**Background (motivation part)**

***Postulate #1***: “*Tracing is helpful*”.

All we are using it from time to time. And me too.

To achieve this goal we use predefined IDE macros/classes, third-party libraries, own techniques, etc.. One of the main problems is that we need to do it manually in code.

This leads to ***Postulate #2***: “All that can be automated must be automated.”

***Postulate #3:*** “Sometimes quick look from afar helps to find problematic areas easier than digging in details”

**Goals**

1. Give user the ability to add trace points from IDE (inspired by ***Postulate #2***)
2. Interactive charts with traced data (inspired by ***Postulate #3***)
3. Minimize overhead

**Important notes (justification part #1)**

***Note #1***

It’s only the first developments stage (cycle) with many limitations. It is research stage. Some parts need to be refactored/changed, e.g. choice of COM exe server as transport, but I didn’t refuse to it's usage in order to understand this more clearly, ☺

***Note #2***

My knowledge of C# is far away from professional level. There were 2 reasons to use it:

* It’s easier to develop MSVS extension using it than to develop in native C++
* It was interesting to me

*hence the conclusion*: please be lenient.

***Note #3***

This project contains many stand-alone parts, which will not be fully covered in this article. I will make common architecture overview and then concentrate on most significant aspects. If this is not enough, please mention this in comments. If will be interested I will gather all questions and write “Part 2”.

***Note #4***

I’ll not describe how to create tool windows, context menu handler, e.g. This is fully described in many samples. First of all in **VSSDK-Extensibility-Samples** (<https://github.com/Microsoft/VSSDK-Extensibility-Samples>). I’ll concentrate on most difficult (IMHO) parts.

***Note #5***

My English is not as good as want I’d like. Sorry..

**Limitations (justification part #2)**

* Supported languages: native C++ only
* Supported IDE: MSVC 2015
* Tracing variables: class members of C++ fundamental types only + some typedefs like std::uint32\_t (because they are useful)

**Used technologies/languages**

* C++ (injected code, COM out-of-proc server)
* C# (MSVS 2015 extension, MSBuild task)

**Used architecture patterns**

Two main common architecture patterns are used in this project: *SOA (Service-Oriented Architecture) & EDA (Event-Driven Architecture)*. Both patterns are long known and well described, so I’ll focus only on details of current implementation.

***SOA***

*Declaration*: …

// base interface for all ChartPoints services (maybe extended in the future)

public interface ICPService

{

}

//public delegate void OnCPServiceCreated<T>(T args);

// singleton service provider

public abstract partial class ICPServiceProvider

{

public abstract bool RegisterService<T>(T obj) where T : ICPService;

public abstract bool GetService<T>(out T obj) where T : class;

//public abstract bool GetService<T>(out T obj, OnCPServiceCreated<T> cb) where T : class;

public static ICPServiceProvider GetProvider()

{

return impl.ICPServiceProvider.GetProviderImpl();

}

}

*Implementation*: …

Service provider stores registered services when calling *RegisterService* for future return by *GetService*. Second (commented) version of *GetService* will allow to provide callback which will be used if service is not registered yet. There is no reasons to implement it now so I left it commented.

This approach allows to manage the order of service creation & query services from any place without worrying about their availability.

***EDA***

*Declaration*: …

public delegate void OnCPEvent<T>( T args );

public abstract class ICPEvent<T>

{

protected abstract ICPEvent<T> Add(OnCPEvent<T> cb);

public static ICPEvent<T> operator +(ICPEvent<T> me, OnCPEvent<T> cb)

{

return me.Add(cb);

}

protected abstract ICPEvent<T> Sub(OnCPEvent<T> cb);

public static ICPEvent<T> operator -(ICPEvent<T> me, OnCPEvent<T> cb)

{

return me.Sub(cb);

}

public abstract void Fire(T args);

}

*Implementation*: …

*ICPEvent* provides usual +/- operators. In current implementation all events are stored (there are not so many of them so we don’t have to worry). When new client subscribes it receives all earlier generated events. If it will be required, simple specialization without history will be added. Current implementation provides all the needs.

It’s done for two reasons:

1. Some objects are initialized not in predefined order (e.g. from event handlers of MSVS) and this guarantees that all of them will be delivered.  
   *Example*: When tagger(object responsible for rendering glyphs in code editor) is created and subscribes to ChartPoints events it successfully receives all fired earlier and have all actual information for rendering glyphs.
2. It gives a possibility to add time marks to them for logging on demand (not implemented now but can easily added).

**Class Factory**

Basic goals of current class factory implementation:

1. Testing purposes. Allows to implement individual class factories based on any of existing ones and provide moqs & stubs
2. Strategy pattern via DI. Allows of implementing different strategies of same interface. As all other objects query it’s construction via factory methods, all that is need to do is to change appropriate class factory method
3. Hide the possibility of explicitly creating objects intended to be created via class factory (in same assembly too). It was hard to understand how to achieve this. In C++ it is done easily, but in C# redundant entities need to be added to emulate ‘friend’ keyword and \*.cpp file-part implementation. Maybe it can be done more easier..

***Declaration of class factory***

// class factory interface

public abstract partial class IClassFactory

{

// set custom class factory instance for DI purposes

public static void SetInstance(IClassFactory inst)

{

ClassFactory.SetInstanceImpl(inst);

}

// returns singleton instance

public static IClassFactory GetInstance()

{

return ClassFactory.GetInstanceImpl();

}

// factory methods

public abstract IChartPointsProcessor CreateCPProc();

**<..>**

}

***Classes intended to constructed via class factory declaration***

Interface declaration

public interface IChartPointsProcessor

{

**<..>**

}

Implementation

***Important:*** *marked abstract to hide possibility to explicitly construct it. Only derived classes can do that*

public abstract class ChartPointsProcessor : IChartPointsProcessor

{

**<..>**

}

***Implementation of class factory***

public abstract partial class IClassFactory

{

// implementation of ordinal class factory

// is accessible only by IClassFactory and class factory implementations (for DI purposes)

private partial class ClassFactory : IClassFactory

{

public ClassFactory()

{

**<..>**

}

private static IClassFactory Instance;

public static void SetInstanceImpl(IClassFactory inst)

{

Instance = inst;

}

public static IClassFactory GetInstanceImpl()

{

if (Instance == null)

Instance = new ClassFactory();

return Instance;

}

// IChartPointsProcessor factory

// Opens the back-door to construct ChartPointsProcessor object

// For access to non-default constructors appropriate delegating ones need to be added

private class ChartPointsProcImpl : ChartPointsProcessor { }

// IChartPointsProcessor factory method implementation

public override IChartPointsProcessor CreateCPProc()

{

return new ChartPointsProcImpl();

}

**<..>**

}

***Individual class factory***

// IChartPointsProcessor implementation for dependency injection

namespace impl

{

namespace DI

{

// Can be fully implemented from IChartPointsProcessor or extend any existing one

public class DIChartPointsProcessor : *ChartPointsProcessor*

{

}

} // namespace DI

}

// dependency injection class factory implementation

public abstract partial class IClassFactory

{

// Can be fully implemented from IClassFactory or extend existing one

// In this case I use ordinal class factory to override IChartPointsProcessor only

class DIClassFactory\_01 : ClassFactory

{

private class ChartPointsProcImpl : DIChartPointsProcessor { }

// IChartPointsProcessor factory method implementation

public override *IChartPointsProcessor* CreateCPProc()

{

return new ChartPointsProcImpl();

}

}

// instantiate di class factory. Needed because IClassFactory declaration are made inaccessible

public static IClassFactory GetInstanceDI\_01()

{

return new DIClassFactory\_01();

}

}

Somewhere (before construction of other class factory objects started) call

Utils.IClassFactory diCF = Utils.IClassFactory.GetInstanceDI\_01();

// after calling this all objects will be constructed via this class factory

Utils.IClassFactory.SetInstance(diCF);

**How it works**

***Contents(????????)***

***ChartPoints***:

* User friendly interface for adding ChartPoints (aka breakpoints)
* Taggers in code editor indicating their placement
* List of ChartPoints in special tool window
* Simple code text changes listener (aka breakpoints)
* Save/Load defined ChartPoints
* Separate ChartPoints mode from ordinal builds
* ChartPoints validation before build, before/after save/load
* User interactive chart view
* Table view of ChartPoints values based on their generation time

***Code generation***

* MSBuild task for code tracing code injection
* Transport between MSVS extension & MSBuild task

***Trace library (publisher side – traced program)***

Minimize overhead between traced & untraced code execution. It’s very important because if they differ much it will provide different behavior and tracing becomes meaningless

***Trace library (consumer side - host)***

The requirements on this side are not as strict as on publisher’s side. The main goal of this project is to perform post analysis. So some lag in run-time allowed.

***Trace transport***

As mentioned earlier I decided to use COM exe server as transport layer between the program being traced and the host. As it seems to me this wasn’t a good idea and need to be changed in the future. I will change it in the future. So I will not describe it in detail.

***Description***

***Step #1 (Selection of ChartPoints)***

Visual Studio contains a set of interfaces for manipulating language code model. *CodeModel* – common for all supported languages. *VCCodeModel* – extended version for C/C++ projects. *FileCodeModel* – code model based on source file structure.

As I indicated in limitations section only class variables of C++ fundamental types are allowed to be traced. So the only one place where we can do it is class method definition.

Before showing context menu in code editor, we check the availability of adding ChartPoint in this place. This is performed in *CP.Code.Model.CheckCursorPos*() method. From *EnvDTE.ActiveDocument* we get current cursor position (*EnvDTE.ActiveDocument.Selection.ActivePoint*) and *FileCodeModel* (*EnvDTE.ActiveDocument.ProjectItem.FileCodeModel*). Using *FileCodelModel.CodeElementFromPoint* method we can check if we are inside method body. If so *Parent* property of returned *CodeElement* points to *VCCodeClass* object, which is used to get all class variables. The future injection point will be at the beginning of the line or immediately after open brace of method if cursor is on this line.

One ChartPoint can contain multiple traceable variables.

All set ChartPoints are added to “ChartPoints:design” tool window.

***Brief ChartPoints classes architecture***



All ChartPoints data is stored in tree structure. This objects provide events for subscribing on their Add/Move/Remove/Status changes. It gives the ability to easily operate them in forward (from root to leafs) and backward (based on events notifications) orders. Both approaches will be actively used further.

**Step #2 (Taggers)**

**VSSDK-Extensibility-Samples** contains samples showing basic usage of taggers. Also MSDN has several articles describing it. The beginning point is here: *Inside the Editor* (<https://msdn.microsoft.com/en-us/library/dd885240.aspx>).

But I want more:

1. Force taggers appearance/change
2. Optimize performance (exclude redundant updates)

**Short taggers overview**

Every time new document is opened/changed MSVS calls our custom *IViewTaggerProvider* implementation (if any) method CreateTagger to create *ITagger* object. It’s method *GetTags* will be later called from MSVS environment in order to determine if (and where) tags are present.

***Problem:***

*IViewTaggerProvider*.CreateTagger is called multiple times for the same document. It looks like it called for each window that can contain tags: code editor, find results (???). As I found the last one is called for code editor window. Yes, it’s works but I don’t have full understanding.. So this needs to be researched more clearly.

**Custom taggers implementation**



All created tagger are stored in association array with file names as keys. ChartPoints tagger provider subscribes on *IFileChartPoints* Add/Remove events with subsequent providing *IFileChartPoints* object to stored taggers. This gives them ability to subscribe on *ILineChartPoint* events notifications.

When document is opened for the first time or changed *GetTags* method of ChartPoints tagger is called. In this method intersection of lines in SnapshotSpan and stored containing ChartPoints numbers is performed.

If needed to update tag manually from outside *IChartPointsTagger.*RaiseTagChangedEvent is fired with parameter containing line number. *ITagger<ChartPointTag>.TagsChanged* event with *SnapshotSpan* containing only 1 line where ChartPoint is placed fired. This helps to exclude redundant checks on tags creation and provides the ability to force (re-)draw tags.

***Important***: all indexes (line/character numbers) used here are 0-based. *EnvDTE* indexes, which are used to calculate ChartPoints positions, starts from 1. And this is the cause of constant headache.

**Step #3 (Save/Load ChartPoints)**

All information about contained ChartPoints is saved per solution basis in \*.suo (Solution User Options) file.

In order to do it I use implementation of *IVsPersistSolutionOpts* interface which provides overloaded methods and a reference to *Microsoft*.*VisualStudio*.*OLE*.*Interop*.*IStream* object. This object is cloned & stored for use after solution loaded.

**Step #4 (Text changed tracker)**

Code changes are tracked only by text changes now. Perhaps this is enough. I slightly experimented with VCCodeModel but decided that it is too complicated & expensive.

Tracking system divided into 2 parts: UI (MSVS side) and Model (ChartPoints). It was done so when I thought that I will use both text change listeners & code model changes. Maybe someday I will return back to this.

***UI***

MSVS services for listen text changes are: *IWpfTextViewCreationListener* and *IWpfTextView*. Implementation of the first one provides to handle TextViewCreated(IWpfTextView). The second gives the ability to subscribe to *TextBuffer*.*Changed* event.

***Model***

*ICPTrackService* tracks ChartPoints Add/Remove events and provides small wrapper objects which hides ChartPoints objects references. This service and several events bind UI & Model.

*ChartPoints track service sequence diagram*

**

*IWpfTextViewCreationListener.*TextViewCreated(IWpfTextView) is called for each opened document. If no FileTracker objects for this file registered in *ICPTrackService* *TextChangeListener* will save store IWpfTextView object within filename. Later, If Model.FileTracker create event will be received *FileChangeTracker* object with references to IWpfTextView & FileTracker will be created. It will subscribe to buffer changed event and query validation from *FileTracker*.

**Step #5 (Code instrumentation)**

Code instrumentation is performed via MSBuild task placed in CPInstBuildTask.dll.

When start building (Globals.dte.*Events*.*BuildEvents*.*OnBuildProjConfigBegin* event handler), following actions performed:

1. Check the existence of ChartPoints in current project
2. Disable debug information generation (it’s not needed because after instrumentation executing code will differ from original one)

Here is the code doing it:

*EnvDTE*.*Project* proj = ..

...

*VCProject* vcProj = (*VCProject*)proj.*Object*;

*VCConfiguration* vcConfig = vcProj.*Configurations*.*Item*(projConfig);

*IVCCollection* tools = vcConfig.*Tools* as *IVCCollection*;

*VCLinkerTool* tool = tools.*Item*("VCLinkerTool") as *VCLinkerTool*;

tool.*GenerateDebugInformation* = false;

1. Validate ChartPoints
2. Transport for communication between MSVS host and MSBuild task is opened. ServiceHost with *NetNamedPipeBinding* is used for this. Why? It was the first that I saw starting to dig C# capabilities, :). As an address project file full name is used. It looks ugly but gives unique address. Maybe someone someday will decide to synchronously build the same project from several instances of MSVS and problems will be guarantied. But I believe in the power of reason.

MSVS host provides IPCChartPoint interface (and few others placed in the same file) which contains method for calculating ChartPoints injection layout for injection points:

1. Trace variables definitions
2. Trace points
3. Additional include file injection
4. .. and so on

***MSBuild task***

1. Open *ServiceHost* with same address
2. Acquisition of IPCChartPoint
3. Calculation of injection points layout IPCChartPoint.GetInjectionData(<project name)
4. Copy required source files to %TEMP% directory
5. Instrument them
6. Pass instrumented files to MSBuild (add them to build and remove interchangeable ones from build)

**Step #6 (C++ tracing library)**

To organize the correct variables tracking, the following data is required:

1. Identifier of traced variable.   
   The 64-bit value of variable address used for this purpose(\*)
2. Variable name
3. Type id  
   For further usage. Not used now.
4. Variable value
5. Timestamp  
   It’s taken at moment of tracing to provide reliable information

Predefined entities used for code instrumentation:

1. \_\_cp\_\_.tracer.h

**Step #7 (COM exe server)**