Partitioning code base through .NET assemblies and Visual Studio projects

NDepend White-Book

v1.0 May 2011

Should every Visual Studio project really be in its own assembly? And what does 'Copy Local=True' really mean? The NDepend team is no stranger to large .NET projects, and is well placed to lay a few myths to rest, and gives some advice that promises up to a tenfold increase in speed of compilation.

This white book is aimed at

- Providing a list of DO and DON'T when it comes to partitioning a code base into .NET assemblies and Visual Studio projects.
- Shedding light on.NET code componentization and packaging.
- Suggesting ways of organizing the development environment more effectively.

The aim of this is to increase the speed of .NET developer tools, including VS and C#/VB.NET compilers, by up to an order of magnitude. This is done merely by rationalizing the development of a large code base. This will significantly increase productivity and decrease the maintenance cost of the .NET application.

This advice is gained from years of real-world consulting and development work and has proved to be effective in several settings and on many occasions.

Why create another .NET assembly? 2

- Common valid reasons to create an assembly 2
- Common invalid reasons to create an assembly 2
- Merging assemblies 3
- Reducing the number of assemblies 3

Increase Visual Studio solution compilation performance 4

Organize the development environment 6

- Organization of Assemblies 6
- Test assemblies organization 8
- VS Solutions files and Common build actions 8

Guidelines 8

Why create another .NET assembly?

The design of Visual Studio .NET (VS) has always encouraged the idea that there is a one-to-one correspondence between assemblies and VS projects. It is easy to assume from using the Visual Studio IDE that VS projects are components of your application; and that you can create projects at whim, since by default, VS proposes to take care of the management of project dependency.

Assemblies, *.exe* and *.dll* files, are physical units. They are units of deployment. By contrast, a component is better understood as logical unit of development and testing. A component is therefore a finer-grained concept than an assembly: An assembly will typically contain several components.

Today, most .NET development teams end up having hundreds, and possibly thousands, of VS projects. The task of maintaining a one-to-one relationship between assembly and component will have these consequences

- Developers' tools will slow down by up to an order of magnitude. The whole stack of .NET tooling infrastructure, including VS and the C# and VB.NET compilers, work much faster with fewer larger assemblies than it does with many smaller assemblies.
- Deployment packaging will become a complex task and therefore more error-prone.
- Installation time and application start-up time will increase because of the overhead cost per assembly.
- In the case of an API whose public surface is spread across several assemblies, there will be latency because of the burden for client API consumers to figure out which assemblies to reference.

All these common .NET development problems are a consequence of the usage of a physical object, an assembly, to implement a logical concept, a component. So, if we shouldn't automatically create a new assembly for each component, what are the good reasons to create an assembly? What common practices don't constitute good reasons to do so?

Common valid reasons to create an assembly

- **Tier separation**, if there is a requirement to run some different pieces of code in different AppDomains, different processes, or different machines. The idea is to avoid overwhelming the precious Window process memory with large pieces of code that are not needed. In this case, an assembly is especially created to contain shared interfaces used for communication across tiers.
- **AddIn/PlugIn model**, if there is a need for a physical separation of interface/factory/implementation . As in the case of Tier Separation, an assembly is often dedicated to contain shared interfaces used for communication across the plugin and its host environment.
- **Potential for loading large pieces of code on-demand:** This is an optimization made by the CLR: assemblies are loaded on-demand. In other words, the CLR loads an assembly only when a type or a resource contained in it is needed for the first time. Because of this, you don't want to overwhelm your Window process memory with large amounts of code that are seldom if ever required.
- **Framework features separation:** With very large frameworks, users shouldn't be forced to embed every feature into their deployment package. For example, most of the time an ASP.NET process doesn't do some Window Forms and vice-versa, hence the need for the two assemblies *System.Web.dll* and *System.Window.Forms.dll*. This is valid only for large frameworks with assemblies sized in MB.
  A quote from Jeremy Miller, renowned .NET developer, explains this perfectly:
  Nothing is more irritating to me than using 3rd party toolkits that force you to reference a dozen different assemblies just to do one simple thing.
- **Large pieces of code, that don't often evolve** (often automatically generated code) can become a drain on developer productivity if they are continuously handled in the developer environment. It is better to isolate them in a dedicated assembly within a dedicated VS solution that only rarely needs to be opened and compiled on the developer workstation.
- **Test/application code separation**. If only the assemblies are released rather than the source code, it is likely that tests should be nested in one or several dedicated test assemblies.

Common invalid reasons to create an assembly

- **Assembly as unit of development, or as a unit of test.** Modern Source Control Systems make it easy for several developers to work simultaneously on the same assembly (i.e the same Visual Studio project). The unit should, in this case, be the source file. One might think that, by having fewer and bigger VS projects, you'd increase the contentions on sharing VS .sln, .csproj and .vbproj files. But as usual, the best practice is to keep these files checked-out just for the few minutes required to tweak project properties or add new empty sources files.

- **Automatic detection of dependency cycles between assemblies by MSBuild and Visual Studio**. It is important to avoiding dependency cycles between components, but you can still do this if your components are not assemblies but sub-set of assemblies. There are tools such as NDepend which can detect dependency cycles between components within assemblies.
- **Usage of internal visibility to hide implementations details**. The public/internal visibility level is useful when developing a framework where it is necessary to hide the implementation details from the rest of the world. Your team is not the rest of the world, so you don't need to create some assemblies especially to hide some implementations details to them.
  In order to prevent usage and restrict visibility of some implementation details, a common practice is to define some sub-namespaces named *Impl*, and use tooling like NDepend or others to restrict usage of the *Impl* sub-namespaces.

Merging assemblies

There are two different ways to merge the contents of several assemblies into a single one.

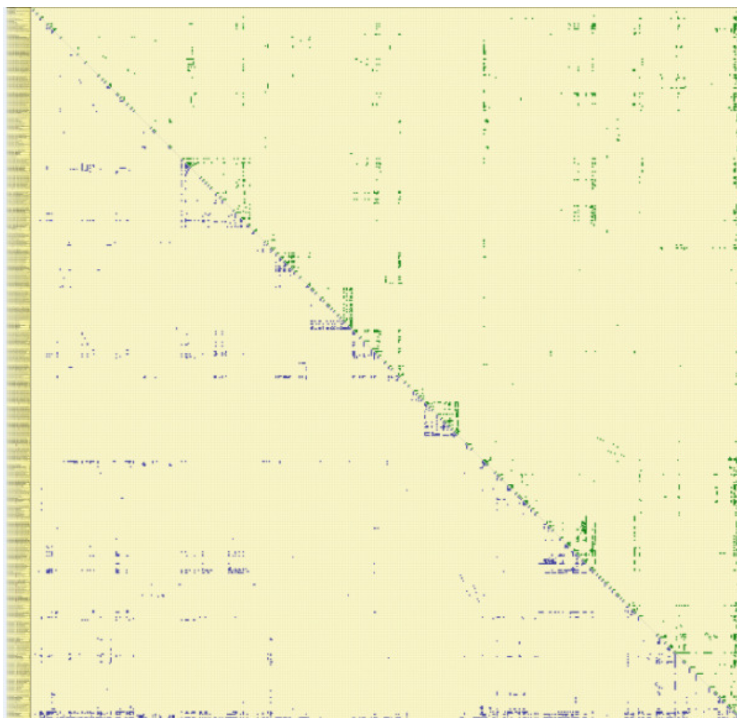
- Usage of the *ILMerge* tool to merge several assemblies into a single one. With *ILMerge*, merged assemblies are losing their identity (name, version, culture, and public key).
- Embedding several assemblies as resources in a single assemblies, and use the event `AppDomain.CurrentDomain.AssemblyResolve` to extract assemblies at runtime. The difference with *ILMerge* is that all assemblies keep their identity.

This doesn't solve those problems that are due to there being too many VS projects, thereby causing a significant slowdown of VS, and the compiler's execution time.

Reducing the number of assemblies

Technically speaking, the task of merging the source code of several assemblies into one is a relatively light one that takes just a few hours. The tricky part is to define the proper new partition of code across assemblies. At that point you'll certainly notice that there are groups of assemblies with a high cohesion. These, are certainly candidates to be merged together into a single assembly.

By looking at assemblies dependencies with a Dependency Structure Matrix (DSM) such as the one of NDepend, these groups of cohesive assemblies form obvious squared patterns around the matrix diagonal. Here is a DSM taken on more than 700 assemblies within a real-world application:

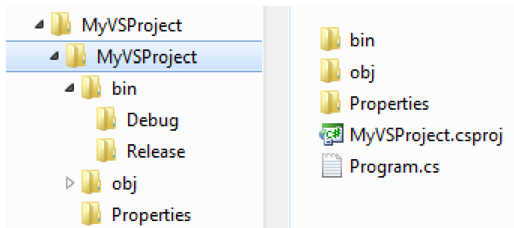


Increase Visual Studio solution compilation performance

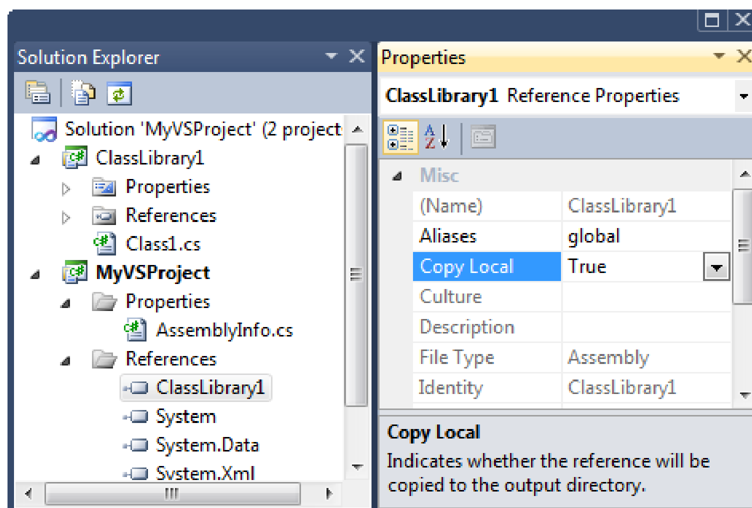
You can use a simple technique to reduce the compilation-time of most real-world VS solutions by up to an order of magnitude, especially when you have already merged several VS projects into a few.

On a modern machine, the optimal performance of the C# and VB.NET compiler is about of 20K logical Lines of Code per second, so you can measure the room for improvement ahead. A **logical Lines of Code (Loc)** represents a Sequence Point. A sequence point is the code excerpt highlighted in dark red in the VS code editor window, when creating a breakpoint. Most of .NET tools for developers, including VS and NDepend measure Lines of Code through sequence points.

By default, VS stores each VS project in its own directory. Typically VS suggests the folder hierarchy for a project, named here *MyVSProject*:

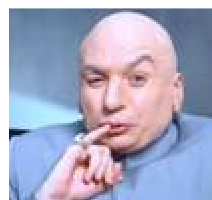


A VS solution typically has several VS projects and, by default, each VS project lives in its own directory hierarchy. At compilation time, each project builds its assembly in its own *bin\Debug* or *bin\Release* directory. By default, when a project A references a project B, the project B is compiled before A. However, the assembly B is then duplicated in the *bin\Debug* or *bin\Release* directory of A. This duplication action is the consequence of the value '*True'* having been set by default for the option '*Copy Local*' of an assembly reference. Whereas it makes sense for a small solution, it will soon cause problems for larger applications



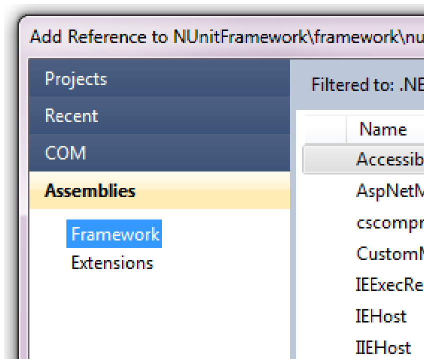
As the size and complexity of solutions increase, the practice of duplicating assembly at compilation time is extremely costly in terms of performance. In other words:

Copy Local = true is evil

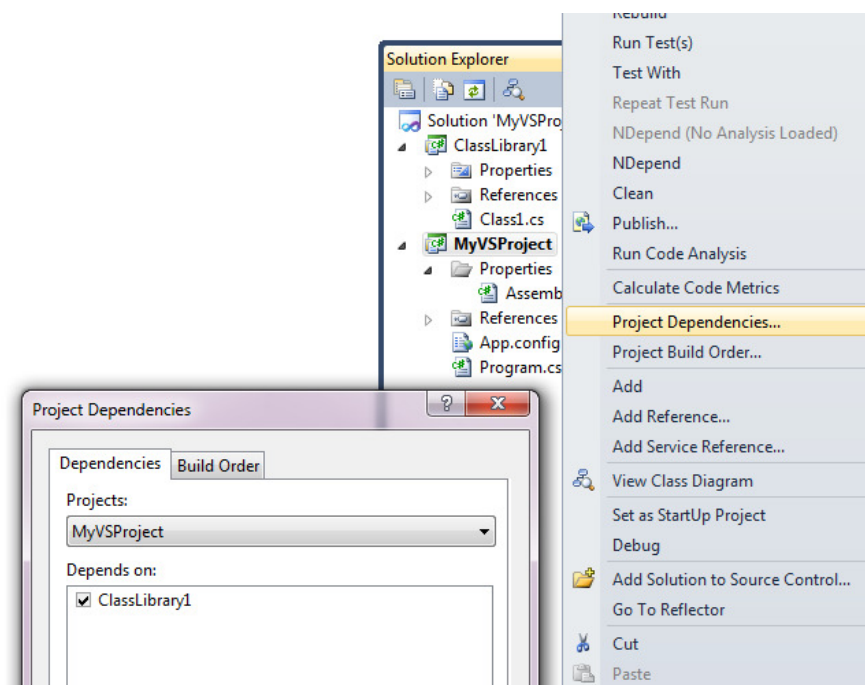


Imagine a VS solution with 50 projects. Imagine also that there is a core project used by the 49 others projects. At *Rebuild-All* time, the core assembly will be compiled first, and then duplicated 49 times. Not only this is a huge waste of disk, but also of time. Indeed, the C# and VB.NET compilers don't seem to have any checksum and caching algorithm to pinpoint whether the metadata of an assembly has already been parsed. As a result, the core assembly has its metadata parsed 49 times, and this takes a lot of time, and can actually consume most of the compilation resources.

From now on, when adding a reference to a VS project, make sure first, to add an assembly reference, and second, make sure that *Copy Local* is set to *False* (which doesn't seem to be always the case)



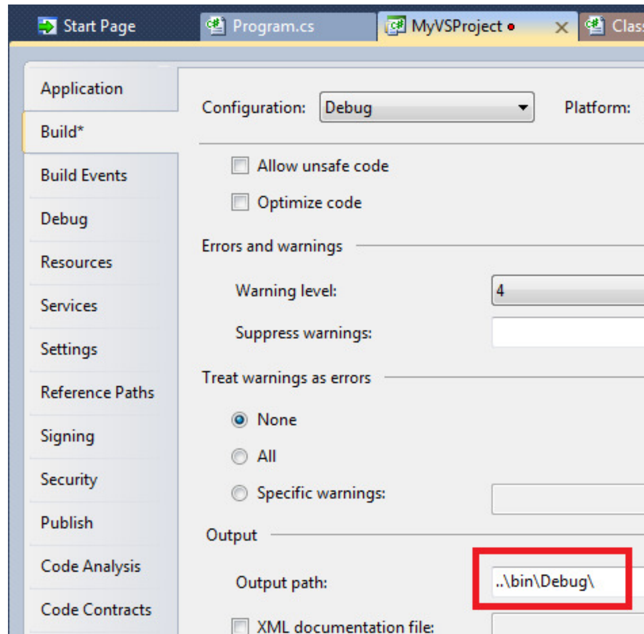
A slight drawback to referencing directly assemblies directly rather than their corresponding VS projects, is that it is now your responsibility to define the build-order of VS projects. This can be achieved through the *Project Dependencies* panel:



Organize the development environment

When *Copy Local* is set to *true,* the top level assemblies, typically executable assemblies, automatically end up with the whole set of assemblies that they used being duplicated in their own *.\bin\Debug* directory. When the user starts an executable assembly, it just works. There is no *FileNotFoundException* since all the assemblies that are needed are in the same directory.

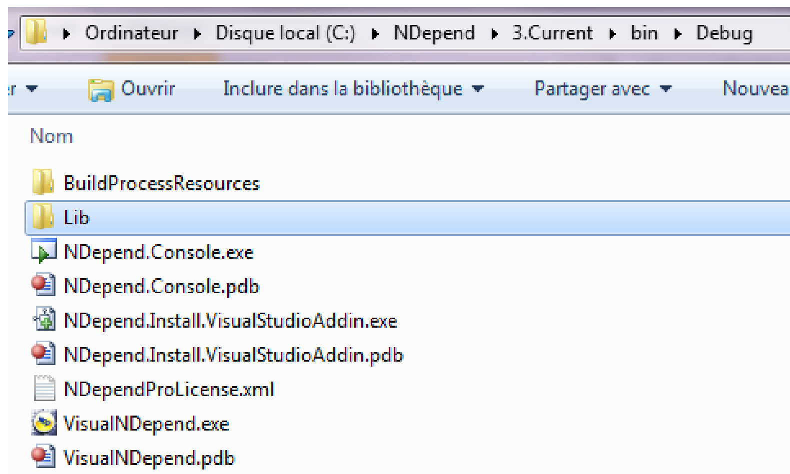
If you set '*Copy Local = false',* VS will, unless you tell it otherwise, place each assembly alone in its own *.\bin\Debug* directory. Because of this, you will need to configure VS to place assemblies together in the same directory. To do so, for each VS project, go to *VS > Project Properties > Build tab > Output path*, and set the *Ouput path* to *..\bin\Debug* for debug configuration, and *..\bin\Release* for release configuration.



Now that all assemblies of the solution reside in the same directory, there is no duplication and VS works and compiles much faster.

Organization of Assemblies

If there are many library assemblies and just a few executable assemblies, it might be useful to display only executable ones in the output directory *..\bin\Debug* (and in the *..\bin\Release* one as well). Library assemblies are then stored in a dedicated sub directory *..\bin\Debug\Lib* (and *..\bin\Release\Lib*)*.* This way, when the users browse the directory, they only see the executables without the dll assemblies and so can start any executable straight away. This is the strategy we adopted for the three NDepend executable assemblies:



If you wish to nest libraries in a sub-lib directory, it is necessary tell the CLR how to locate, at run-time, library assemblies in the sub-directory *.\lib*. For that you can use the `AppDomain.CurrentDomain.AssemblyResolve` event, this way:

```
class Program {

   internal static Assembly AssemblyResolveHandler(object sender, ResolveEventArgs args) {
      string libPath = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location) +
               Path.DirectorySeparatorChar +
               "Lib" +
               Path.DirectorySeparatorChar;
      var assembly = Assembly.LoadFrom(libPath + args.Name + ".dll");
      return assembly;
   }

   static void Main(string[] args) {
      AppDomain.CurrentDomain.AssemblyResolve += AssemblyResolveHandler;
      SubMain();
   }

   static void SubMain() {
      // ...
```

In this piece of code you will notice how we

- construct the path of the sub-directory, by relying to the properties: `Assembly.GetExecutingAssembly().Location`
- bind the event `AppDomain.CurrentDomain.AssemblyResolve` immediately in the *Main()* method, and then call a *SubMain()* method.
- need the *SubMain()* method, because if libraries types are called from the Main() method, the CLR tries to resolve libraries assemblies even before the method Main() is called, hence, even before the event `AppDomain.CurrentDomain.AssemblyResolve` is binded.

Instead of relying on the `AppDomain.CurrentDomain.AssemblyResolve` event, it is also possible to use an *executableAssembly.exe.config* file for each executable. Each file will redirect the CLR probing for assemblies to the sub-directory *.\Lib*. To do so, just add a *Application Configuration File* for each executable assembly, and put the following content in it:

```
<?xml version="1.0"?>
 <configuration>
   <runtime>
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <probing privatePath="Lib" />
     </assemblyBinding>
   </runtime>
 </configuration>
```

The solution involving the event `AppDomain.CurrentDomain.AssemblyResolve` is usually preferable because it avoids having to deploy the extra files in the redistributable.

Test assemblies organization

The other advantage to concentrating all assemblies of your application in the same*..\bin\Debug* directory is that test assemblies can be put under the *..\bin* directory. This way, at test execution time, tests assemblies are running the assemblies of the application directly under the *..\bin\Debug* directory. Therefore, it is not necessary to duplicate the application assemblies just for tests.

To do so, we suggest that you use the *Application Configuration File* trick we've just described, to redirect CLR probing to sub-directories of the *..\bin* directory. If you've put your NUnit assemblies (or equivalent) in a ..\bin\NUnit directory, the probing XML element of the *Application Configuration File* for each test assembly (that can be a library or an executable assembly) will look like.

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="Debug;Debug\Lib;NUnit;" />
</assemblyBinding>
```

The list of directories listed in the *privatePath* XML attribute can only be a sub-directory of the current directory. This is why the directory *..\bin* that contains all application assemblies in sub-directory, is well suited to contain test assemblies.

VS Solutions files and Common build actions

When the code base reaches a particular size, the user is forced to spread the code across several VS solutions. This is because, even with few VS projects and a solid hardware, VS slows down significantly when hosting a VS solution with more than 50K LoC.

We recommend that you put all the VS solutions files in the same root directory, the one that contains the *.\bin* folder described above. If each VS solution's files are stored in its own hierarchy of folders, it quickly becomes a considerable task to find them.

It saves a great deal of time if you put a few *.bat* files in this root folder that execute common build actions. This way, the developer is one-click away from:

- Rebuilding all in Debug mode
- Rebuilding all in Release mode
- Running all tests and Producing a report
- Running all tests with code coverage and Producing a report
- Rebuilding all in Debug mode, Running all tests and Producing a report

Rebuilding a solution file with a .bat file is as simple as writing in the .bat file:

```
%windir%\Microsoft.NET\Framework\v4.0\msbuild.exe MySolution.sln /p:Configuration=Debug
```

Guidelines

- Reduce drastically the number of assemblies of your code base.
- Create a new assembly only when this is justified by a specific requirement for physical separation.
- In a Visual Studio project, use 'reference by assembly' instead of 'reference by Visual Studio project'.
- Never use the Visual Studio referencing option 'Copy Local = True'.
- Put all VS solutions and Build action *.bat* files in a $rootDir$ directory.
- Compile all assemblies in directories: *$rootDir$\bin\Debug* and *$rootDir$\bin\Release*
- Use the directory *$rootDir$\bin* to host tests assemblies.