GenTopicsUniversal developer notes

Jim Galasyn ([jgalasyn@microsoft.com](mailto:jgalasyn@microsoft.com))

26 January 2015

## Summary

GenTopicsUniversal (GTU) is a command-line tool that works with [Doxygen](http://www.stack.nl/~dimitri/doxygen/) to produce API reference documentation from source code comments. Use GTU to produce HTML documentation automatically from native C++ code, C# code, or Windows Runtime (WinRT) IDL. Clone the [GitHub repo](https://github.com/JimGalasyn/GenTopicsUniversal) to get the GTU source code.

## Usage

Start GTU from the command line:

>GenTopicsUniversal.exe [optional path to config file]

GTU is controlled by using configuration files. For more info, see the Configuration section.

## Background

GenTopicsUniversal evolved from the earlier API reference generation tool, GenTopics, which generates stub topics from the main windows.winmd file. The original GenTopics application enumerates all of the public types in windows.winmd and emits topics that writers author into later. These topics are stored in a source repository, and writers author content into them by using XMetaL or another XML editor. The topics adhere to the WDCML (Windows Developer Content Markup Language) schema. Writers are required to keep the source topics in sync with source code changes manually, which can be time-consuming and error-prone.

GenTopics is limited to reading only winmd (Windows metadata) files and generating only WDCML topic stubs. GenTopicsUniversal enables a different workflow by assuming that the source code itself is the single “source of truth”. Reference content is authored by developers and/or writers directly in the source code, so there is no intermediate store that needs to be synched with the source. GTU is designed to be scripted directly into product builds, so fresh documentation is generated and placed with each build.

## Architecture

GenTopicsUniversal comprises three tiers: Deserialization, Business Objects, and Serialization. The workflow proceeds from deserializing Doxygen’s XML output into business objects; then, business objects are serialized to HTML and other formats, *e.g.*, IntelliSense.

GTU’s business objects are classes that represent type information. The most important of these are the DefinedType and DefinedMember classes, which are the primal classes for representing arbitrary types and their members.

Each combination of deserializer, business objects, and serializer comprises a *pipeline*. A collection of related pipelines comprises a *content set*. For example, the Analog content set has pure native APIs (3D audio), WinRT APIs (for UAP), and pure managed APIs (DUSK2). Each of these API sets is represented by a corresponding pipeline.

### Business Objects

GTU must be able to represent types from any type system in the C family of languages (*e.g.*, Java, C++, C#, IDL). This means that GTU must implement a kind of meta-type system, or a “universal” type system. This would be a huge task, but fortunately the .NET Framework type system has done most of the work already. This is the type system that represents WinRT’s Application Binary Interface (ABI) in winmd and assembly files, so it’s a natural choice for building GTU’s meta-type system.

In principle, GTU can consume any XML output from Doxygen, but it hasn’t been tested with more esoteric languages, like F#.

GTU identifies five type-level language elements:

* Namespaces
* Interfaces
* Classes
* Structs
* Enums

All of these are represented by the DefinedType class, which is the base class for all of the type-level language elements. Wherever possible, GTU operates on DefinedType instances, to keep implementation at the highest level of abstraction.

Namespaces aren’t explicitly modeled as types in the .NET Framework; they’re just opaque strings used for the convenience of designers. Nevertheless, every type in the .NET type system must belong to a namespace; declarations at global scope aren’t permitted. This constraint isn’t present in pure-native type systems. GTU models namespaces as first-class types, and it creates a DefinedType instance for each namespace that it finds.

A DefinedType instance has four important properties:

* BaseTypes, represented as a collection of DefinedType instances;
* Members, represented as a collection of DefinedMember instances;
* ChildTypes, represented as a collection of DefinedType instances.
* A parent namespace, represented as a DefinedType instance.

Each type-level language element has specific constraints that define it, as shown in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| Language element | Base types | Members | Child types |
| Namespace | 1 parent namespace | 0 | 0 or more |
| Interface | 0 or more | 0 or more | 0 |
| Class | 0 or more | 0 or more | 0 |
| Struct | 0 | 0 or more fields | 0 |
| Enum | 0 | 0 or more int/char fields | 0 |

Here’s the table expressed in English:

* All types have exactly one parent namespace; types at global scope are children of the global namespace.
* Only namespaces may have child types. (This constraint could be relaxed for inner classes, but this hasn’t been necessary so far.)
* A namespace may have zero or more child types and exactly zero members;
* An interface may have zero or more members and zero or more base types;
* A class may have zero or more members and zero or more base types;
* A struct may have zero or more field members and exactly zero base types;
* An enum may have zero or more integer/char field members and exactly zero base types;

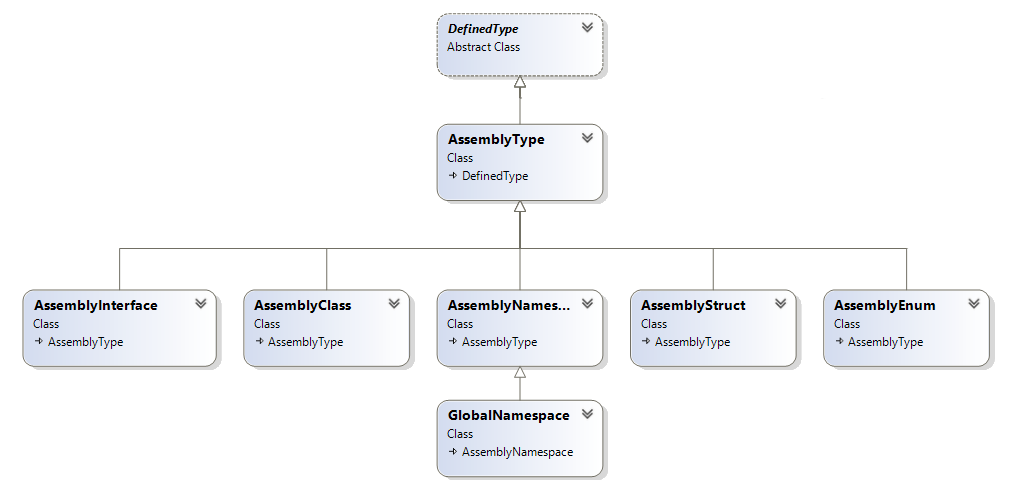
Currently, global functions aren’t supported.

A member of a type can be a property, method, event, or field and is represented as a DefinedMember instance. These are the constraints on members in the type-level language elements:

* Classes may have properties, methods, events, and fields;
* Interfaces may have properties, methods, and events;
* Structs may have only fields;
* Enums may have only integer/char fields.
* Namespaces have 0 members.

These constraints are implemented in classes that derive from DefinedType. Types that come from a winmd file or an assembly are represented by classes that derive from AssemblyType. Types that come from Doxygen output are represented by classes that derive from DoxygenType.

The following diagram shows the inheritance graph for the AssemblyType class and its derived classes. The graph for the DoxygenType class is analogous.



GTU assigns a Type property to each member-level language element. The Type property, which is a DefinedType instance, has different meanings, depending on the context:

* Property: The Type property represents the property type, which is the return value of the property’s get accessor;
* Method: The Type property represents the method’s return value;
* Event: The Type property represents the type of the event handler (delegate);
* Field: The Type property represents the type of the field.

### Deserialization

Deserialization is the process of converting a serialized (XML) representation of a type system into business objects, which is essentially an in-memory database. This database represents the *type graph*, which is the relationship of the types with each other.

The first step is to run Doxygen over the source code to extract code comments and type information. The output from this operation is a folder full of XML files that represent types. These XML files are used by GTU to associate code comments with their corresponding types. In principle, this should be straightforward, but in many cases, GTU has to do a lot of work to make these associations. In particular, Doxygen has difficulty understanding WinRT IDL, a.k.a. “RIDL”. GTU relies on the related winmd file that’s produced by the midlrt compiler to fill in holes and glue together types.

Here are the steps that GTU follows to deserialize Doxygen’s XML output.

1. GTU inspects index.xml, which is emitted by Doxygen as a manifest. This manifest file lists all of the types that Doxygen discovered in the source code, and it lists the corresponding XML files for each type. Usually, there’s one XML file for each namespace, interface, class, and struct. Enums are listed inside the parent namespace file. For WinRT, this rule may not hold; Doxygen doesn’t understand runtime classes, and they show up erroneously as “variables” in namespace files. A number of heuristic patches like this are required to make sense of Doxygen output on RIDL.
2. GTU reads each file specified in index.xml, and creates a low-level representation of the type information in the XML file. This representation is a class named DoxType, which has properties that mimic Doxygen’s compound.xsd schema for modeling type information. Doxygen represents each type as a “compound” element in XML.
3. GTU creates a DoxygenDeserializer instance, which wraps each unique DoxType instance with a higher level of abstraction, implemented in the DoxygenType class. The DoxType instance is cached for later use in the DoxygenType.UnderlyingType property. The DoxygenDeserializer returns a list of DefinedType instances to the pipeline.
4. For pure native and pure managed pipelines, this completes the process of deserializing XML into business objects. For the RidlPipeline and ManagedAssembly pipelines, more work is necessary.
5. A GTU pipeline can take another input, which is either a winmd file or a managed assembly. For the RidlPipeline, the winmd is required, because not all of the type information is correctly represented in Doxygen’s XML output. For a pure managed, the assembly is convenient for providing type metadata that isn’t present in Doxygen’s XML output.
6. The RidlPipeline inspects the contents of the provided winmd file by creating an AssemblyDeserializer, which creates a collection of ObservableType instances. The ObservableType class is reused *in toto* from the original GenTopics implementation and implements a high-level abstraction over Lightweight Metadata Reflection (LMR). The AssemblyDeserializer creates an instance of the AssemblyType class around each ObservableType instance, which is cached for later use in the AssemblyType.UnderlyingType property. The AssemblyDeserializer returns a list of DefinedType instances to the pipeline. The ManagedAssembly pipeline follows the same workflow.
7. The RidlPipeline now has two collections of types, one that represents the types found by Doxygen and another that represents types in the winmd assembly. The code comments that will become the reference content are in the DoxygenType collection, and the authoritative type metadata is in the AssemblyType collection. The RidlDeserializer must rationalize the two type systems, and this task is non-trivial, because in general, API signatures in IDL are significantly different when they’re represented in a winmd file.

# Future work

In lean startup terminology, GTU 1.0 is very much a “minimum viable product”. Currently, it suffices for a number of content sets, but it can be expanded and elaborated in many ways.

## Expand configuration options

## Extension points