number of tips and techniques that you can adopt to help you record and generate better .NET protocol scripts.

Use Visual Studio to Edit your Script

Although VuGen provides support for C# projects, you can also use Visual Studio to edit your script. To open the script in Visual Studio from VuGen, select *Design > Open Script* in the Visual Studio menu, or click the Visual Studio button on the VuGen toolbar:



Manually Edit the Filter File to Record non-Public Types and Members

The .NET protocol can record non-public type and member calls. But in order to do that, you need to modify the filter to include non-public types and members explicitly. The Filter Manager doesn't show non-public types and members, so you need to open filter file and modify it manually. For example to include the **BeginDbTransaction** method, you can add the highlighted line as follows:

```
<Filter>
  <Namespace Action="Default" Name="System">
    <Namespace Action="Default" Name="Data">
      <Namespace Action="Default" Name="SqlClient">
        <Class Action="Include" Name="SqlConnection">
          <Method Action="Include" Name="BeginDbTransaction(System.Data.IsolationLevel)"/>
    </Class>
…
```

Use .NET Reflection to Invoke Non-Public Constructors

If we were to record a call to a non-public constructor, the generated script would be as follows:

```
public virtual int vuser_init() {
  …
  IncludedType_1 = new IncludedType(1);
  …
  return 0;
}
```

The highlighted line will not compile because the constructor is not public. You can use .NET reflection to solve this as follows:

```
public virtual int vuser_init() {
  …
  IncludedType_1 =  (IncludedType) Activator.CreateInstance(typeof(IncludedType),
BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public, null, new object[] {
1 }, null);
  …
  return 0;
}
```

For more information about the `Activator.CreateInstance` method, see the Microsoft Developer Network, at http://msdn.microsoft.com/en-us/library/2ca9yys8.aspx

Use .NET Reflection to Invoke Non-Public Members

If we were to record a call to a non-public method, the generated script would be as follows:

```
public virtual int vuser_init() {
  …
  sqlConnection_1.BeginDbTransaction(xxx);
  …
  return 0;
}
```

The highlighted line will not compile because the `BeginDbTransaction` method is not public. You can use .NET reflection to solve this without changing the generated code, by adding an extension class, which includes an extension method with the same name as the method that doesn't compile. Follow the following steps:

1) Open the script in Visual Studio

2) Create a new C# file called *Extension.cs*

3) Copy the following code to *Extension.cs*:

```
namespace Script
{
    using System.Reflection;
    using System.Data.Common;
    using System.Data;
    public static class Extension
```

```
    {
        public static DbTransaction BeginDbTransaction(this DbConnection target,
IsolationLevel isolationLevel)
        {
            MethodInfo method = target.GetType().GetMethod("BeginDbTransaction",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
            return (DbTransaction)method.Invoke(target, new object[] { isolationLevel });
        }
    }
}
```

4) Add *Extension.cs* to the project in Visual Studio

5) Add a reference to the `System` and `System.Data` assemblies to the project

6) The previously problematic script should now compile successfully when you build the project.

Use .NET Reflection to Work With Non-Public Types

If we were to record an object which has a non-public type, the generated script would fail to compile, because it is not possible to reference the non-public type. For example if you recorded a type called `SqlConnectionString`, the generated script might look like this:

```
Public virtual int vuser_init() {
  …
  sqlConnectionString SqlConnectionString_1;
  …
  return 0;
}
```

The script will not compile, because the `SqlConnectionString` type is not public.

To fix this issue, we can add a proxy type as follows:

```
// The name of the class must be identical to the name of the type we want to proxy
Public class SqlConnectionString {
  Object obj = null;
  Public SqlConnectionString(string connectionString) {
    // The actual object is created here.
    // We need the fully qualified name to create it by reflection
    // (see http://msdn.microsoft.com/en-us/library/system.type.aspx)
    Type type = Type.GetType("System.Data.SqlClient.SqlConnectionString, System.Data,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
    obj = Activator.CreateInstance(type, BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public, null, new object[] { connectionString }, null);
  }

Public string Expand() {
  // Delegate to the actual object's Expand method.
  // (see http://msdn.microsoft.com/en-us/library/system.reflection.methodinfo.aspx)
  MethodInfo mi = obj.GetType().GetMethod("Expand",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
  Return mi.Invoke(obj, new object[] { }) as string;
  }
}
```

The `SqlConnectionString` type that appears in the script will be compiled as our proxy type, and all calls to the instance of our `SqlConnectionString` will be redirected to the actual `SqlConnectionString` instance.

The script should now compile without having to change the generated code.

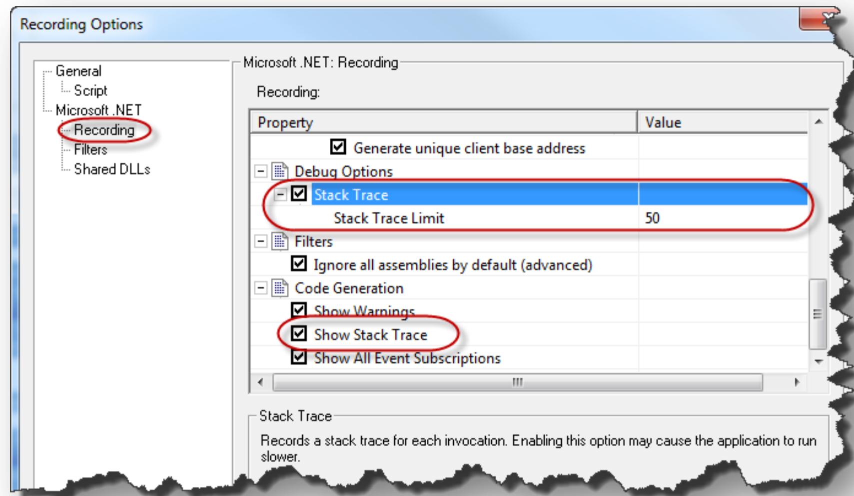
Turn on Stack Tracing

Sometimes we need stack trace information in order to get some insight on the AUT. To turn on stack tracing in VuGen, open the *Record > Recording Options…* dialog, and configure the Microsoft .NET Recording options as follows:

- Check the *Stack Trace* checkbox under '*Debug Options*'

- Ensure the *Stack Trace Limit* is set to a large enough value, for example 50.
- Check the '*Show Stack Trace*' checkbox under '*Code Generation*'.

The dialog should look something like this:



After recording, it will generate a call stack that should look like this:

```
// Stack Trace:
//  at System.Runtime...Connect(Type classToProxy, String url)
//  at System.Runtime...ConnectIfNecessary(IConstructionCallMessage ctorMsg)
//  at System.Runtime...CreateInstance(RuntimeType serverType)
//  at System.Runtime...CreateInstanceInternal(RuntimeType serverType)
//  at System.Runtime...IsCurrentContextOK(RuntimeType...bNewObj)
//  at Core.Login(String userName, String password)
//  at UI.MainForm.buttonLogin_Click()
//  at UI.Program.Main()
```

Discover How an Object is Constructed

If you need to understand how an object is constructed, you can use the following process to generate a script which includes the call stack for the object's constructor:

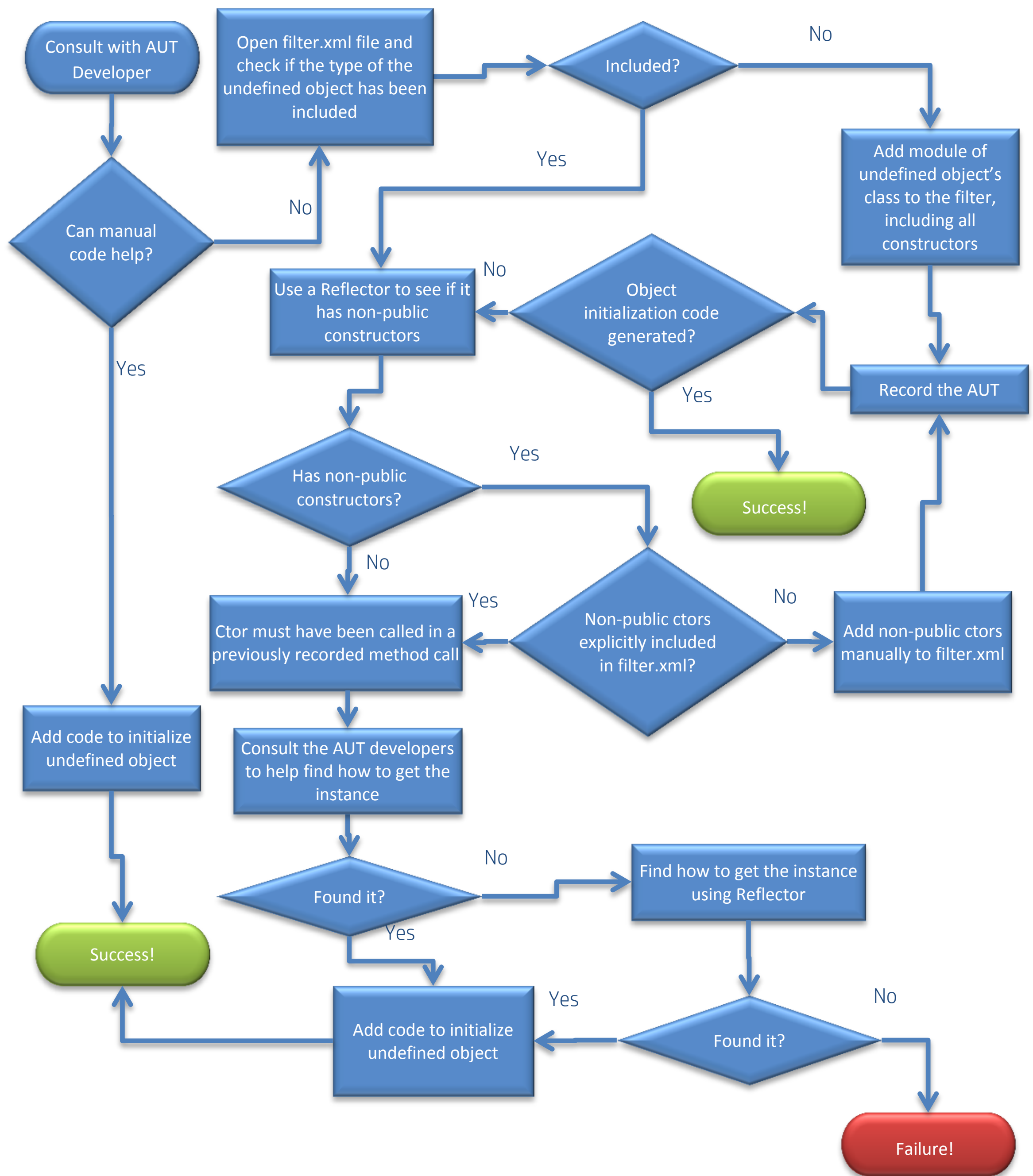
1. Create a new .NET script in VuGen
2. Create a new filter
3. Add the object's type and public constructors to the filter
4. Explicitly add non-public constructors to the filter's XML file
5. Configure stack tracing in the *Recording > Recording Options...* dialog, as described in the "*Turn on Stack Tracing*" section.
6. Record and generate the script

The call stack in the script will show how the constructor is called.

Solve undefined object error

If you have an undefined object that you need to initialize in your script, the quickest route is to talk to the developers of the AUT who might be able to suggest how you can manually add code to the script in order to initialize the object.

If that is not possible, the following flowchart will help you to modify the filter so you can record the script initialize undefined objects to your script (note: the word 'ctor' is used as an abbreviation for 'constructor').



For help with solving the failed scenario, see the "*Methodologies for Recording .NET Applications*" section which explains how to record the AUT in detail.

Address Business Layer

So far, we've discussed specific problems and how to resolve them. But if you include the correct methods in the filter in the first place, you can avoid these problems.

Applications are typically build with a set of methods that are invoked from the user interface in order to complete a specific business scenario. This set of methods is known as the Business Layer. You should consult the developers of your AUT to get information about its Business Layer.

If you enable stack tracing in VuGen, you can investigate it yourself. Refer to the "*Turn on Stack Tracing*" section for an explanation of how to enable stack tracing.

After recording with stack tracing enabled, a stack trace like the following will be generated with the code:

```
// Stack Trace:
// at System.Runtime...Connect(Type classToProxy, String url)
// at System.Runtime...ConnectIfNecessary(IConstructionCallMessage ctorMsg)
// at System.Runtime...CreateInstance(RuntimeType serverType)
// at System.Runtime...CreateInstanceInternal(RuntimeType serverType)
// at System.Runtime...IsCurrentContextOK(RuntimeType svrType...Boolean bNewObj)
// at Core.Login(String userName, String password)
// at UI.MainForm.buttonLogin_Click()
// at UI.Program.Main()
```

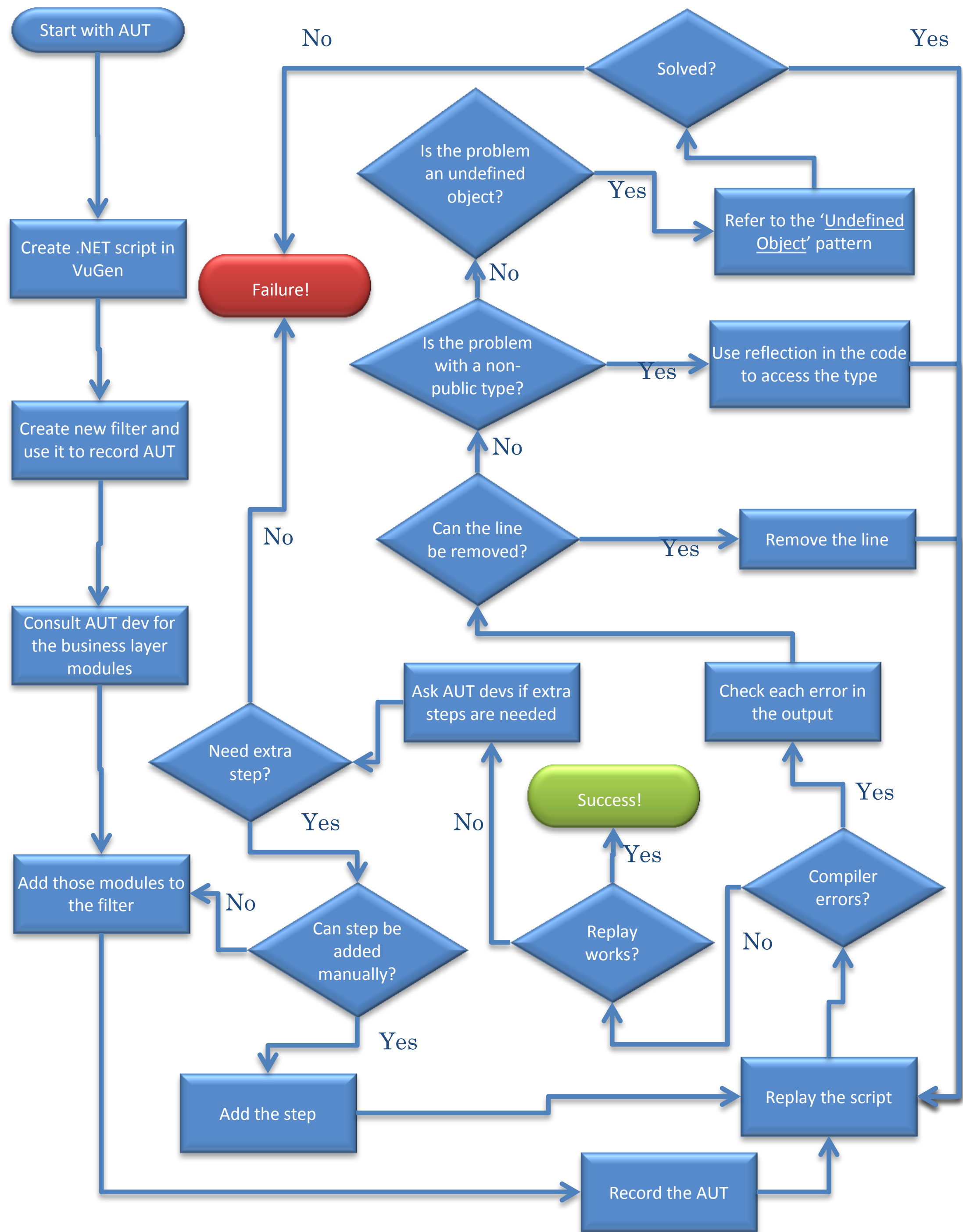
From the stack trace, you can see there is a method called `Core.Login(String userName, String password)`, which is probably the business layer method for user login. We make this assumption based on both its name, and the fact that it lies in between the UI layer and the system layer. Including this method in the filter may help generate a simple and effective script for the user login action.

Methodologies for Recording .NET Applications

There are two methodologies for recording an application; top-down and bottom-up. Top-down refers to starting by recording the high level business logic, while bottom-up refers to starting by recording low-level transport methods. The top-down approach is recommended when you are familiar with the AUT's architecture and implementation, while the bottom-up approach is recommended if you have little or no knowledge of the AUT's structure.

Top-Down Approach

The top-down approach starts the script creation process by recording high-level business logic method calls. If you are able to record enough high-level methods to be able to replay the script correctly, you can use the script in your load testing scenarios as-is. If not, you might have to record at a lower level in order to capture the correct method calls. The following flowchart describes the process in detail:



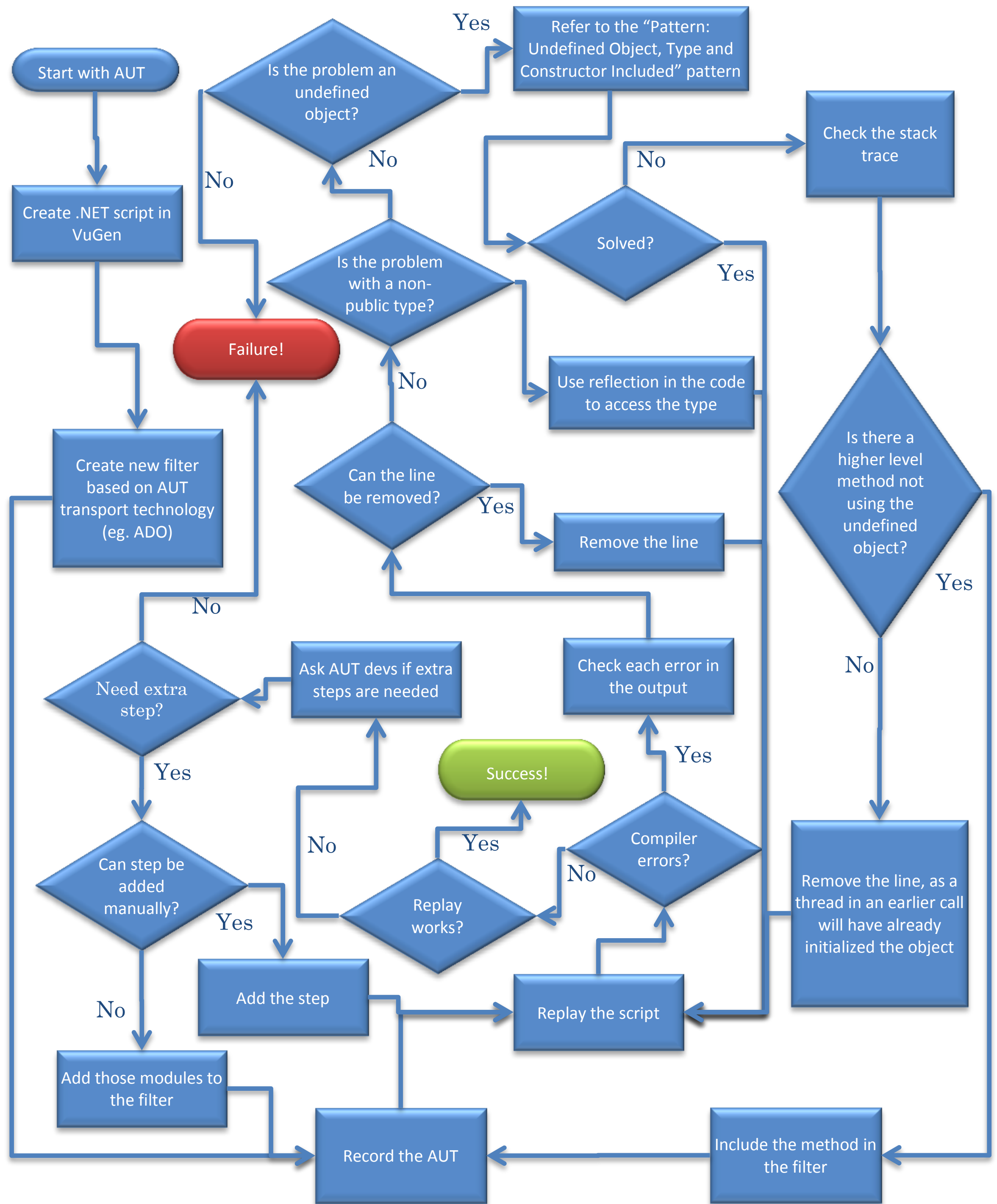
If you follow the flowchart and end up at 'failure', your next course of action should be to consult the AUT developers for more help, or to refer to the "*Bottom-Up Approach*" section.

The top-down record methodology works well for applications that have clear boundaries between its layers. Ideally, the test scenarios should be taken into consideration during the AUT design stage. The AUT should be able to be unit-tested, and you should be able perform all business operation with API calls that do not require any UI.

Bottom-Up Approach

The bottom-up approach starts the script creation process by recording low-level transport methods by using LoadRunner's built-in environment Filter. If the generated script can be replayed correctly, you can use the script in your load testing scenarios as-is. If not, you might need to solve the replay errors by including more classes and assemblies. You can also try to record methods at a higher level. For example, let's assume our AUT has a method called `methodA` which calls another

method, **methodB**. When you record the script, **methodB** was recorded as expected, but when the script is replayed, you get an error that **methodB** has an unassigned parameter. To solve this problem, you might try recording **methodA** instead. The following flowchart describes the process in detail:



If a thread in an earlier call has already initialized the object, see the "*Pattern: Irretrievable Object and Duplicate Calls in Thread*" section.

Here is an abridged step-by-step guide to the bottom-up approach which will is effective in most situations:

1) Enable stack tracing as described in the "*Turn on Stack Tracing*" section.

2) Create a filter based on the correct environment (depending on the communication transport technology).

3) Record and generate the script.

4) Replay the script.

5) If you encounter errors, consult the "*.NET Recording Tips and Techniques*" section for help in identifying the business layer and resolving the errors.

6) Determine which assembly implements the business layer.

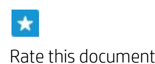
7) In the Filter Manager dialog for your filter (see "*Configuring .NET Protocol Recording*" in VuGen's User Guide), add the business layer's assembly, and include the business layer's type and methods in the filter. If you can't see the type and/or methods in the filter UI, you will need to open the filter's XML file in an XML editor or text editor, and explicitly add them. See the section "*Manually Edit the Filter File to Record non-Public Types and Members*" for more information.

Repeat steps 3 to 7 until everything works as expected.

Learn more at hp.com/go/loadrunner

Sign up for updates
hp.com/go/getupdated

Share with colleagues


Rate this document

© Copyright 2013 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.