Framework

Grundlagen

Die erste Frage, die sich bei einem Framework stellt, ist, wie eine Extension in das Framework geladen wird. Microsoft bietet für diesen Zweck das Managed Extensibility Framework[[1]](#footnote-1) (MEF) an. Technische Details dazu können unter <http://mef.codeplex.com/documentation> nachgelesen werden.

Die wichtigste Funktionalität von MEF, die für die HSR Video Wall gebraucht wird, ist die Folgende:

* Extension exportiert eine Klasse (Einstiegspunkt), die von einem vom Framework (HSR Video Wall Applikation) definierten Interface (IApp) ableitet.
* Das Framework (HSR Video Wall Applikation) importiert alle Klassen, die an einem bestimmten Pfad liegen (Extensions Ordner) und das Interface IApp implementieren.



Abbildung 1 - Poster Applikation (Extension) wird über [Export(typeof(IApp))] als IApp exportiert



Abbildung 2 - Die Video Wall Applikation (Framework) importiert über [ImportMany(AllowRecomposition = false)] alle Klassen, die das Interface IApp implementieren

Problem

Beim Entwickeln eines Frameworks ist es oftmals nicht vorhersehbar, wie dieses in der Zukunft aussehen wird, da sich die Anforderungen an das Framework stetig ändern. Würde man also ein einziges Interface (IApp) zur Verfügung stellen, über das die Services des Frameworks angesprochen werden können, so müsste sich dieses Interface ständig ändern und die Extensions müssten nach jeder Änderung am Interface (IApp) neu kompiliert werden. Somit wird es schwierig, an der Extension neue Funktionalität zur Verfügung zu stellen. Ein weiteres Problem ist, dass dieses Interface beliebig gross werden kann und dadurch die Kopplung steigt und die Kohäsion sinkt, was sehr unschön ist.

Das ursprüngliche IApp Interface sah folgendermassen aus:



Wie am Interface erkennbar ist, ist es relativ gross und stellt verschiedene Services zur Verfügung, die nichts miteinander zu tun haben (bps. ResourceDirectory Property, in dem die Extension Dateien zur Verfügung stellen kann oder der SkeletonChangedEvent, der vom Framework aufgerufen werden soll, sobald sich das Skelett verändert).

Da sich diese Interface sehr wahrscheinlich auch ändern wird, zum Beispiel durch Hinzufügen neuer Funktionalität), müssen dann auch immer alle Extension neu kompiliert werden. Dies ist natürlich suboptimal, speziell dann, wenn die Extensions von verschiedenen Personen gewartet werden muss.

Lösung: Angelehnt an das Extension Interface[[2]](#footnote-2)

Das Extension Interface[[3]](#footnote-3) bietet einen Ansatz, das Problem des ständig ändernden Interfaces zu lösen. In der Video Wall Applikation wurde dies abgeändert angewendet, speziell ohne die Vererbung des Root Interfaces. Zusätzlich wurde Unity[[4]](#footnote-4) verwendet, um die Factory aus dem Extension Interface zu ersetzen. Die Implementation sieht folgendermassen aus:



Abbildung 3 - Das IApp Interface

Das IApp Interface bietet einen Einstiegspunkt. Da jede Applikation dieses Interface implementieren müssen, sind hier nur die Anforderungen beschrieben, die jede Applikation anbieten muss. Speziell ist die Methode Activate, die auf jeder ladenden Extension vom Framework genau einmal aufgerufen wird. In dieser Methode kann die Extension über das IVideoWallServiceProvider Objekt weitere Services anfordern:

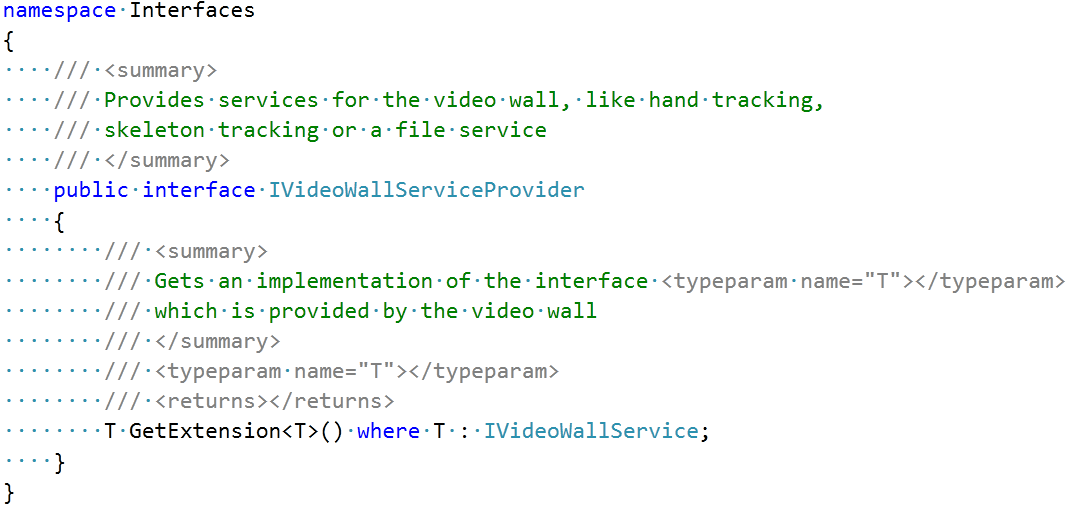


Abbildung 4 - Durch den IVideoWallServiceProvider können weitere Extensions geladen werden

Über die Methode GetExtension können, wie bereits oben beschrieben, weitere Services (IVideoWallService) angefordert werden. Das IVideoWallService Interface ist ein Marker Interface. Es ist nicht vorgesehen, dass Applikationen weitere Extensions registrieren können, was auch der Hauptunterschied zum Extension Interface Pattern darstellt.

1. <http://mef.codeplex.com/> [↑](#footnote-ref-1)
2. <http://wiki.hsr.ch/APF/files/ExtensionInterface.pdf> [↑](#footnote-ref-2)
3. <http://wiki.hsr.ch/APF/files/ExtensionInterface.pdf> [↑](#footnote-ref-3)
4. <http://unity.codeplex.com/> [↑](#footnote-ref-4)