# Debugger Visualizers for SharpDevelop

# Acknowledgements

I would like to thank my advisor MSc. Pavel Ježek for his time and his very 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 were both very helpful and the work on this thesis probably wouldn’t start without them. I would also like to thank Christoph Wille for running the project, Siegfried Pammer and Matt Ward for all their contributions and enthusiasm and of course Mike Krüger for originally starting the SharpDevelop project. They all deserve much respect for their work.

Last but not least, I thank Google for supporting students to work on open source projects. For many students, Google Summer of Code is the first opportunity to collaborate with great programmers on very large and widely used projects.

# Typography

Vertical line denotes citations. …

# Terminology

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

Debuggee – the process being debugged by the debugger.

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

By the term collection this thesis refers to .NET Lists, ObservableCollections, arrays, or any other IEnumerables. Similar classes exist in other environments and most ideas mentioned in this thesis would apply also to them.

Permanent reference – a Debugger.Value which has been made permanent and does not become invalid when the debuggee is resumed. – mention this terminology here or only later in the Debugger section? It is used throughout the thesis, after it was defined in the Debugger section of course.

# Context

This thesis is about debugging managed programs in the context of .NET. Therefore, whenever the text mentions classes and their instances, it refers to managed .NET instances, references etc. Likewise, the term **object** **properties** in the context of this thesis refers to standard .NET properties, that is essentially a getter and/or a setter method usually with a backing field. Still, a lot of concepts and ideas described in this thesis can be directly applied to other managed environments, for example Java.

# Introduction

The idea for this thesis comes from a desire for better debugging features in current IDEs. I have had a vision about better, innovative ways of debugging – namely debugging object graphs and collections, and I implemented this vision as new functionality for SharpDevelop – the open source IDE for .NET. This thesis first describes the motivation behind this functionality – that is why to build new features even though developers could live without them so far. Then, detailed analysis of the solutions is presented, including an overview of SharpDevelop architecture and means of integration of the work into SharpDevelop. Finally, this thesis describes the design of the codebase and highlights interesting parts of the implementation.

## Preview – remove? “Don’t know” that it will be 3 features yet?

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

### Object graph visualizer

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

Screenshot

While debugging, the Object graph visualizer lets users explore object graphs referenced by the variables in user code. For example, if there is an instance of a linked list in the program, the visualizer displays the linked list in a similar fashion it is commonly drawn 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 – an useful 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 which makes multiple properties of each item visible at once.

### Debugger tooltips

The third feature which was added to SharpDevelop are debugger tooltips.

Screenshot IEnumerable

Debugger tooltips are a popular feature of Visual Studio. SharpDevelop 3 has debugger tooltips but two important features were missing - support for IEnumerable collections, and support for large collections. As a part of this thesis the tooltips were reimplemented in WPF for SharpDevelop 4.0, including added support for IEnumerable collections and large collections.

## Contribution of this thesis

The visualizers 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 help users in their every-day development tasks. Some users have already provided positive feedback – for example, they expressed a need for a better Collection visualizer in other IDEs.

## Background

The first ideas for this thesis come from the beginning of 2009 when I was 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 I found about Google summer of code 2009 from a poster at the Faculty of Mathematics and Physics in Prague. Seeing that SharpDevelop IDE was among the mentoring organizations and the team wanted to improve their debugger, applying for SharpDevelop in Google Summer of Code was a clear choice.

Google summer of code (http://code.google.com/soc/) is a program run by Google. Open source organization can apply to the program and Google selects the most attractive organizations to participate. Among participating organizations are such ones as Eclipse, Firefox, gcc, Haskell, Mono, Ogre3D, OpenOffice and many others. The students then 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 had 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.

My experience with working on SharpDevelop has been very positive. The SharpDevelop team is made by the best programmers I have had an opportunity to work with. They deserve a lot of respect not only for contributing their skills and free time, but also for the quality of their work. SharpDevelop is a very good source for learning about design, coding practices and technologies. Moreover, the team members are helpful and discussions with them are effective.

Photo from meeting in Bad 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

Sjednotit a rozhodnout se az v analyze, ze to budou 3 features?

The introduction section provided short previews of the features covered in this thesis. This section described the motivation for this thesis. – jestli nebude preview tak vyhodit

Debuggers and Integrated Development Environments are a very live topic in software engineering and they have seen a lot of improvement over the last years. However, current debuggers still do not solve some scenarios sufficiently and this section identifies such scenarios.

## Object graphs

Currently, most visual debuggers are similar in terms of presenting data from the debuggee to users. The most typical way of presenting such data are **watches**, which show variables in current scope in a tree view fashion - if an object contains references to other objects, these become its children in the tree. **Debugger tooltips** are a very similar feature to watches but they show the expression at the mouse cursor hovered over the code being debugged. This makes debugger tooltips more comfortable to use than watches, since users do not have to look for the right variables in the watch window and switch their concentration between the watch window and code editor. – subjective?

(screenshot Eclipse watch, tooltips)

But neither debugger tooltips nor watches are perfect for all scenarios – take, for instance, a structure of two objects having a reference to each other:



The picture shows how people naturally depict such structures. However, this is how such structure is presented by watches or debugger tooltips:



As we can see, the expansion can continue infinitely and users have 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 a user makes a step in the debugger how can he or she determine the changes to the state of the program? Sometimes the change is easy to understand from the code, 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 whose values were changed.

Red variables in VS watches – screenshot

On this screenshot from Visual Studio we can see the variables whose values were changed since the last time the debuggee was stopped (for example, before a step).

Unfortunately, this approach is not great for visualizing changes in data structures. What if an item was inserted into a linked list? Or what if a tree rotation occurred? I had realized this problem when teaching Introduction to programming to university freshmen. What I frequently observed was that there had been code on one side of the whiteboard and a drawing of a data structure on the other side. I was explaining the code by pointing to the current “instruction pointer” with a finger and moving the finger from one line of the program to the next line. At the same time a student was updating the drawing of a data structure by erasing parts of it and drawing new parts, as the structure was being modified. By seeing how each statement modified the data structure the students could clearly see how the program works.

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

### Goals

This thesis sets two main goals to address the described issues:

* Provide a way for users 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 users to see how data structures are being changed by stepping in the debugger. The more understandable the visualization of the change, the better.

## Collections

The second issue with current IDEs are insufficient possibilities to explore and understand contents of collections of complex objects. For example, when debugging a program that works with a collection of objects of type Person, such collection is commonly visualized in the following way in current debuggers:

Screenshot tooltip with 1 item expanded

The problem is that there is no quick way to get an overview of the contents of the collection and there is also no way to quickly locate an individual item. The only possibility is to drill down the items of the collection, opening and closing them one by one, which takes a lot of time. Combined with the fact that collections are being debugged very often, this is a serious shortcoming of current debuggers.

Specifically to SharpDevelop, its integrated debugger currently lacks support for IEnumerable collections and does not support large collections in an acceptable way (debugging large collections can block whole SharpDevelop for large amounts of time, depending on the size of the collection).

### Goals

This thesis sets the following goals to address the described issues:

* Provide a way for users to get an overview of contents of collections of objects, in an easier, faster way than with watches and debugger tooltips.
* Add support for IEnumerable collections to SharpDevelop’s integrated debugger.
* Add efficient support for large collections (tens of thousands of items) to SharpDevelop’s integrated debugger.

## Debugger tooltips

Debugger tooltips are a feature similar to watches, as shown on the following screenshot from Visual Studio:

Screenshots tooltips VS2010.

The advantage of debugger tooltips compared to watches is that users can just hover the mouse over anything they are interested in. They don’t have to search the watch window which contains typically more than a dozen of available variables. Debugger tooltips are a frequently used feature of Visual Studio’s integrated debugger and SharpDevelop implements them as well. However, the debugger tooltips in SharpDevelop 3 are missing two very important features: support for large collections, and support for the IEnumerable type. The importance of support for IEnumerable is very high since typical programs use IEnumerable extensively. Moreover, the UI of SharpDevelop 3 was built using Windows Forms and starting with SharpDevelop version 4.0, Windows Forms has been deprecated in and the UI has been rewritten to WPF. That means that completely new tooltips implemented in WPF are needed, and this thesis covers a new implementation of debugger tooltips for SharpDevelop’s integrated debugger.

### Goals

To address the absence of needed features in the debugger tooltips and the switch to new UI framework, this thesis sets the following goals:

* 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 high-level goals for the thesis laid out, this section provides detailed analysis of the means, options and solutions.

## 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 April 2011) supports development in 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. Support for C# in SharpDevelop is well comparable to Visual Studio and in some areas, SharpDevelop surpasses Visual Studio. Regarding C++ support, Visual Studio definitely surpasses SharpDevelop, but C++ is not the main focus of SharpDevelop. 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 targeting .NET versions 2.0, 3.0, 3.5 and 4.0. SharpDevelop uses the project and solution file format of 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 currently not implemented.
* 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 out of the box
* Code coverage
* ILSpy integration (open source .NET decompiler, developed by the SharpDevelop team)
* Reports (developed as part of SharpDevelop)
* Debugger visualizers
* Productivity features (ReSharper-like)
* Extensibility

Of course, there are many small differences on both sides, but this should give a basic overview.

The last mentioned feature, Extensibility, is important: SharpDevelop can be extended or modified in almost any way. Its API is well designed and in case there would be an extension point missing, the team is open to good contributions. 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 (SharpDevelop is currently being downloaded around two thousand times a day).

## The architecture of SharpDevelop

This section provides a high-level look at the architecture of SharpDevelop, its extensibility model and how the Debugger visualizers fit into the picture.

### Reusable parts of SharpDevelop

Several parts of SharpDevelop are written as standalone libraries. These include:

* ICSharpCode.Core – the generic extensibility framework on which SharpDevelop is built
* NRefactory (C# and VB parser and AST)
* ICSharpCode.SharpDevelop.Dom (representation of type system)
* AvalonEdit (code editor with syntax highlighting and code completion window)
* Debugger
* Profiler
* Usage data collection (collecting data about how users interact with the application and uploading them to a server)
* Reports (reporting library)

All of these libraries are completely reusable. Most of them are integrated into SharpDevelop by AddIns which act as a “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.

### ICSharpCode.Core

SharpDevelop is an application built using a generic extensibility framework called ICSharpCode.Core (further referred to as the Core). The Core provides an AddIn infrastructure, 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.

The Core was developed for the purposes of SharpDevelop but it is a standalone framework on which SharpDevelop is built.

#### 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.

#### The AddIn tree

The extensibility infrastructure provided by the Core is called the AddIn tree. AddIns are defined in XML files with .addin extension. There are several standard tags, the basic one being a tag called Path which essentially enables adding extension points. Here is an example , where four ToolBarItems are being added to the AddIn tree to a path called “SharpDevelop/Browser/Toolbar”:

<Path name = "/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"/>

[...]

Then the application built using the Core can call the following API:

toolStrip = ToolbarService.CreateToolStrip(this,

"/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 functionality by itself wouldn’t be very interesting. The interesting part comes when someone else writes an addin for the application, specifying the following in the addin definition file:

Code listing title - HelpAddin.addin

<Path name = "/Browser/Toolbar">

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

<ToolbarItem id = "SyncHelpTopic"

icon = "Icons.16x16.ArrowLeftRight"

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

class = "HtmlHelp2.SyncTocCommand"

insertafter = "Separator1"/>

[...]

Now, the call to ToolbarService.CreateToolStrip will now return a ToolStrip with four buttons. That happens because the AddIn tree combines all the addin definitions together. By creating the toolbar using the Core API, the host application has made its toolbar extensible. The more the host applications uses such calls, the more extensible it will be.

The second code listing also shows a standard construct called Condition. The Condition causes the ToolbarItem to be enabled only if the SolutionOpen condition evaluates to true. The Condition has custom logic implemented in a C# class, which has to be registered in the following way:

<Runtime>  
        <Import assembly=":HelpAddin">  
            <ConditionEvaluator name="IsFileOpen" class="HelpAddin.IsFileOpenConditionEvaluator"/>

…

In the listings one can also notice that each of ToolbarItem xml tags has a number of attributes. The attribute **class** determines the name of the class the toolbar button click. The **insertafter** attribute specifies at which position the item should be inserted. Insertafter refers to an **id** of another item. The final order of items is resolved from the insertafter relations by a topological sort algorithm. If the sort fails then an arbitrary order is used.

So far the examples have shown 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 default codons(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 when providing extensibility not related to user interface. A typical usage is to register classes implementing an interface provided by the host application at a well-known Path:

Listing title ICSharpCode.Debugger.Addin.addin.

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

Then the host application can obtain all the available implementations:

Listing title Code in SharpDevelop

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

Such call assumes that all the classes registered at the Path /SharpDevelop/Services/Debugger/Visualizers implement IVisualizer interface.

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

##### 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 – maybe remove

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

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

### ICSharpCode.SharpDevelop

As said before, SharpDevelop is an application built using ICSharpCode.Core. The codebase of SharpDevelop itself consists of many AddIns. Even the IDE “itself” is an AddIn – contained in the project ICSharpCode.SharpDevelop, with the AddIn definition file ICSharpCode.SharpDevelop.addin. The SharpDevelop AddIn contains the base of SharpDevelop UI and a part of functionality but it is not a working IDE - 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.

### NRefactory

NRefactory is the object model for C# and VB code. The debugger is using NRefactory Expressions to represent expressions to be evaluated and the Debugger visualizers use NRefactory Expressions extensively.

#### AST

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

#### Visitors

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

IAstVisitor <- AbstractAstVisitor, NodeTrackingAstVisitor, AbstractAstTransformer which are generated (how?) 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 – maybe remove, will be replaced

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.

### The editor

### Language bindings

### Project system

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

### Debugger

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

The author of both Debugger.Core and Debugger.AddIn is David Srbecký.

#### Debugger.AddIn

Debugger.AddIn is essentially SharpDevelop’s integrated debugger. It contains all of the IDE-specific logic and user interface, including the Debugger tooltips and Debugger visualizers. Debugger.AddIn relies on Debugger.Core for most of its functionality, for example setting breakpoints, stepping, or evaluating expressions.

#### 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**.

##### Architecture

Debugger.Core consists of four cleanly separated layers. Starting from the lowest layer, these layers are:

* COM API:  The low-level unmanaged debugging API of the .NET framework.  The API 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.  ExpressionEvaluator depends on SharpDevelop's NRefactory.

##### Fundamentals of debugging – mostly 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

The sample code in the previous section showed that it is possible to obtain values of variables defined in the debuggee process. This section described how the process of obtaining values of variables works and what the design decision were when building this part of the debugger API.

###### Values

The sample code in the previous section (link) showed a statement for obtaining the value of a variable:

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

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

The object returned by the GetLocalVariableValue call is of type Debugger.Value. The Value class has an AsString() method which returns a string representation of the value of the variable.

What exactly is the Value? As said before, 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, its actual content can be requested.  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".

###### PermanentReferences

The ICorDebug API provides a facility to get around the problem with the lifetime of Values – it is possible to create a strong handle, which does not become invalid when the debuggee is resumed and always points to the right place in the memory where the debuggee instance resides, even after the target instance was moved by the garbage collector. This functionality is accessible using the Value.GetPermanentReference() method in Debugger.Core. In this thesis, we call Values returned from GetPermanentReference() *Permanent references*.

It seems that the problem with the lifetime of Values is solved by immediately obtaining a Permanent reference immediately when obtaining any Value. However, the documentation states that user code should never keep many Permanent references (more than a few hundred). Therefore, the problem remains as we are allowed to create Permanent references and use them shortly but we cannot keep many of them as long as we need to.

###### 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.

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 evaluation

The Debugger.Core uses NRefactory Expressions to represent expressions to be evaluated and expression evaluation is implemented in the ExpressionEvaluator class which is a Visitor. This makes a lot of sense since the evaluation can be defined recursively:

For example, when evaluating expressions *list[i+3].Name* or *person.Name*, *list[i+3]* or person is evaluated first and then the field or property called Name is evaluated on the result of the evaluation. To evaluate *list[i+3]*, *i+3* has to be evaluated first and finally the indexer is evaluated, passing in the result of the evaluation of *i+3* as a parameter.

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

Caching

Performance tricks

There are some clever tricks used to improve performance of the evaluation. For example, 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

Exploiting the fact that getting values of fiels is much faster that using getters (which requires resuming the debuggee and waiting for result), the value of the field is returned directly when possible.

This is done by looking at IL of the getter method to see if it is a method in the form „return field;“. Such IL can have four different versions when generated by Microsoft’s C# compiler (depending on the property being instance/static and the backing field being explicit/generated). Here is the code to recognize all these four versions (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

##### 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 the type of the value 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).

##### Multithreading

Unfortunately from the specification of the underlying ICorDebugAPI, all the debugger calls have to be invoked from the main thread. Evaluating multiple expressions at once would be a performance improvement for some parts of the code working with the Debugger API (including the Debugger visualizers) but this is not possible.

##### Evaluating multiple values at once

In the code of SharpDevelop’s Debugger.AddIn, one can notice a very common pattern – the values of fields and properties

#### The fundamental problem with visualizing debuggee state

Interesting – talk about two properties changing each other. Getter of A increments B and vice versa.

## Division of work – OK?

In the motivation section it was stated that this thesis aims to solve issues with the current state of debugging Object graphs and Collections and to improve SharpDevelop’s debugger tooltips in the direction of collection support. As visualizing changes in object graphs and visualizing contents of collections are distinct topics, it was decided (after a discussion with the SharpDevelop team) that there will be two new separate features in SharpDevelop – the Object graph visualizer and the Collection visualizer. The third feature are the debugger tooltips. The debugger tooltips will conceptually stay the same as they were in SharpDevelop 3, but will be re-implemented in WPF with added proper support for debugging collections. This thesis will analyze the Object graph visualizer, the Collection visualizer, and the Debugger tooltips mostly separately but it will also identify common functionality shared by these three features – such as collection support.

## Dealing with collections

From the goals set for the Collection visualizer and Debugger tooltips, it is clear that both of them will handle collections – the Debugger tooltips need support for IEnumerable collections and large collections and the Collection visualizer will handle the same types of collections as well. As for the Object graphs – the individual nodes in actual object graphs can, indeed, also be collections, therefore it would be best to support collections in the Object graph visualizer as well.

Screenshot same collection in all three visualizers

The common requirement for all of the visualizers is that large collections will be supported without significant degradation in performance. In the previous version of SharpDevelop, when a collection variable was expanded in the debugger tooltips, the tooltips tried to obtain all of the collection items from the debugger. Since the communication with the debugger has to be done on the main thread (as discussed in the section about the debugger), the whole IDE was blocked even for minutes, depending on the size of the collection.

This thesis takes (we take?) a lazy approach to getting the items of collections from the debugger – since only the first few items are displayed, only these are obtained. As the view is scrolled, more items are being queried from the debugger. This approach can be applied to both of the visualizers and the Debugger tooltips.

### Types of collections

The basic inheritance hierarchy of .NET collection interfaces is the following.



As seen from the diagram, all the collection types in .NET framework, including arrays, implement IEnumerable. Also, when dealing with user-defined collection types (including enumerators implemented using the C# construct **yield return**), it is safe to assume that any user type that has semantics of a collection will implement IEnumerable. Therefore by supporting IEnumerable, all possible types of collections would be supported. However, the IEnumerable interface itself is very basic – it only enables sequential iteration over the collection, without random access. The following three sections analyze possible approaches to supporting collection types in the visualizers. Prvni 2 sekce vlastne ukazuji nedostatecna reseni, ale jsou dobre jako dokumentace postupu mysleni – OK?:

### First approach – treating all collections as IEnumerable

The starting point for all the visualizers is a Debugger.Value or a Debugger.Expression (which, when evaluated, yields a Value). First, it must be verified that the debuggee instance represented by the Value implements System.Collections.IEnumerable, which can be done by examining Value.Type.FullName. Then it is possible to obtain the individual Values representing the items in a straightforward fashion: the method “GetEnumerator()” is invoked and the result is stored as a Permanent reference. Then, whenever the collection view is scrolled and more items are needed, the “MoveNext()” method and “Current” getter are be invoked on the enumerator to obtain a Value representing the next item. When “MoveNext()” returns false, there are no more items available.

#### Pros

* Supports infinite collections. It is not a problem if MoveNext() never returns false - first few items will be displayed immediately and more and more items will be obtained as needed.

#### Cons

There are several Cons to the generic IEnumerable solution with important implications which deserve a dedicated section.

##### Expanding individual items

A very important requirement in all of the visualizers is that the individual items of collections can be expanded further. To expand an item, an identification of the item is needed – either a Debugger.Value or a Debugger.Expression. For Values obtained from an enumerator, there are no corresponding Expressions – the instances returned from the enumerator can be new objects which cannot be reached through any variables in user code simply because the user code does not reference the instances. The only remaining possible identification of the Values returned obtained from the enumerator are therefore the Values themselves, in form of Permanent references (because without Permanent references they would all become invalid immediately when obtaining the next item).

The problem is that scrolling the view of the collection would cause more and more Permanent references to be kept and, by specification, it is not possible to hold many Permanent references. The only solution possible here would be not to hold any identification of the Values and when an item at index i is expanded, iterate over the collection from the beginning (by obtaining a new enumerator) to the i-th item and expanding it. Such solution would not be very efficient and moreover, the item obtained by re-enumerating the collection could be a different instance than the one which is being expanded – a significant problem.

##### The length of the collection

Working with IEnumerable means that there is no way to tell how many items the collection contains. This means that users will never know how many items there are in the collection. A scrollbar can be displayed, but it will not reflect the actual state accurately. The actual length of the IEnumerable can be determined by invoking System.Linq.Enumerable.Count(IEnumerable<T>) but it cannot be guaranteed that the debuggee references System.Linq.

##### Fast scrolling

Having to access the items IEnumerable collections sequentially means that scrolling the collection view faster than the items can be obtained from the debugger will not be possible.

#### Summary

The cons of the IEnumerable solution are significant. It can be observed that all of them are caused by the simple nature of the IEnumerable interface allowing only sequential access to the items. The problems with determining the length of the collection and fast scrolling cannot be resolved. Expanding can be solved only partially by enumerating the collection always from the beginning, possibly obtaining a wrong instance. The only Pro of this solution is the ability to visualize infinite collections, which is not a very common use case.

### Second approach – special case for IList

In the .NET collection interface hierarchy there is an interface called IList which adds an indexer allowing random access to the items. This section explores the possibilities of dealing with collections when the collections also implement IList. It can be verified that a Value implements System.Collections.IList by examining Value.Type.FullName. The individual items from the IList can be obtained by invoking the indexer getter on the Value representing the whole IList (using Value.GetPropertyValue(“Item”, i) – “Item” is a special property name for the indexer). This approach is almost equivalent to evaluating expressions such as “list[i]” where “list” is the name of the IList variable, with the exception that it also works when the whole IList was obtained from an enumerator and therefore has no Expression.

#### Pros

* Expanding individual items is not a problem: to expand an item at position *i*, it is sufficient to remember the Permanent reference representing the whole IList and invoke the indexer on this Permanent rerference, passing *i* as a parameter.
* The length of the collection is immediately known by evaluating the “Count” property on the IList Value. A scrollbar of correct size can be displayed.
* There are no limitations on the speed of scrolling. If properly implemented, it will be possible to skip evaluation of items which are being scrolled over fast.

#### Cons

* Works for IList but not for IEnumerable.

#### Summary

The Pros of making a special case for IList are so significant that it is definitely worth it to support IList separately. When an instance does not implement IList and only implements IEnumerable, a fallback to the basic IEnumerable solution can be implemented.

### Third approach – converting IEnumerable to IList

There is a another possible approach to IEnumerable which is not immediately obvious – having an expression “e” representing an IEnumerable<T>, by evaluating an Expression “new List<T>(e)”, the IEnumerable<T> gets fully enumerated directly in the debuggee (at full speed) and a Value representing the new List is returned. Then the visualizers can work with the List, enabling expanding, accurate scrollbar, and fast scrolling. The constructor of List<T> will be always available because List<T> resides in mscorlib.dll which is always loaded in any .NET process (mscorlib defines all the system types, like System.Int32 or System.String).

The solution is then to visualize ILists using the second approach and visualize IEnumerable<T> by converting it into IList and then using the second approach.

#### Pros

* All the Pros of the second approach hold not only for IList and IList<T>, but also for IEnumerable<T>.

#### Cons

* This solution cannot be applied to non-generic IEnumerable collections. Unfortunately, there is no method in .NET 2.0 mscorlib (we want to support .NET 2 programs) that accepts non-generic IEnumerable and returns an IList.
* Infinite IEnumerable<T> collections will not be supported – the evaluation will timeout.

### Conclusion

From the three approaches, the last one was chosen to be used in all of the visualizers. It has very strong Pros only at the cost of not supporting non-generic IEnumerable and infinite collections, which are both quite rare cases. To support these two cases, the fallback to the first approach can be implemented, but it was decided not to do so – it would mean maintaining more code only two support two uncommon cases which arguably most programmers almost never encounter (actually we had the first approach implemented but removed it for maintainability reasons – the first approach works with Values while the third approach enables working with Expression consistently everywhere).

Note: Experiments with Visual Studio 2008 show that its integrated debugger timeouts when expanding an infinite IEnumerable collection in a debugger tooltip. This means that the Visual Studio team is probably using a similar approach.

## Object graph visualizer

The high level picture 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. We want an object graph for a given expression to be presented to the user in a similar way people would draw the object graph on a whiteboard. Can the task be split into separate parts / steps? Yes it can, at least into these two steps:

1. Building the graph - given an expression, determine the vertices and edges of the graph
2. Drawing 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

The first three steps can be done independently but what 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 exactly what happened during the debugger step, when the debuggee was running, which is unfortunately not possible, because the debugger has absolutely no control of the debuggee when the debuggee is running. The code executed during the debugger step could change absolutely everything about the state of the debuggee. This means that incrementally modifying our Object graph based on some diff from the debugger is not possible, because no such diff can be obtained directly.

However, we propose that it is still possible to infer a diff based on observations of the debuggee state before and after the step. After the debugger step, a new Object graph will be built based on current debuggee state and the instances found in the Object graph will be compared to the instances found in the old Object graph from before the step. Having inferred a diff of the graphs, a graphical transition explaining the changes can be produced. More details can be found in a dedicated section (forward link?).

So the conclusion is: Graph building, Layout and Drawing will not depend on Graph transitions and the transitions will be a separate step. However, the Graph transitions will use information from the Graph building phase.

### Graph building

Definition: Object graph is an oriented graph, where vertices represent in-memory instances. Let vA and vB be two vertices representing two instances A and B. 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).

#### The algorithm

We propose the following algorithm to build an object graph given an expression e. The algorithm is quite straightforward – it does a DFS walk down the object graph in the debuggee, checking for already seen nodes. The checking for already seen nodes is done by the function GetSeenNode which is a crucial because it enables detecting shared references and loops correctly. The result is a graph having the same “shape” as the object graph in the debuggee (in other words the graph built by this algorithm is isomorphic to the actual graph in the debuggee).

value = **Evaluate**(e) // evaluate the Expression e using the debugger to obtain a Value

graph = BuildGraph(value) // build the graph

// value is a Debugger.Value representing an instance in the debuggee process

**BuildGraph**(value):

If a maximum recursion depth is reached, return null.

node = create a new graph node representing this value

// get all Values (instances) that this Value points to

foreach referencedValue in **GetReferencedValues**(value)

// do we already have a node reprenting the referenced instance?

existingNode = **GetSeenNode**(referencedValue)

// if yes, add an edge to this node

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

// otherwise continue recursively

else MakeEdge(node, **BuildGraph**(referencedValue))

Will this be possible to implement? The calls Evaluate and **GetReferencedValues** will 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. It is worth noting that the function GetSeenNode is actually what makes the Object graph special. Without this function the algorithm would be equivalent to expanding debugger tooltips or watches recursively and wouldn’t depict the actual structure of the object graph. The main question is therefore whether GetSeenNode will be possible to implement.

#### Analysis of GetSeenNode

In our algorithm, the GetSeenNode function takes a reference to an instance in the debuggee (a Debugger.Value) and returns a graph node that has been already created for this very instance, or null if we such node doesn’t exist in the graph yet.

How to implement such functionality? To be able to distinguish which Values have been seen from those which haven’t, the algorithm will have to keep some unique identifiers of the Values. Another option would be to add some extra information directly to the instances in the debuggee, but that is not possible, as in .NET it is not possible to add fields to objects at runtime\* (\*It is be possible to create a Dictionary in the debuggee, but Dictionary uses overridden equality – not instance equality. A section on this?). Thus, some unique identification of Values is needed. One such unique identification are Permanent. Another unique identification of Value are Expressions because, when evaluated, they also uniquely identify a Value.

A solution using Expressions follows. The main algorithm works with Expressions and keeps an Expression for every node in the graph. GetReferencedExpressions creates Expressions by appending all field and property names to given Expression, based on the actual type of the Expression, (the Type is determined by evaluating the Expression). For example by appending the property name “FooProp” to object “foo.bar” we get an expression “foo.bar.FooProp”.

**BuildGraph**(expression):

If a maximum recursion depth is reached, return null.

node = create a new graph node representing this expression

foreach referencedExpression in **GetReferencedExpressions** (expression)

existingNode = **GetSeenNode**(referencedExpression)

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

else MakeEdge(node, **BuildGraph**(referencedExpression))

When GetSeenNode then is to determine whether an Expression identifies an already seen instance, it queries the debugger to compare the Expression to all of the seen Expressions (by evaluating Expressions such as “foo.bar.Name == e”, where e is replaced by all the expressions seen so far, such as “foo.bar” etc.). This works because an expression “a == b” evaluates to true if an only if expressions *a* and *b* refer to the same instance in the debuggee, because evaluating an expression “a == b” is equivalent to executing “a == b” in the debuggee.

**GetSeenNode**(Expression exprToTest)

foreach node in graphNodesSoFar

if **Evaluate**(BinaryOperatorExpression(op.Equals, node.Expression, exprToTest))

return node

return null

There is one significant problem with this algorithm though: every call of GetSeenNode has to do many Evaluate calls. Since GetSeenNode will be called once per edge and it has to do up to **n** 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. We implemented this algorithm first and found it to be too slow. Even though the graphs to visualize will usually not be very large, the Evaluate calls are unfortunately so expensive that this algorithm is inacceptable.

The solution using PermanentReferences would be very similar – the algorithm would work with Values instead of Expressions, but it would have to query the debugger for reference equality in a similar way. Permanent references also provide addresses (offsets in the memory space of the debuggee) but these are not usable as they can be changed by the garbage collector moving the debuggee instances around during the run of the graph building algorithm, when the debuggee is being resumed and paused again. Therefore, addresses cannot be used as unique instance identifiers.

We can observe that the main problem with identifying debuggee instances by Expressions or Permanent references is that GetSeenNode has to compare all the unique identifiers one by one by querying the debugger. The best would be if we could find some other form of unique indetification of an instance in the debuggee such that GetSeenNode could run faster than in O(n), ideally in O(1) in average case.

##### .NET hashcodes

Can standard .NET hash codes serve as unique identifiers in the Object graph building algorithm?

In .NET, every object has a standard hash code assigned by the runtime environment, accessible by the method RuntimeHelpers.GetHashCode() (we are not interested in the standard Object.GetHashCode() because that has the semantics of user-defined instance equality and the Object graph should depict true instance equality). Regarding the Object graph building algorithm, the hash codes have a good property that they never change during the lifetime of an instance and, being integers, they can be used as keys for a hash table (they even should be reasonably distributed).

However, there is no guarantee that the hash codes will be unique – there is no guarantee that two different instances will have different hash codes. The “theoretical” reason behind this is that the hash code is a 32-bit integer, hence the space of all hash codes is limited and it can be guaranteed that sooner or later two different objects with the same hash code will be encountered. There is, however, one very practical reason. One could think that 32-bit space is large enough and in practice different instances with same hash codes would almost never be encountered. However, our tests on CLR (Microsoft’s implementation of CLI) are quite surprising. The following code generates new objects until two different objects with same hash code are encountered. It turns out that such case occurs very quickly, after creating as few a few thousand instances instances:

Hashtable hashCodesSeen = new Hashtable();    
LinkedList<object> l = new LinkedList<object>();    
int n = 0;  
while (true)    
{    
    object o = new object();    
    // remember object so that they don't get collected    
    // this does not make any difference though :(    
    l.AddFirst(o);    
    int hashCode = o.GetHashCode();    
    n++;    
    if (hashCodesSeen.ContainsKey(hashCode))    
    {    
        // same hashCode seen twice for DIFFERENT

Console.WriteLine("Hashcode seen twice after „ + n + „steps“);    
        break;    
    }    
    hashCodesSeen.Add(hashCode, null);    
}

During our tests, the ouput of this code was “Hashcode seen twice after 5322 steps.” These results seem surprising, but when thought our carefully, they make sense. 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 – is it true? Exact 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 RuntimeHelpers.GetHashCode(Object) method which returns the same value as the default Object.GetHashCode(). While Object.GetHashCode() can be overridden to define equality semantics for structures and classes, RuntimeHelpers.GetHashCode(Object) always returns the original hash code assigned by the runtime.

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 on the current CLR implementation.

#### The final algorithm

Our Object graph building algorithm will take advantage of the .NET hash codes. Because of the properties of the hash codes, the algorithm accounts for different instances having the same hash code. The final algorithm follows. The main BuildGraph algorithm stays almost the same, with the exception that is adds graph nodes into a hashcode->Nodes hashtable. Ideally, when there are no hash code collisions, the hashtable will contain exactly one Node for every hash code. Each node in the graph holds one Permanent reference.

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

**BuildGraph**(value):

If a maximum recursion depth is reached, return null.

node = create a new graph node representing this value (holding a Permanent reference)

add the node to the hashtable at key value.HashCode (where HashCode is the in-debuggee hash code)

foreach referencedValue in **GetReferencedValues**(value)

existingNode = **GetSeenNode**(referencedValue)

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

else MakeEdge(node, **BuildGraph**(referencedValue))

GetSeenNode then takes the advantage of the hashtable in the following way:

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

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

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

#### Expanding nodes

The graph building algorithm as described can be considered finished. It handles very large (and infinite) graphs by limiting the maximum depth of recursion.

There is, however, one user experience aspect that deserves analysis - when the user enters an expression which evaluates to a very large object graph, expanding the whole graph up to some maximum depth would be a bad user experience for two reasons:

* If the maximum depth is small, some nodes the user is interested in can be missing from the graph.
* If the maximum depth is large, there are too many nodes while the user is not interested in most of them. Visualization takes too much time to show nodes that are not needed and the drawing is confusing.

Instead of always traversing the whole graph, a solution is proposed where users can choose which nodes to include in the graph by expanding them manually. The algorithm stays essentially the same but it is split into steps determined by user actions:

// a property is being expanded on this node

Expand(node, propertyName)

// get the value of the property being expanded

targetValue = EvaluateProperty(node, propertyName)

// and add the value to the graph (either an edge to an existing node, or a new node)

existingNode = **GetSeenNode**(targetValue)

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

else MakeEdge(node, **NewNode**(targetValue))

next paragraph – OK?:

The algorithm does not recalculate the whole Object graph on every Expand action – it assumes that property getters don’t have any side effects on the rest of the graph, which is a reasonable tradeoff between graph correctness and expand performance. Having property getters modify other properties is rarely done in practice – the only reasonable use case is when the property getter caches a value. Such scenario would be solved by reevaluating all the properties of the instance on Expand action instead of evaluating a single property. In principle, property getters could modify also anything in the other instances of the graph, but that would a very bad programming practice and it is practically never done, so we decide not to make the general case slower in order to account for a theoretical edge case.

Of course, apart from Expanding, there will be also a Collapse feature, which will just remove the target Node and all its inbound and outbound edges from the graph.

### Graph layout

Having the Object graph (isomorphic to the actual graph in the debuggee) built, this section focuses on how to position the vertices and edges of the graph on a 2D plane so that it looks *natural* to users.

#### 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.

### Graph drawing

Having the graph with all the position information for nodes and edges calculated, graph drawing presents the graph using WPF. If GraphSharp had existed at the time of doing the research of layout and drawing engines, it would have been definitely considered for the graph drawing phase.

### 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?

We will try to answer these questions in the following two sections.

### Laziness, IList vs IEnumerable

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 easily. 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 obtained 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.

#### Trick - turning IEnumerable into IList

So far, it seems that we will be treating IList and IEnumerable separately, mainly because the IList interface is more suitable for the collection visualizer.

This leads us to another option – what if we created a List containing the items from the IEnumerable, right in the debuggee? We would then visualize this List the same way we visualize regular Lists.

This seems like a strange idea at first but actually evaluating an expression “new List<T>(expr)” does exactly what we need. It enumerates the IEnumerable in the debuggee, wraps the values into a List and returns this List in the same way like any other debuggee List.

We will lose the ability to explore infinite IEnumerables but this rare case is outweighed by the benefits of IList.

#### IList vs IEnumerable – summary

This is a short summary of our IList vs IEnumerable analysis so far.

In principle, we could treat all collections as IEnumerable. However, it makes sense to treat ILists separately because the IList interface is more suitable for the collection visualizer, which brings some usability and performance benefits. Actually, these benefits are so significant that we will convert the IEnumerable into an List in the debuggee and then treat everything as an IList. Having to deal only with ILists should also simplify 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 built as a part of this thesis are the debugger tooltips for SharpDevelop 4. We are adding support for IEnumerable collections and large collections into 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 has been almost completely rewritten to WPF it needs new implementation of debugger tooltips. Also, the tooltips in SharpDevelop 3 don’t support exploring instances of IEnumerable collections, while IEnumerable is being used in programs quite extensively. Therefore, we are adding the needed support for IEnumerable to the debugger tooltips.

### IEnumerable support

Similar as in collection visualizer

### Laziness, Large collections support

Similar as in collection visualizer

### Reusing existing code

Most of the existing data model for the tooltips should be reusable

# Implementation

## Common base for visualizing collections

Maybe here describe VirtualizingObservableCollection from Grid visualizer.

## 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.

Namespaces overview

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. Its method BuildGraphForExpression 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

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 some instance (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.

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, when expanding a node we don’t rebuild the whole Object graph from the debuggee, which would make the expanding very expensive.

Node expanding and collapsing is implemented in ObjectGraphControl.

##### 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 form of string expressions, which describe the paths the user has expanded in the graph. After the step, we rebuild the graph and expand the same paths again, 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.

### The ObjectGraph

* Same picture as in the overview – node contains tree of properties. Mention collection nodes

An ObjectGraphNode represents one debuggee instance and contains a tree of its properties. These properties are represented by PropertyNodes, each containing an ObjectGraphProperty. In the property tree there can be also special types of nodes, like BaseClassNode or NonPublicMembersNode.

#### Lazy evaluation of properties

One thing about the ObjectGraphProperties might be a little suprising. All of the them are actually “empty” after the ObjectGraph is built - their Value is an empty string. This is because they are only prepared to be evaluated.

Later, the ObjectGraphNode with the tree of ObjectGraphProperties is presented in the user interface. Screenshot one node

Only the properties which are currently in view are evaluated, and if the user scrolls down, the properties which come to view are evaluated also. This is a big performance saver especially when objects have a lot of properties.

The on-demand evaluation when scrolling is implemented in VirtualizingObservableCollection, which evaluates our ObjectGraphProperties using the IEvaluate interface. The node user interface (on the screenshot) is PositionedGraphNodeControl.

#### Collections

So far, we didn’t mention one thing. We also support collections in the ObjectGraph visualizer. When a graph node actually represents a collection (IList<T> or IEnumerable<T>), the ObjectGraphBuilder prepares one ObjectGraphProperty per one item of the debuggee collection, which is very fast. Then, when scrolling in the user interface, the individual items are evaluated as needed.

Because of collections, the lazy evaluation is not only a nice feature to have, but already a must.

### Graph layout

Having described what’s inside the graph nodes, let’s now see how to position the nodes and edges of the graph on a 2D plane. The 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. There could be an algorithm which tries to optimize the layout globally, considering all nodes and edges.

However, for example Graphviz first positions the nodes and fixes them (of course edge information is considered in the process) and after this step, the edges are being routed independently, one by one. That means the edges are not trying to avoid each other. This might seem quite naïve at first but it is not quite so. The cleverness of Graphviz lies in placing the nodes in a way which requires few edge overlaps.

At first, we were using Graphviz. But as we said in the Analysis section, due to our requirement for incremental graph stability we decided to implement the layout ourselves. We then used Graphviz to find the edges for the splines – Graphviz has a good algorithm which makes the edges look very much like if a person drew them. (link to graphviz paper)

Then, because the Graphviz binaries were too large relative to the size of SharpDevelop installer, we decided to leave Graphviz and implement also the edge routing ourselves. The point is that, like Graphviz, we are doing the graph layout in two separate steps.

#### Node layout

The problem (node layout): Given an oriented graph, determine positions of nodes in the plane so that some property is met (e.g. so that majority of people find the layout *natural*).

Different algorithms try to achieve *natural* output in different ways. For example, Graphviz’s dot algorithm tries to achieve a layout in which as many edges as possible point downwards. Another algorithm might try to minimize edge crossing. And Graphviz neato tries to achieve “natural” layout by doing an iterative physical simulation of springs representing graph edges.

Our solution aims for determinism and incremental stability. We use tree layout with fixed ordering of children.

Of course, the input graph is not necessarily a tree so we definitely have to clarify. What we do is that we select a subset of the edges to get a tree. We forget about the rest of the edges for a while. Then we layout the tree and fix the positions of nodes. Last, we route all the edges, one by one.

##### Determining tree edges – DFS vs BFS

We said that we select a subset of the edges to get a tree. The question of course is which edges to choose.

One possible approach is to start in the root of the graph and doing a standard DFS. The edges used by the DFS will be declared tree edges. Because the DFS enters and leaves every node exactly once, we can see that the resulting structure will indeed be a tree.

For completeness, we show pseudocode for such DFS.

Traverse(node):

NodeAlreadySeen[node] = true

For each edge in node.OutgoingEdges

If not NodeAlreadySeen[edge.target]

edge.IsTreeEdge = true

Traverse(edge.Target)

After some testing, we realized that DFS does not behave well in terms of incremental stability. The problem is that when an edge changes its target, it can influence the layout too much.

Screenshots from VisualizerDemo – before after step next to each other

The problem is that when a node has multiple possible parents, DFS prefers the parent quite arbitrarily – paths are preferred based on alphabetic ordering of edges. Here is an example:



We realized that switching from DFS to BFS could solve all these problems. BFS picks the parent based on the distance from the root. It prefers short paths, not arbitrary paths.

For comparison, this is the exact same situation as on the previous screenshot, now with BFS:

Screenshot

After some testing, we are now satisfied with the incremental stability of the graph layout.

##### Two pass tree layout

After choosing the tree edges, we have a tree so we can perform two pass tree layout. We call the two passes Measure and Arrange, like WPF does. We have two versions of the layout: top-down (root at the top and branches going down) and left-right (root on the left and branches going to the right). This pseudocode performs left-right layout:

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)

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

node.Pos = CenterVertically(position, node.Size, subtreeHeight)

currentChildPos = position

Foreach child in node.Children

Arrange(node, currentChildPos)

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

As we can see, 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

Node layout is implemented in the class TreeLayout.

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

We named the two phases of the layout Measure and Arrange, like in WPF. This is because our algorithm works on exactly the same principle as the WPF layout algorithm. This approach is also common in many other UI frameworks.

In WPF the user interface elements are organized into a tree called the Visual tree.

When an UIElement is asked for its size (Measure() method) it asks its children for their desired size so that it can determine its own desired size.

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

In WPF the users can define their own user interface elements and completely control the layout 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 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

#### Edge layout (spline routing)

Having the positions of all the nodes fixed, we can proceed with routing the edges.

The problem: Given a set of positioned rectangles on a 2D plane and a set of directed edges between pairs of rectangles, find paths for the edges so that the paths avoid the rectangles and look natural.

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 does not provide any ideas for such algorithm.

As said, we were using Graphviz for some time just for the edge routing and the results were good: screenshot

However, the size of Graphviz binaries is too high, given we need them only for edge routing. Graphviz needs about 10Mb of binaries and won’t run without them (it won’t run without dlls for writing jpegs even though the output type is set to *text*).

Therefore we decided to implement also the spline routing ourselves and left Graphviz completely. That said, this was mainly because we had very specific requirements. Graphviz is still excellent graph layout software. It could have richer interfaces than just batch standard input and output but its algorithms are good.

##### 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 is reasonable. The Graphviz paper confirmed our ideas.

Like Graphviz, we route the edges independently, 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 the 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.

This algoritm is implemented to be completely reusable. The implementation can be found in Debugger.AddIn.Visualizers.Graph.SplineRouting namespace in SharpDevelop. The reusability is achieved by defining simple interfaces and working with them. Anything implementing these interfaces is then suitable input to the algorithm. Programming against interfaces is a common design practice.



##### Edge overlaps

Routing edges independently can sometimes result in overlaps. We solved this in the user interface by highlighting the edge under cursor.

screenshot

##### Multiple edges, self edges

Our algorithm 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 for now.

##### 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 a straight line segments at 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 transitions

Now we know how build and layout one object graph. What is missing is how to produce an animated transition between two object graphs when the debuggee is resumed and paused again (e.g. on a debugger step).

To be able to present a change between two graphs, we need to know what changed – which nodes were added or removed, which edges changed their targets and which nodes moved to a new place in the graph. In other words, we need some form of diff.

Because there is no way the debugger could know what changed while the debuggee was running, there is no way the debugger can provide us with such a diff. We have to build the diff ourselves.

We remember the last graph and compare it to the new graph. We match the nodes in the graphs in a similar way when building an object graph – we iterate over one graph and try to find matching nodes from the other graph. We lookup nodes by hash codes. When a node with matching hash code is found, we compare addresses of PermanentReferences to be sure that the hash code wasn’t only a coincidence. All the nodes which could not be found are marked as removed, and all the nodes un-matched nodes in the new graph are marked as added.

We currently do not match edges. That means we can produce an animation where nodes are moving to their new places, but we do not animate edges changing their targets. This is not a big disadvantage though because when an edge is faded out and its new version is faded in independently at the same time, it is quite clear to the user that the edge just changed its target.

Screenshot edge fading out and in

Graph matching is implemented in GraphMatcher.MatchGraphs(). This method returns a GraphDiff describing the matching between nodes in two graphs.

### Graph drawing

Graph drawing is the final step which actually presents the data we have been building until now to the user.

From the layout, we already have full position information about nodes and edges. And the GraphDiff is exactly what we need for the animation.

Based on the diff, we prepare animation for each node. Some nodes slide to new position, some nodes fade out and some fade in. Old edges fade out and new edges fade in. Then we play all these animations at the same time.

Drawing is implemented in GraphDrawer. The drawer also adds tooltips to the edges. These tooltips appear on mouse over. Edges are thin and that makes it hard to target them with mouse cursor. Therefore each edge has invisible thicker line under it, which reacts to mouse movement.

### The result

Big screenshot with code in the background

After the user does a step in the debugger – result.

All of our work is included in the standard SharpDevelop release. The best way to try it is to download SharpDevelop. link

## Collection Visualizer

In the analysis section we spoke about the conceptual difference between IList and IEnumerable collections and the need for obtaining collection items lazily. Let’s now have a detailed look at how we implemented this.

### Lazy loading items when scrolling

Having said that the Collection visualizer must obtain the 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

Originally, we approached these two cases differently. After the implementation was in production for some time, we found a different way - we evaluate IEnumerables into Lists in the debuggee and then treat everything as Lists. However, we think that the first implementation designed separately for IEnumerable is interesting so we will show it here, along with our thoughts during the implementation.

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 getting the items:

#### IList<T>

We use WPF ListView to display the contents of the collection. ListView (being ItemsControl) has an ItemsSource property. ItemsSource is of type IEnumerable and the ItemsControl internally distinguishes between the actual ItemsSource being an IList or 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. We take advantage of this.

What we do is that we supply a special collection for the ListView.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 second. Let’s describe and compare them, and offer yet some alternatives.

##### 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 scroll position 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 (so the scrollbar gets shorter). As the user continues pulling the scrollbar, more and more items are being added.

##### 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 section).

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.)

###### Bypassing expression cache

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, so there would be one permanent reference per collection item. Therefore, we have to evaluate the properties after the indexer using the lower-level Debugger.Value layer (Value.AppendProperty().GetValue()), to avoid expression caching.

###### Summary

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 after the debuggee is given a few seconds 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 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]”.

The “list” wouldn’t be a variable in the debuggee as we can’t define named variables, but it could be a Value on which we would invoke Add method and Indexer. We just wouldn’t use the Debugger.Expression layer, but the lower Debugger.Value layer directly.

##### Conclusion – we chose the second solution

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. We enumerate IEnumerables directly in the debuggee and visualize all collections using a single implementation working with IList.

##### 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) according to documentation.

### Expanding items in the grid

Because the properties of each collection item can be complex objects themselves, we implemented a way to explore their contents as well.

Screenshot

Each collection item has an expand button which opens a standard debugger tooltip for the item.

We implemented the expanding in the following way: Each item is represented by an ObjectValue which holds its index. To expand the tooltip, we simply pass an expression representing the item (like “list[12]”) to a debugger tooltip and open it at a right position.

### Generic vs. non-generic collections

There is an important question when designing the collection visualizer – when given a collection, what should be the columns of the grid? We already discussed this in the Analysis section. We decided that the grid columns represent the properties of the single generic parameter of the collection. That means when we get IEnumerable<Person> we make one column for each public property of the class Person and its base classes.

When the collection contains subclasses of the generic parameter type, their properties are not shown. Showing properties of subclasses would mean leaving the cells empty for some rows, since they would miss the properties, and adding columns dynamically on scrolling. We are not doing this.

Mockup of how it would look (in Excel)

Using the generic parameter also means that we do not support non-generic collections.

Building the columns is implemented in GridVisualizerWindow.

### Lazy loading columns (object properties)

In the collection visualizer, we are evaluating rows on-demand, as the user scrolls down (this was described in the section Lazy loading items when scrolling link). But we are also loading the individual cells lazily - the user can select which columns to show and which to hide. When a column is hidden, it is not being evaluated. When the user decides to show the column later, the values in the column are only evaluated for the few items which are currently in view. In other words, the visualizer is as lazy as it can be.

The implementation exploits the way the WPF ListView evaluates databinding expressions. We bind ObjectValue[propertyName] to each column, and when a column is hidden, we remove it, so the ListView stops evaluating it. When we add the column back, the ListView only asks for the values which are currently in view. The laziness is implemented in ObjectValue and the databinding can be seen in where.

### The result

Screenshot of collection visualizer

The screenshot shows the current version of the collection visualizer. Each row represents one item, each column represents one property. This is very similar to how relational data are typically presented. 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 the same way they are used to.

Show also IQueryable, ObservableCollection, IParallelEnumerable.

## Debugger tooltips

Unlike with the two previous visualizers, the data model for the tooltips was already there in SharpDevelop. It didn’t support large collections so we had to add this to the data model.

The main part of work on the debugger tooltips was to implement the user interface and integrate it in into SharpDevelop’s code editor. We wanted the tooltips to work for C# and VB.

### The result

# Conclusion and future work

Great idea – collection visualizer is not needed – we can integrate it into tooltips. When a collection is expanded in the tooltip, clicking a small arrow adds columns.

In tooltip button to include properties from all base classes, so that user does not have to search. Or add type-to-search feature.

Draw a forest – all the local variables for example, with markers to show which node is which variable.

Plot visualizer (for someone interested).



# User documentation