1: MEF Error Handling & Debugging

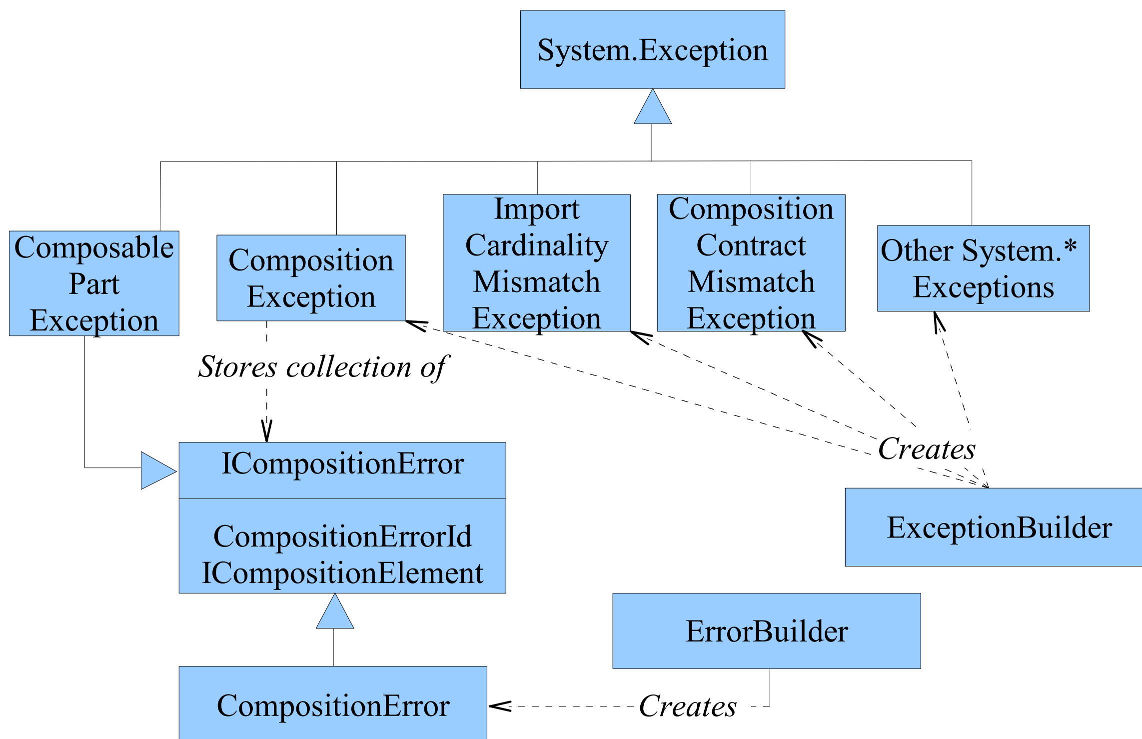
MEF delivers the possibility of building highly extensible and composable applications, but with so must flexibility comes some costs. Specifically, when things go wrong, it can be somewhat complex to discover what the problem is (usually with MEF, when you know the problem, it is easy to fix it).

To assist application developers, the MEF team in Redmond have put in quite a bit of effort in helping with debugging and error handling.

1.1: Big Picture

Many MEF types are involved in error handling and the following diagram shows how they are related.

Error Handling In MEF



1.2: CompositionElement

Due to the nature of MEF, with parts working with other parts in a cascade, a problem may well occur far from where the application developer has first contact with MEF (e.g. `CompositionContainer`). When MEF error handling

wishes to inform the application developer that there is a problem with something, it first has to say what “thing” is involved. It could be a `ComposablePart`, a `ComposablePartCatalog`, an `ExportProvider` or something else. MEF use the term *composition element* to describe any element that participates in composition. This is primarily used for error reporting. The role of composition elements is simply to provide naming and ancestry information. A number of types related to composition elements are provided in the Primitives namespace.

A very simple interface, `ICompositionElement` (from the System.ComponentModel.Composition.Primitives namespace) defines two properties (getters only). The first is called `DisplayName` (of type `String`) and it contains a textual name for the element suitable for humans. The second it called `Origin` (of type `ICompositionElement`) and it is used to say from which “parent” element the current composition element came from. Sometimes this is null (e.g. `TypeCatalog`) while other times it can be very useful in highlighting the chain of elements that resulted in the problems of a particular child element.

Often MEF needs to serialize composition elements (e.g. when used as properties in exceptions). Implementations of `ICompositionElement` that are serializable can be used “as is”, but often they are not, and so MEF provides a `SerializableCompositionElement` (derived from `ICompositionElement`) to act as what it calls a “serializable placeholder”. This contains the `DisplayName` and `Origin` of the element being wrapped.

Many MEF types do not implement `ICompositionElement`, and when needed in these situations MEF provides a class called `CompositionElement` (derived from `SerializableCompositionElement`). It has an extra property (getter only), `UnderlyingObject`, which returns the object for which this instance of `CompositionElement` is acting for. This particular property is not serializable.

To improve the debugging experience for composition elements there is a class called `CompositionElementDebuggerProxy`. The `CompositionElement` type is marked with this attribute, but `ICompositionElement` and `SerializableCompositionElement` are not. In the debugger, it replaces the `CompositionElement` with the `UnderlyingObject`. When you are using the Visual Studio debugger to look at exception information about, say, a MEF catalog error, this is what you are examining.

The `CompositionElementExtensions` class contains extension methods for composition elements. These methods convert from an `ICompositionElement` to a `SerializableCompositionElement`, and convert from a range of types (`Export`, `ExportDefinition`, `ImportDefinition`, `ComposablePart` or

`ComposablePartDefinition`) to an `ICompositionElement` (the instance returned is of type `CompositionElement`).

1.3: Composition Errors

The main use of composition elements is for reporting error information. MEF uses the term *composition error* to describe errors that occur during composition. A number of types related to composition errors are provided in the main System.ComponentModel.Composition namespace.

The `ICompositionError` interface has three properties:

- `Id` (of type `CompositionErrorId`)
- `Element` (of type `ICompositionElement`)
- `InnerException` (of type `Exception`)

The `Id` identifies the type of error, the element identifies in which element it occurred and the inner exception provides details of the error. `CompositionErrorId` is an enum with a list of possible errors:

```
Unknown = 0,
InvalidExportMetadata,
RequiredMetadataNotFound,
UnsupportedExportType,
ImportNotSetOnPart,
CompositionEngine_ComposeTookTooManyIterations,
CompositionEngine_ImportCardinalityMismatch,
CompositionEngine_PartCycle,
CompositionEngine_PartCannotSetImport,
CompositionEngine_PartCannotGetExportedObject,
CompositionEngine_PartCannotActivate,
ReflectionModel_PartConstructorMissing,
ReflectionModel_PartConstructorThrewException,
ReflectionModel_PartOnImportsSatisfiedThrewException,
ReflectionModel_ExportNotReadable,
ReflectionModel_ExportThrewException,
ReflectionModel_ExportMethodTooManyParameters,
ReflectionModel_ImportNotWritable,
ReflectionModel_ImportThrewException,
ReflectionModel_ImportNotAssignableFromExport,
ReflectionModel_ImportCollectionNull,
ReflectionModel_ImportCollectionNotWritable,
ReflectionModel_ImportCollectionConstructionThrewException,
ReflectionModel_ImportCollectionGetThrewException,
ReflectionModel_ImportCollectionIsReadOnlyThrewException,
ReflectionModel_ImportCollectionClearThrewException,
ReflectionModel_ImportCollectionAddThrewException,
Adapter_CannotAdaptNullOrEmptyFromOrToContract,
Adapter_CannotAdaptFromAndToSameContract,
Adapter_ContractMismatch,
Adapter_TypeMismatch,
Adapter_ExceptionDuringAdapt
```

As you can see, there is plenty of potential for errors in MEF, so it is very important to be able to describe them accurately.

MEF provides an implementation of `ICompositionError` called `CompositionError` and this is what is used mostly in MEF code. `CompositionError` is serializable and has four major properties, id, element, exception and a description string. It has a number of internal static `Create` methods which are used by MEF code when creating `CompositionError` instances.

`CompositionError` is marked with the `CompositionErrorDebuggerProxy` attribute, and this shows the `Description`, `Element` and `Exception` in the debugger, and hides the `Id`.

One point to note is that `CompositionError` is public and it implements `ICompositionError`, which is internal.

```
public class CompositionError : ICompositionError {...}
internal interface ICompositonError {...}
```

A public class cannot derive from a base internal class (the derived type can only have the same, or a more restrictive, access modifier). However, `ICompositionError` is an interface and `CompositionError` is using explicit interface implementation to implement it (see section 13.4.1 of the C# Language Specification on the MSDN for details).

1.4: Exception Handling

`System.ComponentModel.Composition.ExceptionBuilder` is a helper class to build localizable exceptions. It has a method to build an instance of each exception type that is needed, including `ArgumentException`, `ObjectDisposedException`, `NotImplementedException`, `ArgumentException` (all standard .NET exceptions) and `System.ComponentModel.Composition.CompositionException`.

`CompositionException` is used for MEF-specific exceptions and includes code to list in full the paths involved and the `CompositionError`(s) detected.

There are two additional custom exceptions defined in System.ComponentModel.Composition:

- `ImportCardinalityMismatchException`: cardinality of imports does not match cardinality of available exports
- `CompositionContractMismatchException` – when there is a problem casting the exported type to the contract expected by the import

There is one exception defined in the Primitives namespace:

- `Primitives.ComposablePartException`

It is noted these exceptions derive from `System.Exception` and not `System.SystemException`. (As an aside, when defining exceptions in your own applications, the current advice is to derive from `System.Exception`, and not `System.ApplicationException`. Originally `ApplicationException` was the preferred option, but it did not deliver much benefit, so now the recommendation is simply to derive from `System.Exception` directly (See its MSDN reference page for more details). All these exceptions are serializable, which is very important when traversing any AppDomain boundaries.

`System.ComponentModel.Composition.ErrorBuilder` is a helper class with static method to construct `CompositionError` instances for various reasons (`CreateImportCardinalityMismatch`, `CreateImportCardinalityMismatch`, `CreatePartCannotActivate`, `CreatePartCannotSetImport`, `CreateCannotGetExportedObject`, `CreatePartCycle`).

The MEF developers have put great effort into returning informative error information to application developers when errors occur. So, when your code detects an exception, pay attention to what it says.

1.5: Debugger Attributes

The MEF source makes frequent use of a number of debugger attributes from the System.Diagnostics namespace:

- `DebuggerDisplay` – controls the information that is display in the debugger, specifically `Name`, `Target`, `TargetType`, `Type` and `Value`
- `DebuggerStepThrough` – skips non-important code in the debugger
- `DebuggerTypeProxy` – supplies a (usually more informative) proxy for use in the debugger (debugger shows the proxy type)
- `DebuggerBrowsable` – how a member is to be shown in the debugger; this sets a `DebuggerBrowsableState` enumeration (`Never`, `Collapsed`, `RootHidden`)

It you are browsing code that makes use of MEF types and what you see in the debugger is not what you expect, pay attention to these debugger attributes. Most of the time, they help you by providing clearer information, but on occasion can trip you up by showing a constrained (or even different) representation of the actual type under examination.