Hosting the .NET Composition Primitives

Nicholas Blumhardt (niblumha@microsoft.com)

1 Contents

2 Introduction 3
2.1 Unity Example 4
3 Composition Primitives 4
3.1 ComposablePart 5
3.2 ExportDefinition 5
3.3 ImportDefinition 6
3.4 ComposablePartDefinition 6
3.5 ComposablePartCatalog 6
4 The Role of Programming Models 7
5 Attributed Programming Model 7
5.1 Attributed Definitions 7
5.2 Consuming the Attributed Programming Model 7
6 High-Level Integration Points 8
7 Exchanging Exports 8
7.1 Executing a Constraint 9
7.2 ContractBasedImportDefinition 9
7.2.1 ContractName 9
7.2.2 RequiredTypeIdentity 10
7.2.3 RequiredMetadata 10
7.2.4 Special Cases when interchanging with Types and Names 10
7.3 Parsing or Transforming Constraints 11
7.4 Using Host-Provided Services 11
8 Standardizing Part Manipulation rather than Object Graph Wiring 11
9 Required Host Behavior 12
9.1 Compliance with ImportDefinitions 12

9.2 ComposablePart Lifecycle 12
9.3 Full Composition of Prerequisite Dependencies 12
9.4 Required Creation Policy 13
9.5 Part Creation Policy 13
9.6 Recomposition 13
10 Host-Defined Behavior 13
10.1 Lazy Deep Object Graph Instantiation 13
10.2 Circular Dependency Detection 13
10.3 Rejection 13
10.4 Defaults and Cardinality 14

2 Introduction

The .NET Framework version 4.0 includes a set of types for general component-oriented development.

The lowest-level types, known as the *Composition Primitives*, represent:

- Component instances capable of being wired together
- Component definitions, with rich metadata support
- Common query interfaces for component catalogs

Layered upon the Composition Primitives is a Type-based component construction library referred to as the *Attributed Programming Model.* The features of this model are:

- Attributes that allow regular .NET types to be marked-up as component definitions
- Types that implement the Composition Primitives by reading, instantiating and manipulating attributed types
- Component catalogs that discover and load attributed component definitions from a variety of sources

Together with the `CompositionContainer` included in the .NET Framework, the Attributed Programming Model is a complete composition system:

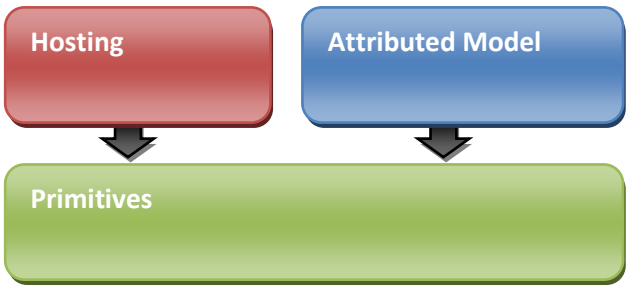

Figure 1 – .NET Composition Architecture

There are many composition systems available on the .NET Framework. Examples include:

- Inversion of Control (IoC) containers that support decoupled application architectures
- XAML readers that instantiate a textual description of an object graph
- Domain-specific frameworks like Windows Workflow Foundation and Windows Communication Foundation that use a fixed set of concepts to support a family of scenarios
- Plug-in frameworks that allow applications to be extended by third parties
- Programming languages that wire up and manipulate objects in an imperative manner

Common to all of these frameworks is an ability to take units of software that were developed independently and use them together to achieve a goal.

There is a great deal in common between most composition frameworks, so the Attributed Programming Model and especially the Composition Primitives are built in such a way that their capabilities can be used without `CompositionContainer`.

The Attributed Programming Model and `CompositionContainer` focus on application extensibility, of the kind typically supported by plug-in frameworks. This whitepaper is a guide to incorporating features of the Attributed Programming Model and Composition Primitives into composition systems with different primary goals.

2.1 Unity Example

This document uses the Unity IoC container from Microsoft Patterns and Practices as an example host composition system. Unity's primary goal is to support decoupled application architectures.

Hosting the Composition Primitives lets Unity-based applications expose extension points in a manner consistent with other .NET Framework 4.0 applications.

This includes:

- Using classes marked-up with `System.ComponentMode.Composition` attributes as Unity components
- Using different catalog types for discovery – e.g. find all components in a directory of assemblies using `DirectoryCatalog`

3 Composition Primitives

The Composition Primitives are .NET types found primarily in the `System.ComponentModel.Composition.Primitives` namespace.

The role of the Composition Primitives is to specify components that can be 'wired' together to create useful software.

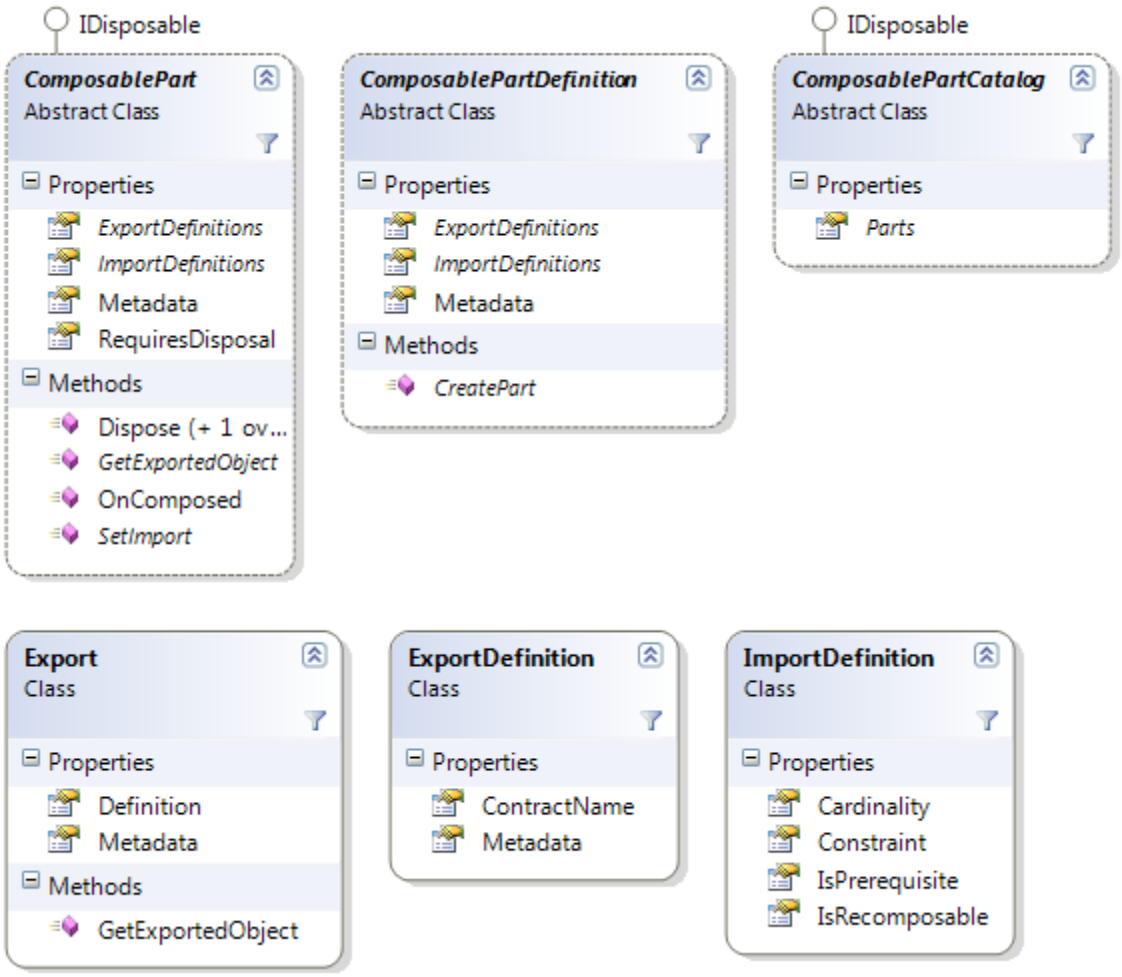

Figure 2 – Composition Primitives Classes

3.1 ComposablePart

The central type in the Composition Primitives is `ComposablePart`. A `ComposablePart` is a live, executing software component instance.

`ComposableParts` provide their capabilities to other components as *exports* and consume the capabilities of other components through *imports*.

Each `ComposablePart` describes its own imports and exports in its `ImportDefinitions` and `ExportDefinitions` collections.

The actual objects represented by the *imports* and *exports* can be exchanged with the `ComposablePart` using its `SetImport` and `GetExportedObject` methods.

3.2 ExportDefinition

An `ExportDefinition` is a structure comprising a string-based `ContractName` and a dictionary of additional information called `Metadata`.

Each ExportDefinition attached to a ComposablePart describes an individual capability of the part. For example, a part providing a parser for both C# and VisualBasic.NET source code would probably represent this through two separate ExportDefinitions. The ContractName would most likely be "parser" in both cases, but each would use metadata to further elaborate on the programming language able to be parsed.

3.3 ImportDefinition

ImportDefinition describes a dependency.

It includes a function called Constraint that selects ExportDefinitions based on the requirements of the *import*. For example, an *import* constraint for a C# parser component might look like:

```
(ExportDefinition ed) =>

        ed.ContractName = "parser" &&

        ed.Metadata["language"] == "C#"
```

Because Constraint is a Linq expression that can be examined at runtime, it is not always necessary for a host to execute the expression in order to work out which exports should be provided. This scenario is discussed below in relation to the ContractBasedImportDefinition type.

Linq is an unambiguous language for specifying arbitrarily complex dependency requirements. It excels in this role because requirements to be precisely tested through constraint execution when necessary.

3.4 ComposablePartDefinition

The kinds of ComposablePart that can be created in a given system are described using ComposablePartDefinitions.

A ComposablePartDefinition is a factory for one kind of ComposablePart. New instances of a ComposablePart can be created using the CreatePart method:

```
ComposablePartDefinition consoleLoggerDefinition = …

var consoleLogger = consoleLoggerDefinition.CreatePart();
```

ComposablePartDefinitions describe the ComposableParts that they create through the ImportDefinitions, ExportDefinitions and Metadata properties. These will match the same properties on the ComposableParts that the definition can create.

3.5 ComposablePartCatalog

ComposablePartDefinitions are grouped into ComposablePartCatalogs, which can optimize the selection of appropriate definitions through the IQueryable<ComposablePartDefinition> Parts property.

4 The Role of Programming Models

The Composition Primitives can be used simplistically by subclassing the various classes described above. The user experience in this direct mode of use is cumbersome; hence the way developers interact with the primitives is usually via a programming model.

Programming models are concerned with letting developers create `ComposablePartDefinitions` in a way that maps closely to their domain-specific goals.

Most programming models optimize the task of mapping a programming language 'type' (e.g. a C# or Ruby class) to a `ComposablePartDefinition` with exports and imports.

5 Attributed Programming Model

The Attributed Programming Model uses attributes to map the interfaces and members of regular .NET class definitions to instances of `ComposablePartDefinition` and its related classes.

5.1 Attributed Definitions

A simple attributed definition might look like:

```
[Export(typeof(IShape))]
public class Square : IShape {
	public int NumberOfSides { get { return 5; } }
}
```

When this type is read by the Attributed Programming Model it will be converted into a `ComposablePartDefinition` with a single `ExportDefinition` in its `ExportDefinitions` collection.

This `ExportDefinition` will have a `ContractName` equivalent to `CompositionServices.GetContractName(typeof(IShape))`.

When `CreatePart` is executed on this `ComposablePartDefinition`, an underlying instance of `Square` will (logically) be created.

When `GetExportedObject` is used to obtain the `IShape` export, the `Square` instance will be returned.

Attributes like `Export` exist in the Attributed Programming Model for declaring a full range of exports, imports and metadata items.

5.2 Consuming the Attributed Programming Model

The primary interfaces to the Attributed Programming Model are:

- `TypeCatalog`
- `AssemblyCatalog`

- `DirectoryCatalog`

When one of the above catalog types is instantiated in such a way that the attributed definitions are discovered, these will appear as `ComposablePartDefinitions` in the `ComposablePartCatalog.Parts` property.

The attributes and the types that expose them are completely encapsulated in the `ComposablePartDefinition` instances that the catalogs provide. Attempting to map between `Type` and `ComposablePartDefinition` is strongly discouraged.

6 High-Level Integration Points

Hosts should interact with the Composition Primitives via the `ComposablePartCatalog` type.

Identifying how catalogs will be provided to the host is the first issue to address.

Anticipate that users of a hosting framework will generally wish to supply their own catalogs – this will allow them to use the Attributed Programming Model in addition to alternative programming models.

Even if the host is an application rather than a framework, favor interaction with the general types that are accessible from `ComposablePartCatalog`.

Unity Example

The code snippet below shows Unity being configured to provide parts from a TypeCatalog, and then the consumption of this part through the Unity interface:

```
IUnityContainer container = …

container.AddNewExtension<CompositionPrimitivesIntegration>();

var catalog = new TypeCatalog(typeof(TestPart))

container.Configure<CompositionPrimitivesIntegration>()

    .RegisterCatalog(catalog);

var part = container.Resolve<ITestPart>();
```

7 Exchanging Exports

`ComposableParts` express dependencies as *queries* that select `ExportDefinitions`. These queries are predicates attached to `ImportDefinitions`.

Hosts must satisfy imports using values that match the `ImportDefinition`'s predicate.

The values can be the exports of other parts, or host-provided services. In either case, the host must have an understanding of what is represented by an `ImportDefinition`.

7.1 Executing a Constraint

In closed or simple systems, it may be possible to create an `ExportDefinition` representing every available `Export` in the system. (An ability to map the `ExportDefinition` to the corresponding `Export` is assumed.)

In these cases, the following code would select correct exports:

```
var allExportDefs = …

var importDef = …

var constraint = importDef.Constraint.Compile();

var matchingExportDefs = allExportDefs.Where(ed => constraint(ed));
```

7.2 ContractBasedImportDefinition

Many imports are simple queries based on three aspects of the ExportDefinition:

- `Equality of ContractName`
- `Equality of RequiredTypeIdentity`
- `Subset of Metadata keys`

These import definitions can be expressed as a `ContractBasedImportDefinition`. Hosts aware of this type can test import definitions and cast to the more specific type in order to gain access to these common data items:

```
var cbid = importDef as ContractBasedImportDefinition;

if (cbid != null)

{

	Console.Write(“ContractName={0}”, cbid.ContractName);

	Console.Write(“RequiredTypeIdentity={0}”, cbid.RequiredTypeIdentity);

	foreach (var requiredKey in cbid.RequiredMetadata)

		Console.WriteLine(“Requires {0}”, requiredKey);
```

7.2.1 ContractName

`ContractNames` are unique string values that describe the role of an exported or imported item.

The `ContractName` value in a `ContractBasedImportDefinition` must be matched using string equality with the `ContractName` property on the `ExportDefinition` being tested.

An example value may be “MyCompany-AppName-ToolBarItem”.

ContractName alone does not indicate compatibility between an exporter and an importer – additional requirements of the importer may be expressed in the RequiredTypeIdentity and RequiredMetadata values.

7.2.2 RequiredTypeIdentity

RequiredTypeIdentity represents a requirement of the importer that the exported value will be able to be cast to a specific type.

The constraint of a ContractbasedImportDefinition will match the RequiredTypeIdentity value with the Metadata[RequiredTypeIdentityMetadataName] string value on the ExportDefinition.

RequiredTypeIdentity is generated using the CompositionServices.GetTypeIdentity method. This value alone will not be sufficient to load a System.Type instance for the intended type. If mapping to a Type is required either:

- scan known types to create a lookup from 'type identity' to Type; or,
- use the ImportDefinition.Metadata[ImportTypeAssemblyQualifiedName] value as a hint.

7.2.3 RequiredMetadata

The string values here specify the keys that must appear in the ExportDefinition.Metadata dictionary.

The meaning of such keys is specific to the importer.

7.2.4 Special Cases when interchanging with Types and Names

Many host environments locate services using a key base on Type, name, or a combination of the two.

When ContractName and RequiredTypeIdentity are the same string value, this can be regarded as a 'default' contract name: implementations can infer that type only is significant. Hosts that allow a 'null' name selector may regard this as the equivalent in the Composition Primitives.

Unity Example

The import definition containing:

```
ContractName = "MyNamespace.IMyInterface"

ExportedTypeIdentity = "MyNamespace.IMyInterface"
```

… is transformed by the Unity integration into a query equivalent to:

```
IUnityContainer container = …

var result = container.Resolve<MyNamespace.IMyInterface>();
```

When RequiredTypeIdentity is null, only ContractName is relevant. Hosts can return implementations of any type that is allowed by the documentation for the particular ContractName.

Unity Example

Unity does not generally support the concept of named instances without an associated type. A basic mapping is made so that such requests assume a registration for `System.Object`.

The import definition containing:

```
ContractName = "Sender.Timeout"

ExportedTypeIdentity = null
```

… is transformed by the Unity integration into a query equivalent to:

```
IUnityContainer container = …

var result = container.Resolve<object>("Sender.Timeout");
```

7.3 Parsing or Transforming Constraints

A third option for selecting exports may apply to systems with existing query mechanisms: Constraints can be transformed to match the query format of a host system.

Most systems do not require this capability and can make use of `ContractBasedImportDefinition`.

7.4 Using Host-Provided Services

In order to supply `ComposableParts` with import values that do not come from other parts, a simple system might simply create `ExportDefinition` instances representing all of the services that the host provides, then evaluate the `ImportDefinition.Constraint` predicate against each of these in turn.

More sophisticated hosts, or those that operate with an open set of available services, will generally use a two-step process:

1. Transform the `ImportDefinition` into the query parameters used internally by the host (e.g. Type and name)
2. Construct `ExportDefinition` instances for all results and supply these to the part

8 Standardizing Part Manipulation rather than Object Graph Wiring

The premise of this document is that individual parts must be able to make certain assumptions about the validity of their environment.

It is *not* necessary that any given set of components will form the same object graph under any host.

The latter 'limitation' can exist because:

- the Composition Primitives are for use in open systems, therefore an author can never assume that a part will receive any particular configuration of dependencies

- extensions use a common set of APIs (e.g. the Attributed Programming Model) but target a specific host environment, with which they are tested

9 Required Host Behavior

Below is a notable subset of the requirements that a host must fulfill in order to successfully host the .NET Composition Primitives.

9.1 Compliance with ImportDefinitions

`ImportDefinition.Constraint` is a predicate on `ExportDefinition`. The following code (in `ComposablePart`) should never throw an exception:

```
public void SetImports(ImportDefinition import, IEnumerable<Export> values) {
    var predicate = import.Constraint.Compile();
    foreach (Export e in values)
        if (!predicate(e.Defintion))
            throw new ArgumentException(
                "Export does not match constraint.");
```

9.2 ComposablePart Lifecycle

The lifecycle for a ComposablePart must adhere to the following sequence:

1. Creation, by calling `ComposablePartDefinition.CreatePart` or by constructing a concrete implementation with `new`
2. Satisfaction of prerequisite dependencies by calling `SetImports`
3. Access to exported objects by calling `GetExportedObject` is now allowed, but the objects themselves cannot be used until the part is activated
4. Satisfaction of non-prerequisite dependencies by calling `SetImports`
5. Completion of the activation process by calling `OnComposed`
6. Exported objects from the part can now be used
7. Disposal with `IDisposable.Dispose`

Disposal is a valid action at any point in the process should it be necessary.

9.3 Full Composition of Prerequisite Dependencies

If an import is a prerequisite, then any values provided for that import must be fully-composed. This is a transitive requirement, so the dependencies of any dependencies must be fully-composed and so on.

9.4 Required Creation Policy

Any `ImportDefinition` that has a `RequiredCreationPolicy` value of `CreationPolicy.NonShared` must be supplied with a unique instance. During the lifetime of the ComposablePart no other caller may interact with the supplied instance.

9.5 Part Creation Policy

If a `ComposablePartDefintion` S has creation policy `CreationPolicy.Shared` then:

- no part instance p that can reach an instance s of S should be able to reach another instance s' of S
- no part p reachable from s should be able to reach another instance s' of S

9.6 Recomposition

If an import definition is not marked recomposable (`IsRecomposable == true`) then `SetImports` may only be called for that import until the first time `OnComposed` is called.

10 Host-Defined Behavior

Below is a list of behavioral features of the `CompositionContainer` type. These are regarded as 'host' responsibilities and do not need to be implemented consistently by other hosts of the Composition Primitives.

10.1 Lazy Deep Object Graph Instantiation

Although the interface to `ComposablePart` enables lazy instantiation of deep object graphs, it is not a requirement that all dependencies supplied to a part are lazily instantiated.

While performance characteristics may differ, an implementation that eagerly instantiates dependencies is still considered correct.

10.2 Circular Dependency Detection

Hosts are not required to enable circular graphs of any kind.

It is recommended that hosts perform circular dependency detection in order to simplify the diagnostic experience.

10.3 Rejection

The Export.GetExportedObject method through which parts access imports should always return a value or throw an exception. Exceptions in such circumstances are rarely recoverable.

Hosts should strive not to provide exports to a part that cannot be accessed. If the host supports lazy composition, maintaining this behavior requires deep dependency analysis. Hosts that cannot do deep analysis to determine whether an export can be satisfied should favor eager composition.

10.4 Defaults and Cardinality

When an import specifies that exactly one instance is required, the host may use its own algorithm to determine what (if anything) will be selected from the available instances that may satisfy the dependency.