*Copyright © 2008 Microsoft Corporation. All rights reserved.*

Global properties setting and synchronization on MPF

Primary use case scenario:

Our nested project system communicates with the parent project with the so called MSbuild task. In order that the MSbuild task to execute successfully the global properties between the nested and the parent project have to match.

Example of an MSbuild task from our nested projects targets file where the ParentProject is our modeling project system:  
<MSBuild Projects="$(ParentProject)" Targets="GetCompileModelItems" Condition="'$(IsParentProjectAvailable)'=='true'">  
 <Output ItemName="CompileModel" TaskParameter="TargetOutputs"/>  
</MSBuild>

In order to be able to communicate with a project system relying on MPF using the MSBuild task the global property handling and synchronization of those global properties that can vary has to be solved.

Currently in the language project system the following global properties are defined.

/// <summary>

/// Defines the global propeties used by the msbuild project.

/// </summary>

public enum **GlobalProperty**

{

/// <summary>

/// Property specifying that we are building inside VS.

/// </summary>

BuildingInsideVisualStudio,

/// <summary>

/// The VS installation directory. This is the same as the $(DevEnvDir) macro.

/// </summary>

DevEnvDir,

/// <summary>

/// The name of the solution the project is created. This is the same as the $(SolutionName) macro.

/// </summary>

SolutionName,

/// <summary>

/// The file name of the solution. This is the same as $(SolutionFileName) macro.

/// </summary>

SolutionFileName,

/// <summary>

/// The full path of the solution. This is the same as the $(SolutionPath) macro.

/// </summary>

SolutionPath,

/// <summary>

/// The directory of the solution. This is the same as the $(SolutionDir) macro.

/// </summary>

SolutionDir,

/// <summary>

/// The extension of teh directory. This is the same as the $(SolutionExt) macro.

/// </summary>

SolutionExt,

/// <summary>

/// The fxcop installation directory.

/// </summary>

FxCopDir,

/// <summary>

/// The ResolvedNonMSBuildProjectOutputs msbuild property

/// </summary>

VSIDEResolvedNonMSBuildProjectOutputs,

/// <summary>

/// The Configuartion property.

/// </summary>

Configuration,

/// <summary>

/// The platform property.

/// </summary>

Platform,

}

Out of these most are static in relationship with a project, thus need to be defined only once, but the Platform and Configuration can change? Therefore the Platform and the Configuration changes have to be synchronized in case of a change to a project’s Active configuration.

The language project systems are also setting and synchronizing the global properties.

Primary entry points in the Language project system:

1. Loading the global properties is done in CXMakeHelper::LoadMSBuildProjectFileInternal in vsproject\xmlparser\XMakeHelper.cpp
2. Synchronizing global properties besides many other things is done in the method CLangProjectConfigs::ChangeConfiguration(BSTR bstrName) in vsproject\langbuild\langprojcfg.cpp called by the CVsProject::OnActiveProjectCfgChange in vsproject\vsproject\vbprject.cpp

The idea around the design in MPF is that:

1. Global properties are set before the project is loaded in the project factory.
2. A project node should be able to handle synchronization of configuration properties on its own way. For example in the Modeling Project System the modeling project would check and pop up a dialog box warning the user of possible abnormality if the nested project’s configuration differs from the parent one. The default behavior on the other hand on the ProjectNode is to set the configuration and platform changes in the global properties.

An alternative idea would be to let the ProjectConfig object handle configuration changes like in the case of the language projects. The problem is that in case of the language projects many other things are done in the configuration change listener and the main semantics there is to handle project to project references if the configuration changes.

Also why would one need to subclass and provide its own ProjectConfig just if it wants to handle global property changes differently than the base?

It could very well be that MPF would need a configuration change listener on the project configuration too but its primary purpose then it would be to handle the same main scenarios for example like dependency change in all the projects that have this project as a Project ref as the language project is handling.

The following describes the static class structure:



As one can see there is a central GlobalPropertyHandler object that can set the initial global properties and can also listen to IVsUpdateSolutionEvents2 interface through the composed nested UpdateConfigProviderListener type. It maybe weird that I have used composition through nesting, instead of sub classing from the UpdateSolutionEvenstListener but I could not sense an “is a” relationship with the abstract UpdateSolutionEvenstListener . Also I did not want to re-implement the IVsUpdateSolutionEvents2 interface.  
This GlobalPropertyHandler object will delegate all its configuration change event sink based events to the DotNet based ActiveConfigurationChanged event. The ProjectNode will register with this event and delegate it to the virtual OnHandleConfigurationChangeProperties.

The following describes a sequence diagram pointing out two things:

1. How Global properties are loaded
2. How a ProjectNode gets the ActiveConfigurationChanged event

