# Debugger Visualizers for SharpDevelop

* Quick introduction to the project (to get the reader really interested)
  + Object graph visualizer, debugger tooltips, collection visualizer
  + Screenshots
  + There is no other IDE that does this
  + Started as GSoC 2009 project
  + maybe a group photo of us, the developers, from the meeting in Austria
* Introduction to SharpDevelop source
  + Overview of SharpDevelop architecture, link to Daniel’s CodeProject article
  + How to build an addin (since debugger is an addin)
* Debugger.Core
  + What are Expressions, Values and how they work, how Evaluate() works
  + what are Permanent references and what they are useful for
  + how expression caching works in Debugger.Core
* Object graph visualizer
  + Motivation
  + The failed attempt to this in VS and why it failed
  + Describe the graph building algorithm, why it is possible in O(n)
  + design – how is model reused when eg. just one node is expanded, what happens if a node is collapsed (just the edge is removed), lazy evaluation on scrolling, example object diagram of node content-tree (ContentProperty nodes etc.). Example object diagram of whole graph (shown also using Object graph visualizer).
  + how tree layout works (classical two-pass, show the “too much wasted space” case, how could it be improved?) – how horizontal / vertical is done.
  + Discussion about layouting software, why none of them is good for incremental layout (dynagraph seems to be dead, link to list of layouting software). Why we picked Graphviz at least for spline routing (and why we abandoned it in the end.)
  + Spline routing
  + graph matching and animation
  + (spline morphing – if we do it, we could try, might be hard / slow)
  + memory leaks and weak events
  + (node control caching- if I get it working) – Creating User controls is *really* slow (baml parsing), I’d just want to reuse the ones from the previous graph
  + How to interrupt graph building if the user continues the debugger (eg. F10 for step)
  + Visualizing a Permanent reference instead of Expression (eg. for recursive methods)
* More interesting details of Debugger.Core
  + What are Expressions, Values and how they work, how Evaluate() works
  + what are Permanent references and what they are useful for
  + how expression caching works in Debugger.Core
  + interesting parts of Debugger.Core, why we decided to derive DebugType from System.Type (end of 2009).
  + ICorDebug + multithreading – unfortunately impossible, it’s STA (also some general talk about COM)
  + Slow evals, why they are slow and what could be done to do batch evals in the debuggee – inject batch-evaluator in the debuggee, then send a command to evaluate set of expressions.
* How profiling helped
* Collection visualizer
  + Motivation
  + Data virtualization for scrolling in collection visualizer (per column also)
  + How data virtualization was implemented generally and is actually used in both visualizers and in the tooltips as well
  + WPF UI virtualization
* Debugger tooltips
  + Motivation, (VS already has them)
  + Design – how the editor knows if to open a text tooltip (eg. over a method name) or a debugger tooltip. Iterating all addins and accepting the first one – common practice.
  + Debugger tooltips for IEnumerable – using “new List<object>(IEnumerable)” in the debuggee. How to do it for non-generic IEnumerable? Pros: no PermanentReferences needed. Cons: cannot display infinite collections – VS does it the same way.
  + How debugger tooltips are linked to the visualizers (opening visualizers from tooltips and how it is done)
  + How they are done using WPF Popup, (“+”) is a styled ToggleButton, what other options we had, how they were done in SD3 using a lot of code which we now don’t have to write thanks to WPF Popups
  + Closing the Popups (by clicking anywhere outside, F10 for debugger step etc.)
  + (Popups and keyboard input – if I figure this out, now they are not focusable)
* SharpDevelop console (ie. VS’s Immediate window)
  + Opening debugger tooltip instead of just text-dumping the evaluation result to the console
  + Overload resolution (if user types Foo(123, Bar().Len), how to tell which overload of Foo to call)
* (WPF – if the thesis is too short, otherwise see WPF Unleashed)
  + General concepts – markup, styles, templates

# Acknowledgements

I would like to thank MSc. Pavel Ježek for his helpful, constructive advice on this thesis.

I would like to extend a very honest “thank you” to David Srbecký, the author of the SharpDevelop Debugger, and Daniel Grunwald, the lead developer of SharpDevelop IDE. They both deserve a lot of respect for their work.

Last but not least, thanks to Google for supporting students to work on open source projects.

# Terminology

Debugger – a program which controls execution and observes the state of another program.

Debuggee – the program being debugged by the debugger.

IDE – Integrated development environment. A collection of tools such as a code editor, a debugger etc.

# Introduction

The idea for this thesis comes from a desire for better debugging features in current IDEs. We have had a vision about innovative ways of debugging – namely exploring object graphs and collections, and we implemented this vision as new functionality for SharpDevelop – the open source IDE for .NET. First, we describe motivation behind these features – why we built them even though developers could live without them so far. Then we describe the problems we had to solve when figuring out whether or not it is possible to implement these features, and how to go about it. We also describe the architecture of SharpDevelop – focusing slightly more on the part most important to us – the Debugger. Finally, an important part of this thesis focuses on detailed design of these features and their integration into SharpDevelop.

Because SharpDevelop is an IDE for building .NET applications written in C#, some parts of this thesis will refer to the .NET framework.

To give the reader a picture of our work right in the beginning, let’s start with a preview of the features we implemented:

## Object graph visualizer

Our first addition to SharpDevelop debugger is a visualizer of object graphs.

Screenshot

While debugging, the Object graph visualizer lets the user explore object graphs referenced by the variables in user’s code. For example, if the user has an instance of a linked list in the program, the visualizer displays the linked list in a similar fashion people would draw it on a whiteboard. The graph is updated live as the user steps in the debugger and the state of the data structure changes.

## Collection visualizer

The second addition to SharpDevelop debugger is a collection visualizer – a hopefully better way to explore contents of collections of objects.

Screenshot

While debugging, the Collection visualizer provides insight into the contents of collections of objects. The main point is that it displays properties of the collection items in a way that the user doesn’t have to dig into them manually.

## Debugger tooltips

The third feature we added to SharpDevelop are debugger tooltips.

Screenshot IEnumerable

Debugger tooltips are a popular feature of Visual Studio. SharpDevelop 3 also has debugger tooltips but they do not support IEnumerable collections, which is quite an important feature. As a part of this thesis we reimplemented the tooltips in WPF for SharpDevelop 4.0 and we added support for IEnumerable collections.

## Contribution of this thesis

The visualizers are not copied from any existing IDE or debugger and come completely from the ideas of the author of this thesis. As a result of this thesis SharpDevelop is currently the only IDE to have such features and we are expecting user feedback to see how these features will bring value to users in every-day development.

## Background

The first ideas for this thesis come from the beginning of 2009 when we were experimenting with visualizing object graphs using the Visual Studio debugger (<http://coding-time.blogspot.com/2009/03/debugger-visualizer-for-visual-studio.html>, thinking about improving the way people debug data structures. Then we stumbled upon a poster for Google summer of code 2009 at the Faculty of Mathematics and Physics in Prague. Seeing that SharpDevelop IDE was among the mentoring organizations and they even wanted their debugger improved, applying for SharpDevelop in Google Summer of Code was a no-brainer.

Google summer of code (http://code.google.com/soc/) is a program run by Google. Open source organization can apply and Google selects the most interesting of them - among participating organizations are such ones as Drupal, Eclipse, Firefox, gcc, Haskell, Mono, Ogre3D, OpenOffice and many others. Then students apply to individual organizations with detailed proposals of their ideas. Google distributes approximately 1000 slots to the organizations, based on how popular the organization is (that is how many applications the organization received). For example, SharpDevelop got five slots in 2009. Then it is up to the mentoring organization members (that is, the long-term contributors to the open source project) to select the students they like the most. These students then work for 3 months fulltime on their projects and receive $5000 from Google, provided the mentoring organization confirms that the student did a good job. The not-so-unofficial goal of Summer of code is that students stay with the project after the summer and become contributors.

We would like to thank all the members of the SharpDevelop team. They are without doubt the best programmers we had an opportunity to work with. They deserve a lot of respect not only for contributing their skills and free time, but also for doing very high quality work - SharpDevelop is a very good source for learning about design, coding practices and technologies. Moreover, the team members are glad to help with any questions and discussions with them are always effective.

Photo Ischl?

SharpDevelop meeting in August 2009 in Austria. Left to right: Tomasz Tretkowski (Gsoc: C++ Backend Binding), Daniel Grunwald (Senior Developer, Architect), Martin Koníček (Gsoc: Debugger visualizers), Siegfried Pammer (Gsoc: Xaml Binding), David Srbecký (Debugger), Peter Forstmeier (SharpDevelop Reports), Christoph Wille (Project Management).

# Motivation and goals

## Object graph visualizer

### Motivation

To see why visualizing object graphs could be useful let’s look at current state of visualization of data structures.

The most typical way of displaying data from the debugger to the user are **watches** (screenshot Eclipse) where the user can see variables visible in current scope and expand their properties in a tree view. If an object is expanded and it contains references to other objects, children representing these objects can be expanded further.

Debugger tooltips are a very similar feature to watches but they show the expression right under the mouse cursor as the user hovers the mouse over the code being debugged. This makes debugger tooltips a little more comfortable to use than watches.

But neither debugger tooltips nor watches are perfect. Let’s imagine a simple structure of two objects pointing to each other:

Hand drawing of A B boxes poiting to each other on a whiteboard.

The picture shows how people naturally depict such structures. However, this is how such structure looks using debugger tooltips:



As we can see, the user can expand infinitely and has very little means of determining how the structure actually looks in reality.

Another thing which is not very well solved today is visualizing *changes*. If the user makes a step in the debugger how can he determine what changes this step caused to the state of the program? Sometimes the change is easy to see, as in:

IFoo foo = GetFooImplementation(context);

But if the code being stepped over changes multiple variables, it is useful for the user to see what just happened. This is solved in some IDEs by highlighting the variables which changed.

Red variables in watches – screenshot

On this screenshot from Visual Studio we can see the variables whose state changed since the last time the debuggee was stopped (for example if the user performed a step in the debugger these are the variables that were changed by the step).

But this approach is far from perfect for visualizing changes in data structures. What if an item was inserted into a linked list? Or what if a tree rotation just occurred? We realized this problem when teaching the subject Introduction to programming for university freshmen. What we frequently observed was that there was code on one side of the whiteboard and a drawing of a data structure on the other side. The teacher was explaining the code by stepping through it on the whiteboard. He was moving his finger from one line of the program to the next and at the same time he was “updating” the data structure by erasing parts of it and drawing new parts. By seeing how each of the statements modifies the data structure the students could clearly see how the program works.

After several weeks of running this university class we had an idea – why not automate the process? The IDE could actually let users step through the code and draw and update the underlying data structure in a similar way the teacher did on the whiteboard.

### Goals

We set two main goals for our Object graph visualizer to address the issues described:

* Provide a way for the user to see the state of data structures in the program in a similar way people draw data structures on a whiteboard.
* Make it possible for the user to see how the data structure changes by stepping in the debugger. The more understandable the visualization of the change is the better.

## Collection Visualizer

### Motivation

The second issue we saw are insufficient possibilities of current debuggers to explore and understand contents of collections.

Say we are debugging a program which works with a collection of objects of type Person. By collection we mean a List, ObservableCollection, array, IQueryable, or any other IEnumerable. Currently (as of 2011), collections of objects are commonly visualized in the following way:

Screenshot tooltip with 1 item expanded

The obvious problem is that we cannot view all the Persons at the same time but we have to drill down manually, one item by one. How do we quickly get an overview of the contents of the collection? How do we quickly locate the item we are interested in?

### Goals

We set the following goals to address the issues described:

* Provide a way for users to get an overview of the contents of collections of objects more easily than with watches / debugger tooltips.
* Support all IEnumerable collections because SharpDevelop currently only supports Lists and arrays.
* Support large collections (thousands of items) efficiently because SharpDevelop currently takes too much time to expand a debugger tooltip for a large collection.

## Debugger tooltips

### Motivation

Debugger tooltips are a feature similar to watches. They are implemented by Visual Studio and SharpDevelop. This is how they look in Visual Studio 2010:

Screenshots tooltips VS2010.

The advantage of the tooltips compared to watches is that the users can just hover the mouse over anything they are interested in. They don’t have to look into the watch containing all the available variables.

Debugger tooltips in SharpDevelop 3 are working fine, except for the support of collections. They do not support IEnumerable collections – they support just arrays and Lists. The importance of support for IEnumerable is very high since the introduction of Linq (Language Integrated Query, url) API into .NET in 2008. Another issue is that SharpDevelop 3 has problems with displaying large Lists.

Moreover, the UI of SharpDevelop 3 was built using Windows Forms and for SharpDevelop 4.0, Windows Forms has been deprecated and the UI was completely rewritten to WPF. That means that completely new tooltips implemented in WPF were needed.

### Goals

We set the following goals for the new debugger tooltips:

* Support all IEnumerable collections.
* Support large collections without noticeable degradation in performance.
* Implement the UI using WPF.
* Make it possible to open the Object graph visualizer and Collection visualizer from the tooltips.

# Analysis

Having motivation and goals for our project laid out, let’s now analyze the project in detail.

## Introduction to SharpDevelop IDE

SharpDevelop (<http://www.icsharpcode.net/opensource/sd/>) is a free open source IDE for .NET written almost entirely in C#. Its development started from scratch in 2000. Version 4.0 (as of November 2010) supports C# 4, Visual Basic 10, F#, IronPython, Boo and C++.

screenshot

The level of support for the individual languages varies. For example, F# code-completion is currently under development. Visual Studio’s C++ support is definitely superior to the one of SharpDevelop, but C++ is not the main focus of SharpDevelop. Support for C# is comparable to Visual Studio and in some areas, SharpDevelop surpasses Visual Studio. All this information is for SharpDevelop 4.0 as of April 2011.

SharpDevelop runs on Windows (for Linux there is MonoDevelop which was forked from an early version of SharpDevelop). SharpDevelop uses .NET SDK for the build process (that is MSBuild and the compilers for individual languages). SharpDevelop 4.0 supports targetting .NET versions 2.0, 3.0, 3.5 and 4.0. Older projects can be automatically upgraded to target .NET 4.0. SharpDevelop uses the same project and solution file format as Visual Studio – therefore, it can be used side-by-side with Visual Studio without problems.

SharpDevelop (as of April 2011) does not fully support the following functionality:

* Web application development. It is possible to build and debug an ASP.NET application using SharpDevelop (http://community.sharpdevelop.net/blogs/marcueusebiu/archive/2010/12/28/sharpdevelop-classic-asp-net-websites-using-iis-express.aspx). However, the tooling (.aspx code completion etc.) is missing.
* WPF designer is work in progress.

As SharpDevelop is completely free, it makes sense to compare it to the Express version of Visual Studio. SharpDevelop has the following features which are not present in Visual Studio Express:

* Integrated profiler
* NUnit integration (with test runner in SharpDevelop)
* Subversion and Git integration
* Code coverage (PartCover integration?)
* ILSpy integration (free .NET Reflector alternative, by the SharpDevelop team)
* PInvoke integration
* Reports (think CrystalReports)
* Debugger visualizers
* Productivity features (ReSharper-like), Navigate to symbol
* Extensibility

Extensibility is an important feature: SharpDevelop can be extended or modified in almost any way. Its API is well designed and in case there is an extension point missing, the SharpDevelop team is open to good contributions from anyone. We will definitely include good contributions into official releases.

Being an open source IDE written entirely in C# makes SharpDevelop a very interesting project for programmers interested in .NET who would like to learn advanced topics and have their work used by many people.

## The architecture of SharpDevelop

Let’s have a high-level look at the architecture of SharpDevelop, its system of AddIns and how our Debugger visualizers fit into the picture.

### ICSharpCode.Core

Let’s first explore ICSharpCode.Core (further referred to as the Core) which is the very basis on which SharpDevelop is built.

The Core is a framework for building extensible applications. It provides an AddIn architecture, where AddIns can extend almost anything, including other AddIns. The main point of the Core is to allow users to provide extension points in their applications very easily.

Let’s look at the key part of the Core, which is the AddIn infrastructure – the AddIn tree.

#### The AddIn tree

The AddIn tree is a tree of objects defined by xml files with .addin extension. The Core then provides functionality of turning these xml definitions into objects.

An .addin definition file might look like this:

<Path name = "/SharpDevelop/Browser/Toolbar">

<ToolbarItem id = "Back"

icon = "Icons.16x16.BrowserBefore"

tooltip = "${res:AddIns.HtmlHelp2.Back}"

class = " SharpDevelop.BrowserDisplayBinding.GoBack"/>

<ToolbarItem id = "Forward"

icon = "Icons.16x16.BrowserAfter"

tooltip = "${res:AddIns.HtmlHelp2.Forward}"

class = " SharpDevelop.BrowserDisplayBinding.GoForward"/>

<ToolbarItem id = "Separator1" type = "Separator"/>

<ToolbarItem id = "GoHome"

icon = "Icons.16x16.BrowserHome"

tooltip = "${res:AddIns.HtmlHelp2.Homepage}"

class = "SharpDevelop.BrowserDisplayBinding.GoHome"/>

[...]

We can see that there is a path called “SharpDevelop/Browser/Toolbar” being defined. This path contains some definitions of toolbar buttons. Then in our application built using the Core we can call:

toolStrip = ToolbarService.CreateToolStrip(this,

"/SharpDevelop/Browser/Toolbar");

this.Controls.Add(toolStrip);

Which returns a ToolStrip object with 3 buttons and a separator, as defined in the .addin xml file. This ToolStrip is ready to be used in our application.

This functionality by itself wouldn’t be very interesting. The interesting part comes when an AddIn is written for our application by someone else, containing the following in its .addin definition file:

<Path name = "/SharpDevelop/Browser/Toolbar">

<Condition name = "SolutionOpen" action = "Disable">

<ToolbarItem id = "SyncHelpTopic"

icon = "Icons.16x16.ArrowLeftRight"

tooltip = "${res:AddIns.HtmlHelp2.SyncTOC}"

class = "HtmlHelp2.SyncTocCommand"

insertafter = "Separator1"/>

[...]

Our call to ToolbarService.CreateToolStrip will now return a ToolStrip with four buttons. That happens because the AddIn tree combines all the objects belonging to the same path together. This way, our application has already an extensible toolbar. If we will use the AddIn tree anytime we want to obtain some objects our application will be very extensible. And indeed there are not many good reasons not to use the AddInTree at many places because it is easy to use.

In the second .addin listing we can also see a Condition. The Condition tag causes the ToolbarItem to be enabled only if the SolutionOpen condition evaluates to true.

In the listings we can notice that every of ToolbarItem xml tags has a number of attributes. The attribute **class** determines the name of the class which handles click on the toolbar button. The **insertafter** attribute specifies at which position the item should be inserted. Insertafter refers to an **id** of another item. The id-insertafter relations are resolved using a topological sort algorithm and if the sort fails then an arbitrary order is used.

So far we have seen two types of tags which can be inserted inside a Path in the AddInTree: Condition and ToolbarItem. These xml tags are called **codons** in SharpDevelop terminology. There are six types of codons supported by default (make it a table):

* Class - Creates object instances by invocating a type's parameterless constructor.
* FileFilter - Creates file filter entries for the OpenFileDialog or SaveFileDialog.
* Include - Includes one or multiple items from another location in the addin tree. You can use the attribute "item" (to include a single item) **or** the attribute "path" (to include all items from the target path).
* Icon - Used to create associations between file types and icons.
* MenuItem - Creates a System.Windows.Forms.ToolStrip\* item for use in a menu.
* ToolbarItem - Creates a System.Windows.Forms.ToolStrip\* item for use in a toolbar.

The **class** codon is very useful for making anything not related to user interface extensible. A typical usage is to register classes implementing a common interface at a given Path and then obtain all registered implementations using a call similar to the following:

AddInTree.BuildItems<IVisualizerDescriptor>("/SharpDevelop/Services/Debugger/Visualizers")

Such call assumes that all the classes registered at the Path /SharpDevelop/Services/Debugger/Visualizers implement IVisualizerDescriptor interface. Again, AddIn writers can just register their implementations and we will obtain all of them from all AddIns combined. To register a class, stating its fully qualified name is sufficient:

<Path name="/SharpDevelop/Services/Debugger/Visualizers">  
        <Class class="Debugger.AddIn.Visualizers.ObjectGraphVisualizerDescriptor" />  
 </Path>

Further in Implementation mention how we are using Class to register debugger visualizer.

##### ICSharpCode.Core for WPF and Windows Forms applications

There are two versions of the Core: ICSharpCode.Core.WinForms and ICSharpCode.Core.Presentation, designed to be used in Windows Forms and WPF applications respectively. They both reference the assembly ICSharpCode.Core which contains all the common non-UI-specific functionality.

##### Lazy loading

To improve application startup time, the parts of the AddIn tree are only loaded when needed. For example if an AddIn only adds items to a menu, the AddIn assembly will not be loaded until the menu is first opened.

##### Doozers and more extensibility

So far, we have seen six default types of codons (Class, FileFilter, Icon, MenuItem, Toolbar). All the codons are actually not hardwired into the AddIn tree implementation. There is a general mechanism of turning codons (i.e. the xml tags) into objects. And even this mechanism is extensible as well. Each of the codons has a corresponding builder which can build an object out of the codon. These builders are called **doozers** – e.g. ClassDoozer for Class codon, MenuItemDoozer for MenuItem codon etc. The extensibility means that we can implement our own doozers for our own new codons, to define e.g. a new xml tag (codon) called *Debugger*:

<Path name="/SharpDevelop/Services/DebuggerService/Debugger">  
        <Debugger id="DefaultDebugger"  
                  supportsStepping = "true"  
                  supportsExecutionControl = "true"  
                  supportsAttaching = "true"  
                  supportsDetaching = "true"  
                  class="ICSharpCode.SharpDevelop.Services.WindowsDebugger"/>

</Path>

This is a good way to pass parameters from the .addin xml file to the object that will be constructed. As we said, we defined a new *Debugger* codon by defining a new Doozer. The last thing we have to do is to register this doozer in the Runtime section of the AddIn tree.

<Runtime>  
        <Import assembly=":ICSharpCode.SharpDevelop">  
 <Doozer name="Debugger" class="ICSharpCode.SharpDevelop.Debugging.DebuggerDoozer"/>  
 [...]  
 </Import>  
</Runtime>

Now the Core knows that when it encounters an xml tag called *Debugger*, it processes it using our *DebuggerDoozer* class.

Maybe show IDoozer definition.

Another codon we saw in one of the listings was a Condition:

<Condition name = "SolutionOpen" action = "Disable">

The SolutionOpen condition is provided by a class called SolutionOpenConditionEvaluator. Like Doozers, the condition evaluators have to be registered in the Runtime section of the AddIn tree in the following way:

<Runtime>  
        <Import assembly=":ICSharpCode.SharpDevelop">  
<ConditionEvaluator name="SolutionOpen" class="ICSharpCode.SharpDevelop.SolutionOpenConditionEvaluator"/>  
 [...]  
 </Import>  
</Runtime>

Maybe show IConditionEvaluator definition.

#### Localization

In XML: label = "${res: AddIns.Profiler.ProfilingView.CpuCyclesText }"

In code: StringParser.**Parse**("${res:AddIns.Profiler.ProfilingView.CpuCyclesText}")

The resource identifier can be any string but in SharpDevelop we are using namespace-like notation to prevent collisions.

#### PropertyService

maybe

#### Remarks

The Core is not dependent on SharpDevelop. Rather, SharpDevelop is an application built using the Core.

For more information about SharpDevelop.Core see <http://www.codeproject.com/KB/cs/ICSharpCodeCore.aspx> and http://www.codeproject.com/KB/cs/LineCounterSDAddIn.aspx

### Reusable parts of SharpDevelop

SharpDevelop is an application built using ICSharpCode.Core. Several parts of SharpDevelop are written as completely standalone libraries. These include:

* NRefactory (C# and VB parser and AST)
* ICSharpCode.SharpDevelop.Dom (representation of type system)
* AvalonEdit (code editor with syntax highlighting)
* Debugger
* Profiler
* Usage data collection
* Reports

All of these libraries are completely reusable. Most of them are integrated to SharpDevelop by AddIns which act as “glue” to provide the functionality of the libraries in SharpDevelop. E.g. Debugger.Core is a managed debugger library and Debugger.AddIn contains user interface and SharpDevelop-specific logic. In the same fashion, AvalonEdit is a code editor with support for syntax highlighting and AvalonEdit.AddIn adds SharpDevelop-specific behavior, like split-view, context actions etc.

Picture of editor, document, parser: parse text to NRefactory AST, update DOM, discard AST

Screenshot of CC with picture – design: obtained from DOM, ILanguageBinding.HandleKeyPress, AvalonEdit.ShowCompletion.

### NRefactory

#### AST

INode <- AbstractNode <- different types, most important Statement and Expression. Almost all of them generated – from where?

#### Visitors

Classical visitor pattern: INode.AcceptVisitor. e.g. ForStatement.AcceptVisitor just calls visitor.**VisitForStatement.**

IAstVisitor <- AbstractAstVisitor, NodeTrackingAstVisitor, AbstractAstTransformer which are generated (from where?) and traverse the AST in the “standard” way. Most user visitors subclass one of these (e.g. AbstractAstVisitor), override some methods and in the end always call base to continue the traversal.

Used on many places for many things. Code conversions are a neat feature in the past but not maintained anymore.

#### Parser

cs.atg -> Coco -> CSharp.Parser (partial class calling methods of its second part written by hand)

#### Lexer

Written by hand in Lexer.cs.

### ICSharpCode.SharpDevelop.DOM

Type system representation:

IEntity <- IClass (has members), IMember <- IMethod (which has parameters), etc. In SharpDevelop they represent both user’s types and types loaded from referenced assemblies.

IReturnType is a reference to a type. IReturnType.GetUnderlyingClass() returns the IClass this type represents. Can also be a ConstructedType, e.g. List<string> which doesn’t “exists” anywhere and then GetUnderlyingClass returns null.

Entities converted to string representation by IAmbience.

IExpressionFinder.FindExpression returns ExpressionResult: string + location + context (e.g. AttributeContext).

IResolver.Resolve takes the ExpressionResult and returns ResolveResult (e.g. TypeResolveResult which contains the resolved IClass, MemberResolveResult which contains the resolved IMember, UnknownIdentifierResolveResult, UnknownMethodResolveResult, etc).

### New NRefactory for SharpDevelop 5.0

Will replace NRefactory and DOM. Short explanation and link to github.

ExpressionFinder will be replaced by keeping the AST + looking up the node.

### ICSharpCode.SharpDevelop

As we said earlier, SharpDevelop is built using ICSharpCode.Core. SharpDevelop is composed out of AddIns. The main AddIn is SharpDevelop “itself” – contained in the project ICSharpCode.SharpDevelop, with the AddIn definition file ICSharpCode.SharpDevelop.addin.

The SharpDevelop AddIn contains the base of SharpDevelop functionality and UI. The SharpDevelop AddIn is not a working IDE though – it rather provides interfaces to be implemented by AddIns. For example, code editing, code completion, and debugger are all implemented as AddIns extending ICSharpCode.SharpDevelop. Also all the other AddIns extend ICSharpCode.SharpDevelop.

### The editor

### Language bindings

### Project system

IProject. MSBuildBasedProject is the base, does loading using Microsoft.Build.

### Debugger

SharpDevelop’s debugger consists of two components – Debugger.Core and Debugger.AddIn. Debugger.Core is a standalone debugging library for .NET. Debugger.AddIn integrates this library into SharpDevelop.

Both Debugger.Core and Debugger.AddIn were written by David Srbecký.

#### Debugger.Core

Debugger.Core is a standalone debugging library for .NET. It provides features typically present in debuggers: attaching to and controlling a user program, stepping, setting breakpoints, exploring state of the variables in the program etc.

Debugger.Core uses the low-level debugging COM API provided by the .NET runtime, called ICorDebug (<http://msdn.microsoft.com/en-us/library/ms404520.aspx>).

##### Terminology

The program under the control of the debugger is called a **debuggee**.

##### Fundamentals of debugging – copied from David

http://community.sharpdevelop.net/blogs/dsrbecky/archive/2010/07/29/debugger.aspx

The debugger can start a new debuggee process or it can attach to an existing one.  While the debuggee is running, there is not much the debugger can do.  Almost all operations are forbidden.  The debugger has to wait until the debuggee pauses - usually because user's breakpoint is hit.  Once the debuggee is paused, the debugger can investigate its state - it can look at the callstack, read local variables and so on.  Stepping or pressing "Continue" will put the debuggee into running state again.

An important thing to realize is that the debugger and the debuggee are running in separate processes.

##### Sample

To demonstrate Debugger.Core in practice, let’s look at an example of its usage. Assume we have the following “Hello world” program:

class Program

{

**public** static void **Main**(string[] args)

{

string message = "Hello World!";

System.Console.**WriteLine**(message);

}

}

This program can be debugged using the following code:

NDebugger debugger = **new NDebugger**();

Breakpoint breakpoint = debugger.**AddBreakpoint**("Program.cs", 6);

breakpoint.Hit += delegate { MessageBox.**Show**("Breakpoint hit"); };

// Start the debugee

Process process = debugger.**Start**("HelloWorld.exe", "C:\\", **null**);

// Waits until the breakpoint is hit if it did not

// already happen.

process.**WaitForPause**();

// The breakpoint hit message should be shown now

// Show the name of the current method on the stackframe

MessageBox.**Show**("Current method = " + process.SelectedStackFrame.MethodInfo.FullName);

// Get reference to the local variable

Value localVariable = process.SelectedStackFrame.**GetLocalVariableValue**("message");

MessageBox.**Show**(string.**Format**("message = {0} (type: {1})", localVariable.AsString(), localVariable.Type.Name));

// Resume execution after the breakpoint

process.**AsyncContinue**();

The program produces the following messages:

• Breakpoint hit

• Current method = Program.Main

• message = Hello World! (type: String)

##### Investigating state of variables

We saw that the sample code could output contents of some variables in the debuggee. Let’s see how that works and what the design decision were when building this part of the debugger API.

###### Values

In the sample code above (link), we saw a statement for obtaining a value of a variable:

Value localVariable = process.SelectedStackFrame.**GetLocalVariableValue**("message");

The object returned from this call is of type Debugger.Value.

We can then invoke e.g. Value.AsString() to obtain the string representation of the value.

What exactly is the Value? As we said, the debugger and the debuggee are running in separate processes. That means the Value cannot hold a direct reference to the value in the debuggee process (because memory spaces of individual processes are strictly separated by the operating system). Instead, some sort of interprocess communication must be used. The ICorDebug API used by the Value class under the hood takes of this.

If the value is of primitive type like string or integer, we can simply request the actual content.  However if the value is a class, we must enumerate its fields and properties and get the values for the ones that we are interested in.  We are of course free to get fields of the new values as well and drill down as much as we want to.

There is a good reason why this model is appropriate.  When the debugger's code was compiled, it did not know that the user will create a field "myHelloWorldMessage" and therefore it could not reference it.  Even if direct reference to the object in the other process was somehow available, the debugger would still have to use reflection to figure out what fields the object contains and then get their values one by one.  In fact, most of the debugger's API inherits from the abstract reflection classes (like Type, MethodInfo) so anyone familiar with reflection should have no problems using the debugger API.

###### Lifetime of Values

The .NET garbage collector (GC) presents a significant complication to the debugger.  When the debuggee is paused no code can be executed including the garbage collector so it is safe to investigate it as much and as long as we want.  However, if the debuggee is resumed even for just a few instructions, the GC might have been run and it might have moved all variables around in memory.  The GC takes care to update all references within the debuggee so that it does not even notice.  However, it unfortunately does not tell the debugger.  This means that whenever the debuggee is resumed, all debugger's Values become invalid because they might be pointing to wrong memory (Value holds a reference to the COM object identifying the value in the debuggee).  The next time the debuggee is paused, it has to obtain all values again.  This problem is more problematic than it might initially seem - getting a value of a property or calling Object.ToString() both require that the debuggee is resumed for a while so that the methods can be injected into the debuggee and executed.  Imagine that we are debugger tooltips to drill down to object "foo.bar.Person" which contains two properties - FirstName and Surname.  After we evaluate the "FirstName" property, all values will become invalid and we will have to obtain "foo.bar.Person" again just so that you we evaluate "Surname".

###### Expressions

To get around the problem with Garbage collection invalidating Values, Debugger.Core provides **expressions**. Expression represents a way to obtain Value, for example "foo.bar.Person“. Expression itself is not stored in the string form, but as a tree. Actually, Debugger.Core uses NRefactory Expressions to represent expressions. This is why Debugger.Core has a dependency on NRefactory.

Expressions can be turned into their string representation (e.g. "foo.bar.Person“) and parsed from a string in C# format. This functionality already comes from NRefactory.

Expressions can be Evaluated using Expression.Evaluate(), producing a Value. That makes them very useful – instead of keeping a Value and never knowing when it becomes invalid, we keep an expression an evaluate it whenever we need its value. Indeed, this how all the debugging UI of SharpDevelop uses the expressions: When the user has "foo.bar" open and expands "Person", SharpDevelop first generates the expression "foo.bar.Person" and then evaluates it.

At one point in the past, the Value class was designed so that it would remember the expression using which it was obtained and automatically reevaluate itself if needed.  However, this approach turned out to be quite difficult to debug since a relatively simple call could cause complicated chain of events.  The expression based approach is more explicit and thus allows better reasoning about the program - both in terms of behaviour and performance.

###### Expression caching

As we said – expressions are useful. Instead of keeping a Value and never knowing when it becomes invalid, we keep an expression an evaluate it whenever we need its value. Here obviously comes space for caching – there is no need to evaluate the same expression twice if the debuggee has not yet been resumed. We can remember the Value obtained by evaluating the Expression and when evaluating the same expression next time, just return the Value if it is still valid.

###### Expression evaluation

In the section about NRefatory we talked about Visitors. Visitors are used to the walk the expression trees for many various purposes (e.g. turning the expression tree into its string representation).

As Visitors are the primary way to work with NRefactory expressions, ExpressionEvaluator is also a Visitor. This makes a lot of sense because the evaluation can be defined recursively:

No matter whether we have an expression „list[i+3].Name“ or „person.Name“, we always want to evaluate whatever is on the left of „.Name“ and then on the result of the evaluation, get the value of the property called Name.

To evaluate „list[i+3]“ we evaluate „i+3“ (or whatever expression is in the indexer) and then evaluate the indexer, passing in the result.

This is exactly how the ExpressionEvaluator visitor works. Methods are also supported (so „foo.Bar(foo).Foo“ can be evaluated).

The evaluation uses some clever tricks to improve performance. For example, getting a value of a field from the debuggee is much simpler than calling a method. At the same time, in .NET it is very common to have properties with getter methods just returning a backing field, such as (in C#)

string Name { get { this.name; } }

or

string Name { get; set; } // backing field is generated by the compiler

To get the values of the properties, instead of invoking the getters, the debugger looks at the IL of the getter method to see if it is a method in the form „return field;“. Such IL can have 4 different versions (depending on the property being instance/static and the backing field being explicit/generated). Here is the code to recognize them all (can be found in DebugMethodInfo):

(**Read**(code, 0x00) || **true**) &&                      // nop || nothing   
(**Read**(code, 0x02, 0x7B) || **Read**(code, 0x7E)) &&   // ldarg.0; ldfld || ldsfld  
**ReadToken**(code, **ref** token) &&                      //   <field token>  
(**Read**(code, 0x0A, 0x2B, 0x00, 0x06) || **true**) &&    // stloc.0; br.s; offset+00; ldloc.0 || nothing  
**Read**(code, 0x2A);                                  // ret

Doing this speeds up a very common case of getting values of properties with a backing field but probably even more importantly, it does not resume the debuggee to obtain a value of every property, thus keeping the expression cache valid much longer.

* What are Expressions, Values and how they work, how Evaluate() works
* what are Permanent references and what they are useful for
* how expression caching works in Debugger.Core
* interesting parts of Debugger.Core, why we decided to derive DebugType from System.Type (end of 2009).
* ICorDebug + multithreading – unfortunately impossible, it’s STA (also some general talk about COM)
* Slow evals, why they are slow and what could be done to do batch evals in the debuggee – inject batch-evaluator in the debuggee, then send a command to evaluate set of expressions.

##### The type system

The debugger can not only provide information about the values in the debuggee, but also about their types. In fact, to be able to obtain contents of a complex value (instance of a class) we must know its type to be able to iterate its fields and properties and get their values one by one. This is a very common pattern used everywhere in SharpDevelop, including our visualizers.

The API provided by the debugger to investigate the types in the debuggee is very easy to understand if one is familiar with Reflection. In fact, the API is exactly the same as reflection. The class DebugType implements the abstract class System.Type, so it has methods like GetProperties() and GetMethods() which return System.Reflection.PropertyInfo, MethodInfo etc. They actually return debugger-specific implementations of these types but that is not a concern to the user.

Again, under the covers, the debugger uses the low level COM API provided by the .NET runtime (for example the IMetadataImport interface).

##### Architecture

The overall composition of Debugger.Core can be summarized like this:

* COM API:  The low-level unmanaged debugging API of the .NET framework.  It contains interfaces such as ICorDebug or ICorDebugManagedCallback.
* COM wrappers:  Auto-generated thin layer over the COM API which makes it a bit easier to use.  It converts 'out' parameters to return values and tracks returned COM objects so that they can be explicitly released (this is necessary so that the debugger does not lock assemblies).  The layer also contains several hand-written methods that handle marshaling of strings and other objects.
* NDebugger:  The debugging library itself.  It provides access to variables and types via reflection-like interface.  It provides commands for setting breakpoints, stepping and basically everything usually expected from a debugger.
* ExpressionEvaluator:  Extension on top of NDebugger which can evaluate C# expressions.  It depends on SharpDevelop's NRefactory.

#### Debugger.AddIn

UI, Visualizers, Tooltips.

## Object graph visualizer

Our idea of the Object graph visualizer is the following:

The user will enter an expression to be visualized. The visualizer will then explore the object graph starting at this expression and present it to the user in a similar way people draw data structures on a whiteboard.

When the user performs a step in the debugger, the drawing of the graph will be updated by a transition from the old state to the new state. This transition should help the user understand which changes occurred in the graph.

### Existing work

There have been attempts to do something similar. Probably the most significant effort so far has been the Data Display Debugger (<http://www.gnu.org/software/ddd/>) which is a graphical frontent to the command line debuggers like GDB or pydb, which is a Python debugger. Unfortunately, none of these debuggers support debugging managed .NET programs.

### What needs to be done

Let’s take a high level look at what needs to be done. Can the task be split into separate parts / steps? We want an object graph for a given expression to be presented to the user in a similar way people would draw this graph on a whiteboard. We can immediately see that this can be broken into two steps:

1. Given an expression, determine the graph (its vertices and edges)
2. Draw the graph

The first step we call “Building the graph”.

The second step (drawing) can be split into two independent steps:

* Determining the layout of the graph (that is the positions of nodes and edges on a plane)
* The actual drawing to the screen

The last required feature are the transitions which visualize the changes caused by steps in the debugger. This leaves us with:

1. Graph building
2. Graph layout
3. Graph drawing
4. Graph transitions

Actually, let’s think more about the Graph transitions. Should they be a completely separate step or will they interact with Graph building / layout / drawing?

To answer this question we have to realize that a debugger step can change absolutely anything about the state of the debuggee. To modify the graph incrementally, we would have to recognize what exactly happened during the step, when the debuggee was running, which is unfortunately not possible. This means that reusing any parts of the built graph / layout is not an option. We will not reuse the existing drawing either since different graphs can also have completely different drawings.

Therefore, the approach we will take will be to completely rebuild, relayout and redraw the graph. Only when we will have the drawings of two subsequent graphs (before/after the step) we will try to infer the transition between them.

So the conclusion is: yes, we will treat Graph transitions separately. The four steps (building, layout, drawing, transitions) will be four independent tasks.

### Graph building

Definition: Object graph is an oriented graph. Its vertices are in-memory instances. There is an oriented edge from vA to vB if and only if instance A has a direct reference to B (through a field or a property).

The problem: Given a reference to an instance in the debuggee, build an object graph representing all instances reachable from this instance (up to a maximum depth in case the graph is too large).

#### (Capabilities of the SharpDevelop debugger)

First we should look if the debugger we are going to use has sufficient capabilities.

#### Our solution

We propose the following algorithm to build an object graph given an expression e.

value = **Evaluate**(e)

graph = MakeGraph(value)

MakeGraph(value):

node = MakeNode(value)

foreach reference in **GetReferences**(value)

existingNode = GetSeenNode(reference)

if existingNode != null then MakeEdge(node, existingNode)

else MakeEdge(node, MakeGraph(reference))

As we can see this algorithm is quite straightforward – it does a DFS walk down the object graph in the debuggee, checking for already seen nodes. The result is a graph having the same “shape” as the object graph in the debuggee. The check for maximum depth is omitted for simplicity.

Let’s now analyze if this is possible to implement. The calls Evaluate and GetReferences will surely need to access the debugger API. Both of them will be possible to implement using the debugger API: Evaluating expressions as well as enumerating and evaluating fields and properties of objects is supported.

#### Determing whether an instance has been already seen – word order? Ask teacher Logio

Let’s now look at the function GetSeenNode in our algorithm. This is the key function because it detects cycles and shared references in the object graph. It takes a reference to an instance in the debuggee (a Debugger.Value) and returns a graph node that we already created for this instance, or null if we haven’t seen this instance yet.

How to implement this functionality? We will definitely have to keep some identification of already seen instances to be able to determine whether they have been already seen or not. We could even keep the Values themselves, in the form of Permanent references\*. (\*We are doing a lot of evaluations inside GetReferences – one for every field or property. Each of these evaluations would immediately render all our Values invalid. Luckily, PermanentRefences stay valid even after the Evaluation possibly causes a garbage collection in the debuggee). We could also keep seen expressions because they, when evaluated, also uniquely identify a Value. One algorithm could look like this:

MakeGraph(value): // same as before

node = MakeNode(value)

foreach reference in **GetReferences**(value)

existingNode = GetSeenNode(reference)

if existingNode != null then MakeEdge(node, existingNode)

else MakeEdge(node, MakeGraph(reference))

GetSeenNode(Expression expr)

foreach node in graphNodesSoFar

if Evaluate(BinaryOperatorExpression(op.Equals, node.Expression, expr))

return node

return null

In this algorithm we keep an Expression for every node in our graph. We start with the expression given by the user. In GetReferences, when we evaluate a property of an object we get a new Expression identifying the target. E.g. by evaluating the property “Name” of object “foo.bar” we get an expression “foo.bar.Name”). We then call GetSeenNode(“foo.bar.Name”). GetSeenNode then tries to ask the debugger whether this Expression equals to any of the already seen Expressions (trying to evaluate “foo.bar.Name == e{1..n}”, where e{1..n} are the expressions seen so far). This will work because an expression such as “a == b” evaluates to true if an only if expressions *a* and *b* refer to the same instance in the debuggee. This is equivalent to executing “a == b” in the debuggee.

This algorithm has one problem: GetSeenNode has to do many Evaluate calls. The total number of Evaluate calls in O(n.E) where **n** is the size of the resulting graph and **E** is the number of edges in the graph: GetSeenNode will be called once per edge and it has to do up to **n** Evaluate calls.

We implemented this algorithm first and found it to be too slow. Even though the graphs we want to visualize will not usually be very large, the Evaluate call is unfortunately so expensive that we have to do better.

Our idea is to use some other form of indetification of an instance in the debuggee so that we can use a hashtable to implement GetSeenNode better than in O(n), ideally O(1).

Let’s see if we can add our own identification directly to the instance in the debuggee. We cannot add new properties to the instances because it is not possible using the debugger API (and moreover the .NET environment is statically typed). What could work would be to create an instance of a Dictionary in the debuggee (this is possible by simply evaluating an expression “new Dictionary()”). This Dictionary would reliably map instances of the object graph in the debuggee to integers. As we would encounter new instances, we would add them to the Dictionary in the debuggee (by evaluating an expression such as “dict.Add(instance)”). To check if an instance has already been seen, we would evaluate “dict.TryGetValue(out instance)” which would give us our integer identifier which we could use to identify the corresponding node in our graph.

Such approach sound quite complicated and requires tampering with the user program. Let’s see if we can use something else.

##### .NET hashcodes

What about .NET hash codes?

Hash codes identify object instances and should be more or less unique. But there is no guarantee that they really will be unique – in other words two distinct instances could have the same hash code.

The theoretical reason is that the hash code is a 32-bit integer, therefore the space of all hash codes is limited and therefore it can be guaranteed that sooner or later we will encounter two different objects with the same hash code.

Apart from theoretical reasons there is, however, one very practical reason. One could think that 32-bit space is large enough and in practice we will almost never encounter different instances with the same hash codes. However, our tests on CLR (Microsoft’s implementation of CLI) are quite surprising. This code generates new objects until it sees two different objects with same hash code. It turns out that this happens very quickly, just after generating as few as 5000 objects:

Code from SO - <http://stackoverflow.com/questions/750947/-net-unique-object-identifier>

However, on second thought this result might not be that surprising at all. Even if the hash codes were generated completely at random with a good random distribution there would still be quite a high probability of hitting one value twice after 5000 attempts (this is an instance of the birthday paradox – numbers?). At the same time, trying to avoid this phenomenon could be contra-productive. For example, generating hash codes sequentially would not be a good idea as we know – hash codes should be well distributed if we want them to serve their purpose – to be keys for hash tables.

###### Hash code implementation in the runtime

In CLR, the runtime representation of every instance has a data member which stores the hash code for the instance. This data member is assigned by the CLR and it is accessible from managed code through Object.GetHashCode() method. User can override this method but the original runtime-assigned hash code stays accessible through the RuntimeHelpers.GetHashCode(object) method.

Note: the experiment Code from SO (ref to listing.) always ends after the same number of attempts on a given machine. This shows that the hash code generation is deterministic.

#### Conclusion

We are using hash codes to speed up our graph building algorithm. Because of the properties of the hash codes, the algorithm accounts for different instances having the same hash code. The final algorithm looks like this:

hashtable: 'hashCode' -> (list of objects with hash code == 'hashCode')

MakeGraph(root):

rootNode = MakeNode(root)

foreach reference in **GetReferences**(root)

existingNode = GetSeenNode(reference)

if existingNode != null then MakeEdge(rootNode, existingNode)

else MakeEdge(rootNode, MakeGraph(reference))

GetSeenNode(o) {  
    candidates = hashtable[o.GetHashCode()] // instances with same hashCode  
    if no candidates, the object is new  
    if some candidates, **get and compare their addresses to o.Address**  
      if no address equal (the hash code was just a coincidence) -> o is new  
      if some address equal, o already seen  
}

In GetSeenNode, we can use addresses in the short comparison because getting an address of a Value does not resume the debuggee.

Note: We always obtain hash codes using RuntimeHelpers.GetHashCode so that the user defined hash codes don’t interfere with the graph building algorithm. If we instead used Object.GetHashCode() and the user would override GetHashCode to always return zero (for example), our algorithm would still work. It would, however, run in O(n.E) – the same as the original slow algorithm.

### Graph layout

Once we have the representation of the object graph in form of vertices and edges between them, we have to think how to position these vertices and edges on a 2D plane so that it looks *natural* to the user.

#### Existing layout engines

Graph layout is a not a new problem. Naturally, we researched a lot about existing solutions. As SharpDevelop is LGPL-licensed free software, any commercial solutions were out of question.

A good comprehensive list of graph layout engines can be found at <http://blogs.msdn.com/b/saveenr/archive/2009/07/29/a-list-of-tools-for-automatic-graph-and-diagram-layout.aspx>. Most of the layout engines also handle graph drawing so we can think whether we could use them for the drawing as well.

##### Graphviz

It turned out very soon that the most commonly used engine is Graphviz (http://www.graphviz.org/). Graphviz consists of a set of executables (dot, neato, etc.) where each of these implements one particular layout algorithm. All of the executables act as batch jobs – they accept some parameters and standard input and produce some output.

As for the layout, Graphviz can probably do anything we need. As for the drawing, it can write the output into an image file. This is not good enough for us – we want to control the look of the drawing and also provide some interactive tooltips, zooming, and maybe rearrangement of nodes.

We are therefore looking at how to integrate just the Graphviz’s layout engine into our application (and do the drawing ourselves on top of it). We want to pass a graph to Graphviz and obtain the same graph with position information added. The communication has to be done by launching an executable, writing to its standard input, and reading from its standard output. Graphviz uses a text format for graph description as its input and one if its output methods is the same text format, annotated with position information. We would have to parse the position information from the output. This is not as easy as it could be if Graphviz provided some object-oriented API but it is definitely doable.

Maybe an example of how the text format looks, parameters to produce it (--output=plain?).

##### Dynagraph

Dynagraph (http://www.dynagraph.org) is an incremental layout engine based on Graphviz code. The communication is done through pipes like with Graphviz (there is also a COM interface which seems very incomplete).

What is really interesting for us is that Dynagraph is designed for *incremental* layouts. That means that we can send a command like “add edges between nodes A and B” to the standard input and Dynagraph reponds “node B moved down by 2cm, added edge A->B as a straight line” (<http://www.dynagraph.org/documents/dynagraph.html>). This could be very useful.

However, Dynagraph is an old library and unmaintained library (last update 2007), which is a big minus when deciding whether to add it to SharpDevelop.

##### MAGL

Microsoft Automatic Graph Layout (which started as a free Microsoft Research project called Glee) looks good, is written in .NET and could be integrated very easily. Unfortunately it is commercial.

##### GraphSharp

GraphSharp (http://graphsharp.codeplex.com/), originally written by one person as a hobby project, could also be a good option. It is written in .NET so it could be integrated easily. It even provides drawing using WPF, which would be probably sufficient for us. GraphSharp does not support smooth curved edges like Graphviz does, but we could probably add this functionality directly to GraphSharp.

Unfortunately, at the time of our research GraphSharp wasn’t available yet, so we did not consider it.

##### The rest

All the other layout engines we found are either not maintained anymore or serve a different purpose than we need. For example they only output drawings without providing the position information etc.

##### Our choice

So far, it seems that we will be going with Graphviz.

#### Dynamic graphs and incremental stability

There is one particular problem that we have to think about when choosing a layout engine. It is the requirement that the Object graph drawing should update dynamically as the user performs steps in the debugger.

As we said earlier, we will have to treat the graph before and after the debugger step as completely separate (because we can’t tell what exactly changed while the debuggee was running). This means calculating the layout for the graphs separately, draw them separately, and try to infer a smooth transition between them (by matching the vertices somehow).

However, there is a serious problem with most layout engines – **a small change to the graph can cause drastic changes in the layout**. Say for example that one node or edge is added to the graph and most nodes are rearranged completely in the new layout. In other words, the layout is *incrementally unstable*. This is a very bad user experience. On every debugger step, the nodes would move around randomly.

Dynagraph probably solves this problem but as we said, we are not going with Dynagraph. Graphviz suffers from this problem – both dot and neato algorithms do. That means we cannot just use Graphviz as it is.

We thought about how to overcome this problem and we had the following idea. We could draw the graph as a tree and always sort the children of every node alphabetically by the names of the edges, left to right. The names of the outgoing edges are names of the properties so the user would always see the properties of an object in the same order, which makes sense.

The layout will be still done separately for every graph but the restriction on the order of children reduces makes similar graphs look similar, therefore reducing the incremental instability.

This looks promising. But we are talking about trees and order of children and, of course, not every graph is a tree. This can be solved by choosing some edges as tree edges, calculating the layout for the tree, and only then adding the remaining edges to the layout.

So the solution we have so far is laying out the graph as a tree by picking a tree subgraph, and then adding the non-tree edges back to the layout.

Now we have to think how we will add the order-of-children restriction into the layout algorithm. This is possible to do with Graphviz using a slight hack – we could add edges between children 1-2, 2-3, … n-1 – n for every node. This should make Graphviz respect the order because it tries to optimize for total length of edges. However, it is a quite complicated solution considering we only want to calculate tree layout now.

In fact, it will be easier to just calculate the layout ourselves. We can still use Graphviz to calculate routes for the smooth splines avoiding nodes. Screenshot This will be possible – neato can accept a graph with fixed position information for nodes and add position information for edges. We will then parse the edge positions.

#### Size of the Graphviz binaries

We decided to calculate the layout for the nodes ourselves and to use Graphviz to calculate the paths for the edges.

But then there was an issue with Graphviz. The members of SharpDevelop team had objections against the size of the Graphviz binaries affecting the size of the SharpDevelop setup package. And they certainly had a point – the Graphviz binaries we needed to include into SharpDevelop were about 10MB, while the whole SharpDevelop setup is currently just 15MB.

The problem with Graphviz binaries is that they statically link a lot of libraries. Even when dot.exe or neato.exe are only used for text input and output, they won’t start without libraries for writing jpegs etc.

Here we had three options:

1. Modify the source code of Graphviz so that it does not reference all the libraries statically. Could be too complicated, and we would have to maintain our own version of Graphviz – how to merge bug fixes?
2. Ask the Graphviz team to do 1.). Could take too much time.
3. Do not use Graphviz at all and implement edge routing ourselves. The edge routing would be probably not as sophisticated as the Graphviz version, but should not be too hard to do it reasonably.

We decided for option 3.). Therefore, we are not using any existing layout engine in the end.

#### Summary

Here is a summary of our progress on Graph layout.

* Naturally we looked at existing engines.
* What about incremental stability? We should already think about graph transitions.
* No engine really solves incremental stability.
* Our idea for better stability: restriction – children ordered by property name.
* No engine provides reasonable restricting features. Tree layout is not worth hacking.
* Layout nodes ourselves, use Graphviz for nice splines.
* Graphviz binaries too heavy.
* Decided to implement also edge routing, therefore doing whole layout ourselves.

### Graph drawing

Having the graph with all the position information for nodes and edges calculated, we will want to display it. This should be not hard and we will use WPF to do it. If GraphSharp had existed when we were doing the research, we would have probably used GraphSharp for the drawing.

### Graph transitions

We also want to implement smooth transitions between object graphs when the user performs a step in the debugger. The crucial part of this will be determining which nodes represent the same debuggee instances in the two graphs pre/post the step. We call this *matching* the graph. When we have the matching, making a visual animation transforming the first graph into the second graph will be possible.

To realize the matching, we can use hash codes, PermanentReferences and addresses, in a similar way to what we are doing during graph building. We can remember a hash code and a PermanentReference for every node in both graphs. Then for every node in the new graph we can find a matching node in the old graph like this:

FindMatchingNodeInOldGraph(Node nodeFromNewGraph)

Find a node with the same hash code as nodeFromNewGraph’s hash code

If found, compare the addresses of the PermanentReferences of the found node and nodeFromNewGraph

If the addresses are the same, we have found a matching node

Otherwise nodeFromNewGraph has no matching node in the old graph (it was just added)

This will work because the hash code of an instance never changes during the lifetime of the instance, as we will be using RuntimeHelpers.GetHashCode(). That means we can safely search for instances which existed before the step by their hash code.

Comparing the addresses is important because different instances can have the same hash code accidentally. To compare the addresses reliably, we need PermanentReferences because the garbage collector could have moved the instances around in memory during the debugger step.

## Collection Visualizer

Our collection visualizer should provide a new way to see the contents of collections of objects in the debugger. The user should be able to see and understand contents of collections more easily than with watches or debugger tooltips.

We propose the following way: Display a grid where rows represent individual items of the collection and columns represent properties of individual items. This is how relational data is commonly being represented. We believe this is a good way to see data and that current debuggers are missing it.

### Existing work

There are some existing tools to visualize collections in the debugger in a similar way to what we are proposing, for example a Visual Studio debugger visualizer <http://davidhayden.com/blog/dave/archive/2005/12/26/2645.aspx>. The problem with this visualizer is that it is designed to visualize only one specific type of objects – e.g. a ShoppingCart, and the debuggee and the visualizer must both have a reference to this type. In other words, the visualizer is not generic.

On the other hand, we want our visualizer to be completely generic. The user should be able to visualize collections of any objects. We haven’t found any visualizer which does this (in the .NET world. Some extensions for Eclipse / NetBeans?).

### Current state of debugging collections in SharpDevelop

Apart from bringing the Collection visualizer to SharpDevelop, we would also like to solve the following two problems of SharpDevelop’s debugger:

* The debugger cannot display IEnumerable collections.
* The debugger cannot display collections of reasonable sizes.

In SharpDevelop 3 it was not possible to display contents of “collections” which were merely of type IEnumerable.

It was possible to debug large arrays using the debugger tooltips or watches. However, when trying to display a large IList, the implementation in Debugger.AddIn would eagerly get all the IList items from the debuggee through the debugger API, using one inter-process COM call to obtain each item. For a List with just a thousand items this would block whole SharpDevelop for several seconds (or minutes, depending on the size of the collection) since all the expensive debugger calls have to be executed on the main thread.

Our Collection visualizer should be able to get contents of plain IEnumerables and should not suffer a noticeable slowdown even when displaying very large ILists.

### What needs to be done

Summarizing our previous thoughts, let’s look at what the requirements are for the Collection visualizer:

* Displaying contents of a collection of objects: rows are objects, columns are their properties.
* Should support plain IEnumerable, IList, one-dimensional array.
* Should be fast enough for collections containing thousands of items.

Should IEnumerable and IList be treated separately? Why not just support IEnumerable, since IList and array are also IEnumerable?

What about the columns? If the collection contains Persons and Employees at the same time, will there be columns for the properties of Person, Employee, or both?

Are the speed requirements possible to meet?

We will look at these three questions in the following three sections.

### IEnumerable vs IList

Should IEnumerable and IList be treated separately, or is it enough to just support IEnumerable?

Indeed, if we find a way to support IEnumerable we will get the support for IList for free. But is this a good way to go? Let’s think about how to support IEnumerable first.

#### IEnumerable

Let’s say we have a Value pointing to an IEnumerable instance in the debuggee and we would like to display the items of the IEnumerable. We can obtain an enumerator by invoking “GetEnumerator” on the Value. Then we can invoke the standard “MoveNext” method and the “Current” getter in a loop to pull the Values from the IEnumerable into the debugger, one by one. Naturally, we want to do this lazily, that is to pull more Values on-demand as the user scrolls the view.

Pulling all the Values eagerly could take minutes, depending on the number of items. IEnumerable can also yield items infinitely - an uncommon case, but we can’t forget about it.

The approach we just decribed has some drawbacks:

* The number of items will not be immediately visible to the user. The user would have to scroll to the end of the IEnumerable to see the total number of items. This could be solved by invoking Count() on the IEnumerable but we cannot be sure that the debuggee references System.Linq.
* The user will not be able to scroll very fast. When the user pulls the scrollbar very fast, it would be good to skip some items. This will be not possible, as to get to an item, we have to evaluate all the preceding items first.

#### IList

We could indeed just support IEnumerable and ignore special cases like IList. However, when we know we are actually dealing with an IList, we can do better:

* IList has a Count property, so that the user will immediately see the total number of items.
* IList has an indexer which means we can randomly access any item we want. The user will be able to pull the scrollbar at maximum speed; we will just skip some items without evaluating them.

#### Expanding items in the grid

There is one important aspect we should think about when deciding how to support IEnumerable and IList: We could want to provide some additional information about the individual items in the grid. Showing the values of all the object properties at once is good, but it would be good if the users could drill down even more - for example open debugger tooltips on individual items.

Screenshot mockup opening a debugger tooltip on a grid item

How to do this? To be able to obtain some information about single item, we need a Debugger.Value or a Debugger.Expression representing this item. We could remember a Value for every item when evaluating its properties, but as we know, Values become invalid very quickly. We would have to hold PermanentReferences, but holding many PermanentReferences is strongly discouraged. The last option then is, for every item, to hold an Expression.

With an IList, accessing every item by an Expression is not a problem - we can build Expressions such as “list[i]”. However, the items obtained from an IEnumerable have no Expression. They are simply not referenced by anything in the user program. They were just generated on demand and passed to the debugger.

So far, it seems that providing additional information about individual items will only be possible for ILists, and not for IEnumerables, unless we find some clever trick. This is another reason to treat ILists and IEnumerables separately.

Continues in the Implementation – there is a more detailed discussion in the implementation.

### Grid columns, generic vs non-generic collections

One important thing we didn’t think about yet is the following: Given a Debugger.Value representing the collection to visualize, how will we determine what the columns are?

The columns should represent the properties of individual items in the collection. But what if the collection contains items of different types? We cannot look at all the items in the collection – that would take too much time. We have the following three options:

1. Look at the first item and take its public properties. These will be the columns for the grid.
2. When scrolling and evaluating new items, add columns to the grid dynamically when new properties are encountered.
3. Look at the generic parameter of the collection (IList<T>, IEnumerable<T>) and take the public properties of the parameter type.

Option 3 is simple and would only work for generic collections. That would not be a big problem though since the non-generic versions of IList<T> and IEnumerable<T> (ArrayList and IEnumerable) are quite rare. Does IQueryable<T> need special handling? What about ObservableCollection?.

Option 2 is quite complicated and probably wouldn’t be a very good user experience.

Option 1 is very similar to option 3 but it would also work for non-generic collections. Both options 1 and 3 share one property: there can be items in the collection with some properties not displayed. For example a Car in a List<ITransportVehicle> has a property NumberOfWheels, but the columns only show properties of ITransportVehicle. This problem could be partially solved by the Expanding feature – the user could expand the item to see all its properties.

## Debugger tooltips

The third feature we describe in this thesis are the new debugger tooltips for SharpDevelop 4. We are adding support for IEnumerable collections and large collections to the SharpDevelop debugger.

### Existing work

There is a standard feature of Visual Studio very similar to our debugger tooltips. SharpDevelop 3 has debugger tooltips as well. Since SharpDevelop 4 was completely rewritten to WPF it needs new implementation of debugger tooltips. Also, the tooltips for SharpDevelop 3 didn’t support exploring instances of IEnumerable collections, which is quite a needed feature. Therefore, we are adding IEnumerable support to the debugger tooltips.

# Implementation

## Common base for visualizing collections

## Object graph visualizer

In the Analysis section, we decided about the very high level architecture of the Object graph visualizer. We decided to split it into the following parts:

1. Graph building
2. Graph layout
3. Graph drawing
4. Graph transitions

The graph layout part is intricate, because we want to achieve at least some degree of incremental stability of the layout – small changes to the graph should not result in large changes to its layout.

The design we propose is the following (put it landscape on a dedicated page):



Somewhere in the thesis (introduction?) mention this cool diagram:

Debugger.Addin -> Debugger.Core -> ICorDebug and around it the boundaries of SharpDevelop and the CLR (with JITted program in it)

### Graph building

The graph building algorithm was largely described in the Analysis section. In our implementation, this algorithm resides in the ObjectGraphBuilder class, which takes a Debugger.Expression and produces an ObjectGraph. Let’s now at look at one aspect of Graph building we haven’t discussed yet – *Expanding of nodes*.

#### Expanding nodes – mention in analysis?

So far, whenever we described our graph building algorithm, we mostly omitted the fact that the graphs can be very large. We just mentioned that the graph building algorithm should have some maximum recursion depth limit. But when the users enter an expression which evaluates to very large object graph, do the users actually want to see the whole graph up to some maximum depth?

We think that it makes more sense to let the users expand the graph themselves as they need. We display just the root node and a “plus” button next to every field or property which points to something (is not null). When the user expands the field or property by clicking the button, we add a new node to the graph, or just an edge if the target is already present in the displayed graph. We do not recalculate the whole graph – we assume that property getters don’t have any side effects on the rest of the graph, which we think is a reasonable tradeoff between graph correctness and expand performance. Having property getters modify objects would be a very bad practice and it is practically never done.

Of course, the user can also collapse nodes. Then we just remove the node and all its inbound and outbound edges from the graph.

Whenever we expand or collapse a node, we recalculate the layout for the graph. This is because we need to make space for the new node, or get rid of empty space after the node was removed. However, as we said, we don’t rebuild the whole Object graph from the debuggee, which is a rather expensive operation.

##### Remembering expanded nodes between debugger steps

A big part of the design of the Object graph visualizer deals with debugger steps. We are dealing with graph transitions (graph before step -> graph after step) and incremental stability. The expanding of nodes must fit into the picture. When the user expands some nodes, the nodes must stay expanded also after the step.

We realize this by remembering which nodes are currently expanded in a data structure called ExpandedPaths (good name). We remember them in the form of string expressions, which describe the paths the user has expanded in the graph. After the step, we expand the same paths, which is equivalent to the user manually expanding them – clicking the same “plus” buttons on the same nodes. Of course, these nodes might not represent the same debuggee instances anymore. But in general, we believe this is quite an intuitive approach to the user.

We said that after the step, we expand the same expressions (paths) again. Actually, we do not perform many expand operations to maintain the expand state after the step. Instead, we make sure the graph is expanded correctly right when it is built. We incorporate expanded paths right into the graph building algorithm. The ObjectGraphBuilder gets the ExpandedPaths data structure and when recursively exploring the graph, it only follows the paths which are present in the ExpandedPaths.

### Lazy evaluation of Node contents

### Node layout

Any graph layout algorithm needs to solve two problems:

* Determine the positions of nodes (node layout)
* Determine the paths of edges (edge routing)

These do not necessarily have to be two separate steps. However, Graphviz first determines positions of all nodes (of course edge information is considered in this procedure) and after this step is finished, the positions of nodes are fixed and edges are being routed, one by one.

As we said, we implemented the first step (node layout) ourselves and used Graphviz for the second step (edge routing). The reason is that no existing layout engine was satisfying in terms of incremental stability (i.e. small graph change should result in small layout change).

Let’s see how we approach node layout.

The problem (node layout): Given an oriented graph, determine position of each node in the plane.

Such description of the problem is too vague. It should also state what property of the layout the solution should try to optimize. For example, Graphviz’s dot algorithm tries to achieve layout in which as many edges as possible point downwards. Other algorithm might try to achieve minimal edge crossing. Other algorithm (Graphviz neato) tries to achieve “natural” layout by doing a physical simulation of springs representing graph edges.

Our solution aims for determinism and incremental stability.

Realizing that the algorithm will be used to layout object graphs, we decided that the algorithm would be a tree layout algorithm with deterministic order of node’s children. The input graph is not necessarily a tree but we layout is as it were a tree (by keeping n-1 tree edges and forgetting the rest of the edges).

The question is, if the graph has m > n-1 edges (n is the number of nodes), how to determine which edges to keep and which to discard. This is done by depth-first traversal of the graph:

Traverse(node):

NodeAlreadySeen[node] = true

For each edge in node.OutgoingEdges

If NodeAlreadySeen[edge.target]

edge.Discard = true // not a tree edge -> discard it

Else

Traverse(edge.Target)

In the end, all the edges for which edge.Discard is true are removed from the graph and the rest of edges is kept. This way we obtain a tree so that we can layout it using a two-pass tree layout algorithm:

First pass: “measure”

Measure(node)

Foreach child in node.Children

Measure(child)

subtreeHeight = node.Children.Sum(n => n.DesiredHeight)

node.DesiredHeight = max(node.OwnHeight, subtreeHeight);

Second pass: “arrange”

Arrange(node, position) // place node at given position

// center node relative to its children

node.Pos = CenterVertically(position, node.DesiredHeight)

childPos = position

Foreach child in node.Children

Arrange(node, childPos)

childPos.Y += child.DesiredHeight // place next child below this child

The important part of this algorithm is its determinism: the first child on input will always be the topmost child and the last child will be the lowermost child. In object graph the order of children corresponds to the order of properties in the class so it makes a lot of sense to order children like this:

Screenshot of object graph node with outgoing three edges

Screenshot of object graph collection node with outgoing three edges

Note that many layout engines do not order children deterministically. In Graphviz it is possible to force deterministic ordering of children by adding artificial edges between the children but this adds computational complexity and it is not completely reliable.

#### Similarity to WPF’s “Measure-Arrange” layout algorithm

It is not a coincidence that the two phases of the algorithm above are named “measure” and “arrange”. This algorithm works on exactly the same principle as the WPF layout algorithm. The same algorithm is probably used in most UI models (XUL, html?).

In WPF, the user interface elements form so called visual tree.

When an UIElement is asked for its size (Measure() method) it asks its children for their size and then determines its own size.

When an UIElement is told its new position (Arrange() method) it also repositions its children, knowing their sizes because Measure() has already been called before Arrange().

In WPF the user can define his own user interface elements and completely control layout of their children by subclassing FrameworkElement and overriding MeasureOverride() and ArrangeOverride() methods.

This is also how existing UIElements are implemented. For example, StackPanel is a panel which stacks its children one next to each other. This logic is implemented in the ArrangeOverride method. The Measure method tells that the StackPanel needs as much space as all its children together.

Measure and Arrange actually each have a parameter. Measure(Size maxSize) specifies the maximum available size which the element can use. Arrange(Size arrangeSize) means what? Read WPF Unleashed

### Spline layout (routing)

The problem:

Example of a solution (Graphviz): screenshot from Graphviz

One approach to routing splines is routing the edges one-by-one, that is treating every edge as completely separate input. This is how Graphviz does it. This approach can lead to spline overlaps.

Another approach could be routing the splines globally, i.e. every edge path can affect paths of other edges. Such approach could try to reduce edge overlaps while still maintaining reasonable edge paths. The Graphviz paper mentions this, but they do not provide any ideas for such algorithm.

As said, Graphviz was used for routing the edges and the appearance of resulting graphs was good: screenshot

However, the members of SharpDevelop team expressed a concern about the impact of the size of Graphviz binaries on the size of SharpDevelop installer: Graphviz needs about 10Mb of binaries and won’t run without them even if they are not needed (for example it crashes when a library for writing jpegs is missing even though the output type is set to *text*).

Therefore we decided to implement also the spline routing ourselves and get rid of Graphviz completely.

#### Our algoritm

We realized that when routing edges to avoid rectangles, the crucial points in the plane are the *corners* of the rectangles. Second, to make edges look natural to humans, some sort of shortest path routing would be reasonable. The Graphviz paper confirmed our ideas.

Like Graphviz, we route the edges one-by-one. This is the algorithm:

For each edge e in G:

Determine edge start and end point of edge e: draw a straight line from the center of edge’s source rectangle to the center of edge’s target rectangle. Where this line intersects the source rectangle is the start point es. Analogically end point et.

Build following graph Gv (visibility graph):

V = (every 4 corners every rectangle on input) + (for all e: es) + (for all e: et)

E = (pairs (u, v) from V where u is visible from v: straight line can be drawn from u to v without crossing body of any rectangle)

For each edge e in G:

In Gv find shortest path from es to et (using e.g. A\* or Dijkstra’s algorithm)

Smoothen the joins of obtained path by using Bezier curves

Note that the algorithm deals with the situation when boxes overlap.

Here is an example of a result:

screenshot

This algorithm is O(n3) where n is the number of boxes due to the construction of visibility graph: O(n2) vertex pairs are tested for visibility and each test needs O(n) line-rectangle intersections. It could be probably optimized but for our needs it is good enough.

Note that Graphviz also uses O(n3) algorithm for visibility graph construction.

This algoritm is implemented to be completely reusable. The implementation can be found in /SplineRouting folder of SharpDevelop repository (link). The reusability is achieved by writing the algorithm to work with interfaces. Programming against interface is a common design practice.



#### Edge overlaps

Like with Graphviz, the edges are routed one by one and sometimes can produce long overlaps. We solved this in the user interface by highlighting edge under cursor.

screenshot

#### Multiple edges, self edges

Our algorithm also deals with the situation when there are multiple edges between one pair of boxes. This is solved by distributing the edge starting and ending points along the border of the box. Self edges are solved in a similar way.

Screenshot multi edges

Screenshot self edges

#### Explicitly determined start and end point

Some layout engines (including Graphviz) let the user specify starting and ending point of the edge. This could be useful in object graph visualizer because the edge outgoing from an object property could start directly next to the title of the property. We considered this and concluded that the layout would get problematic when the contents of a node are scrolled – the edge point would have to move and the edge would have to be re-routed when scrolling. Therefore we decided not to implement this feature.

#### Join smoothing

The last step of the algorithm – join smoothing – makes the result visually much more appealing:

Screenshot with join smoothing

Screenshot without join smoothing

The principle is the following:

Given two consecutive line segments, replace the segments by a Bezier curve of order 3 extended by straight line segments on each end:

Screenshot

The control points of the Bezier curve lie on the original lines. The distance of the control points from the original points determines the “smoothness” of the curve:

(Screenshot)

### Graph matching and animation

### Graph drawing

### Graph transitions

### The result

## Collection Visualizer

### Lazy loading items when scrolling

Having said that the Collection visualizer obtains values from the debuggee lazily as the user scrolls down the grid, let’s now see how we designed this.

There is a conceptual difference between two types of collections:

* IList collections, allowing random access using the indexer
* IEnumerable collections, allowing only sequential access

We approached these two cases differently. When the visualizer is asked to visualize an instance of a collection it first checks whether the instance implements IList<T>. If the instance does not implement IList<T> it further checks if the instance implements IEnumerable<T>. Based on whether the instance is an IList<T> or IEnumerable<T> we choose between two approaches to display its items:

#### IList<T>

We use WPF ListView to display the contents of the collection. ListView (as any ItemsControl) has an ItemsSource property. ItemsSource is of type IEnumerable and the ItemsControl distinguishes whether the actual ItemsSource is an IList or just an IEnumerable: if it is an IList the ItemsControl uses its indexer to query individual items as it needs to render them when they come into view, mostly by scrolling. We take advantage of this.

What we do is that we supply a special collection as the ItemsSource. This collection is a wrapper around some data source (in our case the debuggee collection) and when asked for an item at index *i* it queries the underlying data source for item *i* and returns it (and caches it). We call this collection VirtualizingCollection and this principle is sometimes called data virtualization. We call the underlying data source IListValuesProvider.

The top part of the class diagram is a generic data virtualization implementation. The bottom part is specific to virtualizing collections of objects in the debuggee. ListValuesProvider is the implementation of IListValuesProvider which wraps the collection in the debuggee and makes calls to the debugger API. The items returned by the indexer of ListValuesProviders are instances of ObjectValue – our representation of a collection item in the debuggee. ObjectValue contains values of item’s properties converted to string by the debuggee-defined ToString() method. It also contains index of the item in the collection.

The last thing left unexplained is the Count property of the VirtualizingCollection. The value of this property is needed for the ItemsControl to display a scrollbar correctly. The VirtualizingCollection asks the data source for the count of its items and in our case the data source queries the debugger API for the value of Count property of the collection in the debuggee.

The good thing about this implementation is that when the user scrolls fast the ItemsControl only queries items which are needed to be rendered, skipping indices which were skipped by the fast scrolling (and saving unnecessary debugger API calls).

Note on data virtualization: As we said an implementation of IListValuesProvider wraps some underlying data source and queries items from it. In some cases getting a group of items from the data source one-by-one is more expensive than getting all the items in one call due to the overhead of each query. One such scenario are relational databases. In such case it is reasonable to implement some sort of paging in the value provider so that when it is asked for an item *i* it queries and caches a number of items around index *i* because it is very probable that these items will be needed right after.

#### IEnumerable<T>

As in case of IList<T> we use WPF ListView to visualize contents of IEnumerable<T> collections in the debuggee. The difference lies in the data virtualization. We designed two solutions – the first later replaced by the other. Let’s describe and compare both of them.

##### The first solution

As said, the ItemsControl’s ItemSource property distinguishes between IList and IEnumerable: whereas in case of IList it uses its indexer to query individual items, in case of IEnumerable it always enumerates all the items. Wrapping the IEnumerable in our special IEnumerable and relying on the ItemsControl to query next items as needed is therefore not an option. We will have to do more work.

Our solution is to subclass ListView and implement the “lazy” functionality we need there. The subclass is called LazyListView. LazyListView watches when its scrollbar reaches the bottom margin and queries more subsequent items from the data source and adds them to its ItemsSource collection.



In the beginning only a number of items (given by LazyListView.InitialCount) is loaded from the data source. Then as the user scrolls to the end of the view more items are added to the end of the collection and the scrollbar gets shorter. User can continue pulling the scrollbar down. The overall user experience is quite natural.

##### Alternative to the first solution

This was actually an idea of David Srbecký. In the previous diagram the logic of adding additional items when needed is implemented in the LazyListView class. We could also implement this in the VirtualizingIEnumerable in the following way:

VirtualizingIEnumerable is an ObservableCollection. Its Count initially returns some fixed number (analogy to InitialCount of LazyListView). Its indexer ensures that at least *i* items are loaded from the data source and returns the *i*-th item. When item at index near the value of Count is requested (that means the ListView wants to render an item near end of the collection), Count is incremented and PropertyChanged(“Count”) event is raised. The ListView therefore updates its scrollbar.

This solution is a little more complicated than the first one.

##### The second solution

The second solution is quite different from the first one. Instead of keeping a reference to an IEnumerator in the debuggee and querying subsequent items on demand, we enumerate all the collection at once, directly in the debuggee:

Say we have a reference to an IEnumerable<Foo> variable in the debugee, *v* be the name of the variable in the debuggee. We construct the following expression: “new List<Foo>(v)” and evaluate it. This allocates a List in the debuggee and runs its constructor. The constructor enumerates the IEnumerable and stores its items. The result of the evaluation is the reference to the new List. Finally, we visualize this reference exactly the same way as we visualize ILists (described in the previous chapter).

Note that in most cases enumerating whole collection in the debuggee is almost instant (or takes up to five seconds, then evaluation timeouts). What determines the speed of the visualizer almost completely is the expensiveness of the debugger API. Therefore it rarely makes a difference to enumerate whole collection in the debuggee and fetch first *n* items or fetch first *n* items from an IEnumerable in the debugee.

##### Comparison of the solutions

The first solution and its alternative are essentially the same and differ only in the way of determining when the user has scrolled to the end of the collection. We will therefore compare the first and the second solution.

As we said, the performance difference is in most cases negligible, unless each GetNext() call of the enumerator is quite expensive. In such cases the first solution wins.

The first solution can, unlike the second, visualize “infinite” collections. By infinite we mean IEnumerables such as:

IEnumerable MakeInfiniteCollection()

{

int i = 0;

while(true) {

yield return i++;

}

}

The first solution will allow the user to scroll “infinitely” (until memory runs out) and more importantly it will display the first few items of the infinite collection. The second solution will try to enumerate the whole collection in the debugee and therefore always timeout / run out of memory and display error message.

On the other hand the second solution lets the user immediately see how many items there are in the collection and scrolling is much smoother as many items can be skipped when scrolling fast.

The first solution has some quite significant disadvantage though: We wanted to implement a feature to be able expand each individual item of the collection as a debugger tooltip:

screenshot

This is very useful as the collection visualizer only shows stringified values of item properties and often the item is a composite object and we would like to explore it more deeply. Now, if we look at the first solution – it obtains items (Values) from an enumerator but these Values have no Expression. They are simply Values in the debuggee but there is not a way to describe a “path” (e.g. in C#) to obtain each Value. That means to be able to expand each Value we have to keep a PermanentReference to it. But this means we will be holding a lot of PermanentReferences (adding up as the scrolls through a long collection) and as we know this is discouraged by MS documentation of the ICorDebug API.

On the other hand if we are working with an IList each item has a clear expression – e.g. “list[5]”. Therefore we do not need to keep any PermanentReferences and when used expands item at index *i* we evaluate “list[i]” again and display a debugger tooltip for obtained value.

(Note that our list actually has no name such as “list” because we allocated it dynamically by evaluating expression “new List<Foo>(identifier)”. Indeed, the expression “new List<Foo>( identifier)[i]” is what we will be evaluating and it will work without reallocating the list each time because of caching in the ExpressionEvaluator: “new List<Foo>(identifier)” is cached as a PermanentReference and each evaluation of any expression that contains this expression will use the cached value.)

Expression caching is useful but we have to be careful with it: if we simply evaluated expressions like “new List<Foo>(identifier)[i].Property”, the “new List<Foo>(identifier)[i]” would be cached as a PermanentReference. We have to call the indexer and evaluate the property without using Expressions to avoid the Expression cache.

To sum up, what is important about the second solution is that we have a store for the items in the debuggee (the List) providing named access to any individual item of the collection. In the first solution we had to keep PermanentReference to each item to be able to access it again to expand it.

Visual Studio 2008 probably uses something similar to our second solution. This can be tested by trying to expand a debugger tooltip for an infinite IEnumerable. Instead of displaying the first few items a timeout error is displayed as the debuggee is given a few seconds to try to enumerate the collection.

##### Third solution

In theory it would be possible to combine the two solutions – have lazy access from the first solution and a store of items in the debuggee providing named access to each item. What we would have to do would be to evaluate “list.Add(value)” after obtaining each value from the enumerator. This way the list would grow as the user would scroll and any item could be reobtained and expanded by querying “list[i]”.

##### Our choice

The third solution would still suffer from the slow scrolling and unability to see the total count of items in the IEnumerable collection. We feel that not being able to visualize infinite collections is not such a big disadvantage of the second solution.

That is why we chose the second solution.

##### Note on garbage collection

Note that all the described difficulties are caused only by the fact that the Garbage collector can move instances in memory when compacting the heap. If the address of each instance stayed always fixed we would just enumerate the items, remember memory address for each of them and then accessed any item by its address. In garbage collected environment, PermanentReference brings us the same comfort with the exception that we should keep the number of PermanentReferences reasonably row (in order of hundreds).

### Expanding the items in the grid

### Generic vs. non-generic collections

### Lazy loading columns (object properties)

Show also IQueryable, ObservableCollection, IParallelEnumerable.

### Heterogeneous collections

### The result

Screenshot of collection visualizer

As can be seen from the screenshot each row of the table represents one item of the collection. Each column then represents one property. This is very similar to how relational data are typically being displayed. Indeed, collection visualizer can also be used to display collection of objects loaded from a database. In such case the user sees the data in a very natural way – the same way a database management tool would display them.

## Debugger tooltips

### The result

# Conclusion and future work

# User documentation

# Object graph visualizer

## First attempt to implement an object graph visualizer

The implementation described in this thesis is actually a second version of object graph visualizer. The first version was implemented for Visual Studio. The attempt was a failure (details described further) but provided important lessons on how some problems should (not) be approached.