Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Battle of the Code Generators

Source Code Accompanies This Article. Download It Now.

- codegen.zip
- codegen.txt

Code generation involves generating source code in some target programming language from some simpler input.

May 01, 2005
URL:http://www.drdobbs.com/architecture-and-design/battle-of-the-code-generators/184406075

Gigi is a software developer specializing in object-oriented and component-oriented programming using C++. He can be contacted at gigi_sayfan@playstation .sony.com.

Most nontrivial programs are event based. The events can be UI events, incoming network packets, or hardware interrupts. These events are usually translated to software events in the form of Windows messages, callback functions, delegates, and the like. The Observer design pattern suggests one way of doing event-driven programming. The Model-View-Controller (MVC) is a GUI-specific architecture for updating multiple views of a data model and handling UI events. Many large-scale programs are heavily event based and employ some kind of event propagation system. In this article, I examine one of my favorite event propagation designs—Interface-based publish-subscribe event propagation. Note that native .NET delegates and events are a different implementation of a publish-subscribe with different semantics.

The idea behind interface-based publish-subscribe event propagation is this:

- There is a globally accessible *Switchboard* object that every event goes through.
- Event producers send events to the *Switchboard* (publishing).
- Event consumers subscribe for specific event interfaces.
- Whenever the *Switchboard* receives an event, it propagates it to all the subscribers.
- *Event* consumers may unsubscribe at any time.

The upside of this approach include:

- Complete decoupling of event producers and event consumers.
- Allows passing any type of data with the event.
- Explicit event names.
- Type safety; no casting whatsoever.
- Missing/misspelled/wrong signature events/event handlers are detected at build time.
- Allows grouping of related events.
- Allows subscription/unsubscription of an entire interface in one call.
- Easy debugging even when events are chained (meaningful call stack).

On the other hand, the downside is:

- The *Switchboard* object is a maintenance nightmare.
- Event consumers must implement a handler for every event on an interface they subscribe to.
- Sometimes, it isn't easy to decide how to group events to interfaces.
- It isn't easy to queue events, persist events, or otherwise handle events as objects.

The EventDemo Program

Listings One through Four present an app that demonstrates the publish-subscribe mechanism. (The complete listings and related files are available electronically; see "Resource Center," page 5.) Listing One contains the definition of the event interfaces. There are two interfaces: *ISomeInterface*, which has methods with different signatures; and *IAnotherInterface*, which has methods with no arguments (just to show that the code works with multiple interfaces). The *[EventInterface]* custom attribute that decorates each interface is the key for detecting the event interfaces during code generation. I'll get back to it shortly. The return value of all the event methods is void. This isn't mandatory and *EventSwitchboard* can handle arbitrary return values just like it handles arbitrary lists of arguments. The reason is that the essence of event propagation is that senders have no idea who (if anyone) is listening and handling its events; hence, they can't take any meaningful action by intercepting a return value. There is also the issue of getting multiple return values from multiple handlers. If the sender needs some information from a receiver of an event, then these two objects actually exchange messages according to some protocol and it is not event propagation.

Listing Two is the *Switchboard* class, which is derived from all the event interfaces (C#/CLR allows single implementation inheritance but multiple interface inheritance). It has a pair of *Subscribe*/*Unsubscribe* methods for each interface, an *ArrayList* field per interface that holds all the subscribers to this interface, and implementation of each interface method that simply forwards the event to every subscriber. Granted, this is less than exciting, especially when you consider real-life production systems that might have tens/hundreds of event interfaces with hundreds/thousands of events. This class is bad—real bad. The only good thing about it is that it is a Singleton, so you don't have to deal with more than one (unless you want to, and sometimes you do).

Listing Three contains three *EventHandlers* that implement various event interfaces. All they do is write to the console some text that identifies them and the event they received. *EventHandler_1* and *EventHandler_2* implement one interface each and *EventHandler_3* implements two interfaces. Trivial so far.

The *MainClass* in Listing Four gets the show on the road. It gets a reference to the *Switchboard*. It creates instances of all the handlers and starts sending events to the *Switchboard*, which diligently forwards each event to its subscribers—which promptly blurb to the console (see Figure 1). Finally, *Main* unsubscribes all the event handlers. This is not necessary in this example, but in real systems where subscribers are created and destroyed and events fly back and forth, you'd better unsubscribe every object that needs to be destroyed; otherwise, it continues to live forever due to the reference the *Switchboard* holds. In nonmanaged C++ systems, it is easy to destroy an object without unsubscribing, and the next event that *Switchboard* tries to deliver crashes the system.

The problem is that the *Switchboard* class is a world-class abomination. It must be modified every time someone, somewhere adds a new event or modifies an existing event. It is completely inadequate if third-party developers need to create new events and use your *Switchboard*. If that's not enough, *Switchboard*'s code is boring and error-prone due to copy-and-paste reuse tactics. It is easy for the *Switchboard* to receive event A and propagate event B. This kind of error can be difficult to track in a dynamic system, especially if it's multithreaded.

The solution is tied up with the capability of the *Switchboard* class to be generated automatically without human intervention. Here is the general idea: Every event interface will be annotated with a custom *[EventInterface]* attribute. A special switchboard generator program traverses all the candidate assemblies that may contain interfaces. The generator reads the metadata information of every type in the candidate assemblies, detects the event interfaces, and for each event interface, it magically conjures the appropriate code in the *Switchboard* class (it will create). The *SwitchboardGenerator* can be run at development time against a known set of assemblies as a prebuild step or it can be run at runtime to generate the *Switchboard* class on-the-fly to address third-party assemblies that send/listen to events. In the latter case, a more sophisticated approach is necessary because the third-party assemblies must be programmed against some existing object that implements the event interfaces they use. This requires a dummy object that is replaced with the *Switchboard* after it is generated.

What Is Code Generation?

Code generation has different meanings to different people (or different programs). For purposes here, I define it as generating source code in some target programming language from some simpler input (domain-specific languages, for instance) via a code-generator program plus some optional templates plus some optional configuration files. This is the most interesting type of code generation for programmers. The generated code should not be modified manually by humans (it is okay to provide it as input to yet another code generator or some postprocessing program) and it should not be checked into the source-control repository; it should be treated as an intermediate product of the build process. Finally, you probably want to keep the generated code around for debugging purposes.

Creating and using code generators is fun, especially compared to performing the same boring task manually again and again. It definitely raises the level of abstraction with which you understand the problem you try to solve. It also lets you generate efficient code targeted for a specific situation.

Code Generation versus Data-Driven Programming

With data-driven programming, the flow of the program is controlled by external data and is not hard coded. Code generation and data-driven programming are similar but have some important differences.

Data-driven programming is:

- More flexible (decisions can be made according to real-time data).
- Incurs performance overhead.
- Much more difficult to debug.

Code generation is:

- Less flexible in general (all decisions must be made at code-generation time).
- Excellent performance (just like hand-coded in general).
- Potentially excellent debugging (depends on your attention to detail when generating code).

The boundaries are not clear. Code generation can generate data-driven code. Data-driven code can decide to compile some code before execution, and so on. For example, the CLR's JIT compiler generates code at load time (on a function-by-function basis) and can perform all kinds of interesting optimizations according to runtime conditions.

Every program that parses command-line arguments or reads an external configuration file or registry entry is probably data-driven to a degree.

CLR Metadata, Custom Attributes, and Reflection

The .NET platform elevates the programming model, tries to solve pervasive problems, and provides countless services, including: modern verifiable type system; versioning (side-by-side installation); deployment (XCOPY installs); code access security; streamlined interlanguage interoperability (including inheritance); streamlined intraprocess, interprocess, and cross-machine communication (AppDomains and remoting); and a huge and consistent class library. Many of these facilities—and especially the almost transparent way by which they are exposed to you—are based on the rich metadata embedded in every assembly and is available to the (JIT) compiler and your code at runtime. The metadata is stored in a binary format in the CLR Portable Executable (PE) file when you compile code. The key observation here is that assemblies are self-describing. When an assembly is loaded into memory, the CLR loader (and JIT compiler) has all the information it needs right there in the assembly PE file. The assembly's metadata is also loaded into memory and is available for runtime inspection by the CLR and your code. This is a huge jump from the COM world where you had to mess around with registry entries, IDL files, and type libraries.

The CLR lets you annotate elements of code with attributes—declarative tags that contain additional information and are available at runtime through reflection. The compiler emits attributes into the assembly's metadata. There are predefined attributes and custom attributes you can define. Take another look at Listing One. The *[EventInterface]* attribute is a custom attribute. Figure 2 shows how the attribute is stored as part of the metadata of the *ISomeInterface* interface. The CLR uses many predefined attributes such as *[Serialized]* and *[WebMethod]*. Attributes are actually instances of classes that derive from the *System.Attribute* base class. Listing Five contains the definition of *[EventInterface]*. Attribute classes may be named arbitrarily but, by convention, they always have an *Attribute* suffix. You can omit the *Attribute* suffix when using the attribute (hence, *[EventInterface]* and not *[EventInterfaceAttribute]*). When defining an attribute, you have to use yet another attribute—*AttributeUsage,* which describes how the attribute is supposed to be used. You can specify what type of element your attribute may annotate, whether multiple instances of your attribute are allowed per code element, and if the attribute may be inherited by subclasses (applies only to class attributes). The *[EventInterface]* attribute applies only to interfaces (*AttributeTargets.Interface*) and doesn't allow multiple instances (*AllowMultiple = false*).

The reflection API lets you drill down the loaded assemblies and discover the types they contain. For each type, you can discover constructors, methods, interfaces, and events. For each method you can discover its return value, name, and parameters. In addition, you can access all the attributes, predefined and custom, which annotate every element. Listing Six demonstrates how to dump an entire assembly to the console. The API is straightforward: You call *GetXXX* methods to get collections of objects, such as *Type*, *MethodInfo*, and *ParameterInfo*, then you iterate through the collection and can access properties such as *MethodInfo.Name*, *MethodInfo.ReturnType*, and *ParameterInfo.DefaultValue*. The reflection API also allows invoking methods on types and instances using information discovered at runtime using the *Type.InvokeMember* method.

The SwitchboardGenerator MainClass

SwitchboardGenerator is a program (available electronically; see "Resource Center," page 5) that generates a *Switchboard* class implementation that supports arbitrary event interfaces. It accepts various arguments on the command line, so much of its operation can be configured without actually modifying the code. How does it work? The input consists of the full path of the generated *Switchboard* class, the namespace of the generated *Switchboard* class, the custom attribute type name that identifies event interfaces, and a list of paths of assemblies that need to be scanned for event interfaces. SwitchboardGenerator iterates over the types in every assembly; for each interface it finds, it checks to see that it is annotated with the custom attribute (passed on the command line) and adds the matching interfaces to a list. After compiling the interface list, it launches the selected code generator dynamically and invokes its *Generate* method. The naming convention I use is that all code generators are called "SwitchboardGenerator" (and support a static '*Generate*' method, but they belong to a different namespace). The namespace is passed on the command and the full name of the selected code generator is *{namespace}.SwitchboardGenerator* (see Listing Six). The code generators use the interface list to create the corresponding *Switchboard* class. SwitchboardGenerator expects five

arguments on the command line in this order: qualified filename of the generated *Switchboard* class, the namespace that contains it, the custom attribute (including namespace) that identifies event interfaces, which code generator to use (CodeDOM or TextTemplate), and the assembly that contains the event interfaces. Figure 3 illustrates a short session with the SwitchboardGenerator.

Template-Based Code Generator

The core of the template-based code generator is simple and doesn't rely on any CLR- or Windows-specific feature. I have used similar generators on UNIX-based systems and embedded systems (such as PlayStation). Usually, I use Python to implement it since that language's string manipulation capabilities are excellent. However, since I wanted to contrast it against the CodeDOM generator, I implemented it in C#. The goal is to simply generate some code. Most of it is just boilerplate code but requires embedding some dynamic parameters. The solution is to have a text template that contains the boilerplate code, have special place holders for the dynamic parameters (à la *sprintf* masks), and replace them at generation time. Listings Eight, Nine, and Ten (available electronically) contain the templates I used to generate the *Switchboard* class in Listing Two. The *$$$-something-$$$* is my notation for place holders that should be replaced when generating the code. This is pretty much a glorified *sprintf*, but generating the text for the place holders is not trivial. Listing Eight contains the *Switchboard* class skeleton. The *$$$-subscribe_unsubscribe_methods-$$$* and *$$$-event_methods-$$$* are created using multiple instances of the templates in Listing Nine and Listing Ten, correspondingly. All the information necessary to replace the place holders with real data is available through the reflection API from the interface list provided to the generator. Note that if I wanted to decouple the code generation completely from the CLR, I could extract this information and provide it to a generic generator through a dictionary of *key=value* pairs, but preparing it would be a lot of unnecessary work, when the reflection API provides a nice object model. As you recall, the *MainClass* (Listing Six) iterated over all the interfaces and if they had the custom *[EventInterface]* attribute, it added them to the list. The generator (Listing Eleven, available electronically) walks through this list and for each interface, it updates the base interfaces (the *Switchboard* must derive from every event interface to forward them) and creates subscribe and unsubscribe methods; for each event of the interface, it creates a corresponding event handler and finally creates a subscriber list for this interface. The most interesting part is creating the event methods because the signature of each event is arbitrary. The *CreateEvent* method has to gather the following information: return type, event name, parameter types (list of all parameter types), parameter names (list of all parameter names), the interface name itself, and the name of the subscriber list. The *CreateEvent* method traverses the *MethodInfo* object to extract most of this information (requires access to properties and iterating over the *ParameterInfo* collection) and uses the *GetListName* helper method.

CodeDOM Code Generator

The *System.CodeDOM* namespace contains myriad classes that lets you generate a code abstract object model that represents some code, and generate source files from it. There is also a *System.CodeDOM.Compiler* namespace that lets you compile code at runtime, but I don't deal with this namespace in this article. Here are the steps to generate code using the CodeDOM:

1. Create a root CodeNamespace.
2. Add types to the namespace.
3. Add members to the types (methods, properties, events, and fields).
4. Add code statements to the methods.
5. Request a specific provider and a code generator from that provider (*CSharpCodeProvider*, for instance).
6. Generate the code and write it to a file.

The promise of the CodeDOM is enormous. You may generate code from the same CodeDOM graph to multiple languages. As long as a language provider exists, you may generate code for this language. Internally, the web services infrastructure uses CodeDOM to generate client-side proxies for you. However, the reality is less then perfect. The CodeDOM doesn't cover every CLR feature (unary operators, for example), it doesn't give you complete control on the format of the generated code, and some (many?) language constructs are missing. For example, there is no way to generate C# *foreach* or *while* statements using the CodeDOM. The only iteration statement is *'for'*. The worst drawback of the CodeDOM is its extreme verbosity. This may be okay for automatically generated code you never see and debug (like web services proxies), but if you want to integrate code generation into your software-development arsenal, it just doesn't cut it. You should be able to debug easily through the generated code and you should be able to easily modify the code generator itself.

The CodeDOM needs a serious facelift to become useful for custom code generation. Listing Twelve (available electronically) contains the CodeDOM switchboard generator. The basic structure is similar to the template-based code generator—it gets the information it needs through the reflection API from the interface list, it creates a skeleton *Switchboard* class, subscribe/unsubscribe methods, and event methods. However, the code generation itself is done by creating *CodeDOM* classes and painstakingly composing them together. The end result is not on par with the complete control and ease of modification that the template-based code generator affords. For example, I like to put the private fields at the bottom of the class definition, but the CodeDOM code generator insists on putting them at the top; and the whitespace is not exactly the way I like but there is nothing I can do about it. See Listing Thirteen (available electronically) for the results of the CodeDOM code generator. Note, in particular, how ugly the *'for'* construct with the explicit *GetEnumerator()* and *MoveNext()* looks. Another problem that undermines the multilanguage capability of the CodeDOM is the snippet classes (*CodeSnippetStatement*, *CodeSnippetExpression*, *CodeSnippetTypeMember*). These classes represent literal text that is pasted in the middle of the generated code as is. Of course, you can't write a cross-language snippet because every language has a different syntax. This means that using snippets destroy cross-language code generation. Since some language constructs are not implemented yet, you must sometimes use snippets. Moreover, because creating a proper CodeDOM tree is so complicated, sometimes the easy way is just to use snippets (I shamelessly admit I did it).

Using a Code Generator

The code generators I present here reflect a successfully built assembly. If I would have modified an event interface and tried to regenerate the *Switchboard*, the process would fail. The reason is that the previously generated *Switchboard* is part of the same assembly (*EventDemo*) and it doesn't work with the modified event interface. To regenerate it, I have to fix it manually first. What? That's right, it doesn't make much sense. The solution is to separate the event interfaces (or in general, all the input to the code generator) from the *Switchboard* class (or in general, all the output of the code generator). This can be done by putting either the event interfaces or the *Switchboard* class in a different assembly. I recommend putting the interfaces in a separate assembly and putting the *Switchboard* class in the same assembly as the rest of the application (*EventDemo*). Now, you don't have the chicken-and-egg problem. The assembly that contains the *Switchboard* class should reference and depend (build-wise) on the assembly that contains the interfaces. The SwitchboardGenerator program itself should be invoked as a build step of the main assembly that contains the *Switchboard* class. This way, an up-to-date *Switchboard* class is generated before starting to build its assembly.

Conclusion

Reflection plus code generation are a dynamic duo. They can be used for many purposes and not just to write various stubs and proxies. Use them wisely and forget about CodeDOM until Longhorn ships (at least). Maybe by then, some genius will find a way to make it simpler.

DDJ

Listing One

```
using System;
namespace EventDemo
{
   [EventInterface]
    public interface ISomeInterface
```

```
    {
        void OnThis(int x);
        void OnThat(string s);
        void OnTheOther(bool b, string s);
    };
    [EventInterface]
    public interface IAnotherInterface
    {
        void OnBim();
        void OnBam();
        void OnBom();
    };
}
```

Back to article

Listing Two

```
// Auto-generated Switchboard class
using System.Collections;
using System.Diagnostics;
namespace EventDemo
{
    class Switchboard :
        EventDemo.ISomeInterface,
        EventDemo.IAnotherInterface
    {
        public static Switchboard Instance
        {
            get { return m_instance; }
        }
        public bool Subscribe(EventDemo.ISomeInterface sink)
        {
            if (m_someInterfaceList.Contains(sink))
            {
                Debug.Assert(false);
                return false;
            }
            m_someInterfaceList.Add(sink);
            return true;
        }
        public bool Unsubscribe(EventDemo.ISomeInterface sink)
        {
            if (!m_someInterfaceList.Contains(sink))
            {
                Debug.Assert(false);
                return false;
            }
            m_someInterfaceList.Remove(sink);
            return true;
        }
        public bool Subscribe(EventDemo.IAnotherInterface sink)
        {
            if (m_anotherInterfaceList.Contains(sink))
           {
                Debug.Assert(false);
                return false;
            }
            m_anotherInterfaceList.Add(sink);
            return true;
        }
        public bool Unsubscribe(EventDemo.IAnotherInterface sink)
        {
            if (!m_anotherInterfaceList.Contains(sink))
            {
                Debug.Assert(false);
                return false;
            }
            m_anotherInterfaceList.Remove(sink);
            return true;
        }
        public void OnThis(System.Int32 x)
        {
          foreach (EventDemo.ISomeInterface subscriber in m_someInterfaceList)
              subscriber.OnThis(x);
        }
        public void OnThat(System.String s)
        {
          foreach (EventDemo.ISomeInterface subscriber in m_someInterfaceList)
                subscriber.OnThat(s);
        }
        public void OnTheOther(System.Boolean b,System.String s)
        {
          foreach (EventDemo.ISomeInterface subscriber in m_someInterfaceList)
                subscriber.OnTheOther(b,s);
        }
        public void OnBim()
        {
          foreach (EventDemo.IAnotherInterface subscriber in
                       m_anotherInterfaceList) subscriber.OnBim();
        }
        public void OnBam()
        {
          foreach (EventDemo.IAnotherInterface subscriber in
                       m_anotherInterfaceList) subscriber.OnBam();
        }
        public void OnBom()
        {
            foreach (EventDemo.IAnotherInterface subscriber in
                       m_anotherInterfaceList) subscriber.OnBom();
        }
        ArrayList m_someInterfaceList = new ArrayList();
        ArrayList m_anotherInterfaceList = new ArrayList();
        static Switchboard m_instance = new Switchboard();
    }
```

```
}
```

Back to article

Listing Three

```
namespace EventDemo
using System;

{
    class EventHandler_1 : ISomeInterface
    {
        public void OnThis(int x)
        {
            Console.WriteLine("{0} received OnThis({1})", GetType().Name, x);
        }
        public void OnThat(string s)
        {
            Console.WriteLine("{0} received OnThat({1})", GetType().Name, s);
        }
        public void OnTheOther(bool b, string s)
        {
            Console.WriteLine("{0} received OnTheOther({1}, {2})",
                                                    GetType().Name, b, s);
        }
    }
    class EventHandler_2 : IAnotherInterface
    {
        public void OnBim()
        {
            Console.WriteLine("{0} received OnBim()", GetType().Name);
        }
        public void OnBam()
        {
            Console.WriteLine("{0} received OnBam()", GetType().Name);
        }
        public void OnBom()
        {
            Console.WriteLine("{0} received OnBom()", GetType().Name);
        }
    }
    class EventHandler_3 : ISomeInterface, IAnotherInterface
    {
        public void OnThis(int x)
        {
            Console.WriteLine("{0} received OnThis({1})", GetType().Name, x);
        }
        public void OnThat(string s)
        {
            Console.WriteLine("{0} received OnThat({1})", GetType().Name, s);
        }
        public void OnTheOther(bool b, string s)
        {
            Console.WriteLine("{0} received OnTheOther({1}, {2})",
                                                   GetType().Name, b, s);
        }
        public void OnBim()
        {
            Console.WriteLine("{0} received OnBim()", GetType().Name);
        }
        public void OnBam()
        {
           Console.WriteLine("{0} received OnBam()", GetType().Name);
        }
        public void OnBom()
        {
            Console.WriteLine("{0} received OnBom()", GetType().Name);
        }
    }
}
```

Back to article

Listing Four

```
using System;
namespace EventDemo
{
    class MainClass
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            Switchboard s = Switchboard.Instance;
            EventHandler_1 eh_1 = new EventHandler_1();
            EventHandler_2 eh_2 = new EventHandler_2();
            EventHandler_3 eh_3 = new EventHandler_3();
            s.Subscribe(eh_1 as ISomeInterface);
            s.Subscribe(eh_2 as IAnotherInterface);
            s.Subscribe(eh_3 as ISomeInterface);
            s.Subscribe(eh_3 as IAnotherInterface);
            s.OnThis(5);
            s.OnThat("Yeahh, it works!!!");
            s.OnTheOther(true, "Yeahh, it works!!!");
            s.OnBim();
            s.OnBam();
            s.OnBom();
            s.Unsubscribe(eh_1 as ISomeInterface);
            s.Unsubscribe(eh_2 as IAnotherInterface);
            s.Unsubscribe(eh_3 as ISomeInterface);
```

```
            s.Unsubscribe(eh_3 as IAnotherInterface);
        }
    }
}
```

Back to article

Listing Five

```
using System;
namespace EventDemo
{
    [AttributeUsage( AttributeTargets.Interface, AllowMultiple = false)]
    public class EventInterfaceAttribute : System.Attribute
    {
   }
}
```

Back to article

Listing Six

```
using System;
using System.Reflection;
namespace ReflectionDemo
{
    class AssemblyDumper
    {
        static public void Dump(Assembly a)
        {
            Type[] types = a.GetTypes();
            foreach (Type type in types)
                DumpType(type);
        }
        static private void DumpType(Type type)
        {
            Console.WriteLine(" ---- {0} ----", type.FullName);
            MethodInfo[] methods = type.GetMethods();
            foreach (MethodInfo method in methods)
            {
                Console.Write("{0} {1}(", method.ReturnType, method.Name);
                ParameterInfo[] parameters = method.GetParameters();
                foreach (ParameterInfo parameter in parameters)
                {
                    string typeName = parameter.ParameterType.Name;
                    string parameterName =  parameter.Name;
                    Console.Write("{0} {1}",
                           parameter.ParameterType.Name, parameter.Name);
                    if (parameter.IsOptional)
                        Console.Write(" = {0}",
                                      parameter.DefaultValue.ToString());
                    if (parameter.Position < parameters.Length-1)
                        Console.Write(", ");
                }
                Console.WriteLine(")");
            }
        }
    }
}
```

Back to article

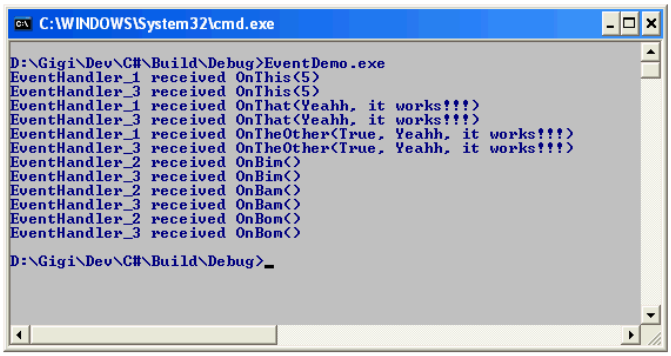

Figure 1: Forwarding events to subscribers.

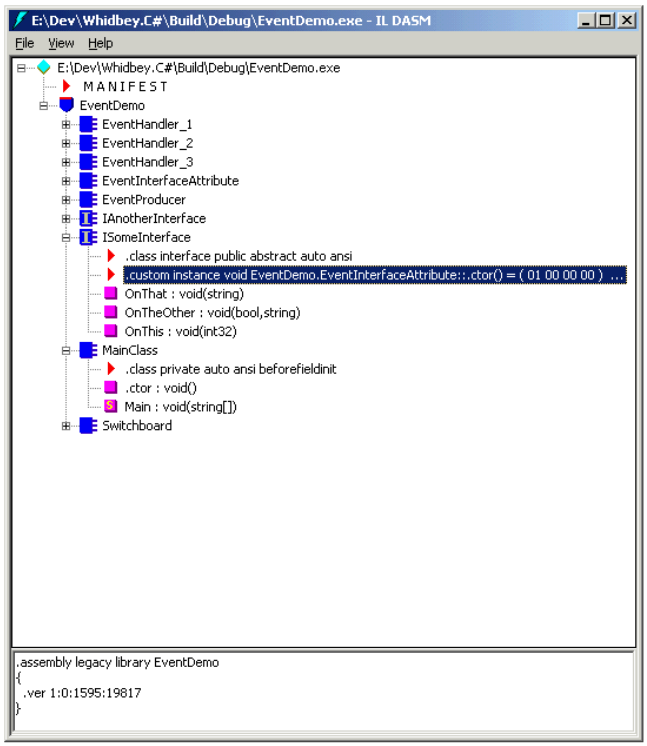

Figure 2: Storing attributes.

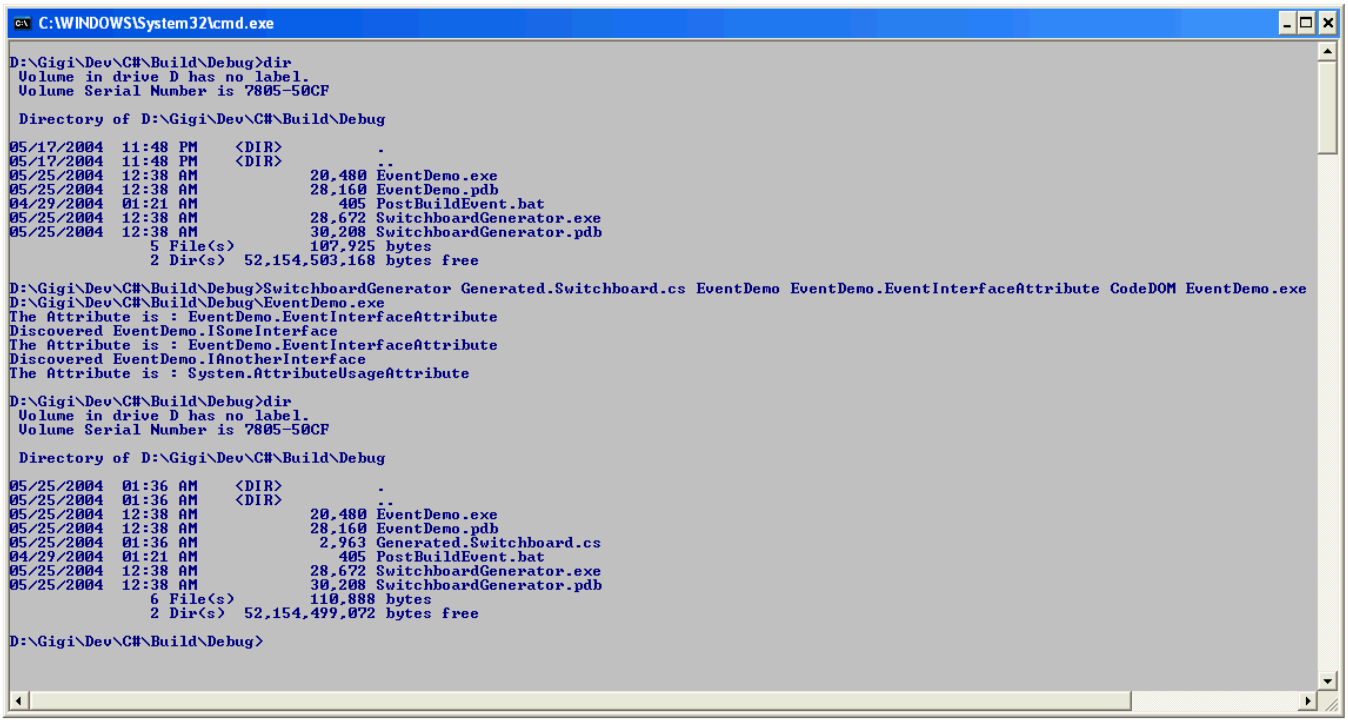

Figure 3: Using the SwitchboardGenerator program.

Terms of Service | Privacy Statement | Copyright © 2020 UBM Tech, All rights reserved.