This document contains four sample chapters from Martin Konicek’s master thesis *Debugger Visualizers for the SharpDevelop IDE*.

**Contents**

Building the object graph ...........................................................................................................................................1
Edge routing ..................................................................................................................................................................6
Fundamental problem of debugging .........................................................................................................................10
NRefactory ...............................................................................................................................................................12

**Building the object graph**

*Definition:* Object graph is an oriented graph, where vertices represent in-memory instances. Let \( v_A \) and \( v_B \) denote two vertices representing two instances \( A \) and \( B \) in the debuggee process. There is an oriented edge from \( v_A \) to \( v_B \) 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 given 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 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).
// evaluate the Expression e to obtain a Value
d value = Evaluate(e)
// build the graph
d graph = BuildGraph(d value)

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

BuildGraph(d value):
   If maximum recursion depth has been reached, return null.
   node = create a new graph node representing this value
   // get all instances this instance points to
   foreach referencedValue in GetReferencedValues(d value)
      // do we already have a node representing this referenced instance?
      existingNode = GetSeenNode(referencedValue)
      // if yes, just add an edge to this node
      if existingNode != null then MakeEdge(node, existingNode)
      // otherwise continue recursively
      else MakeEdge(node, BuildGraph(referencedValue))

Listing 1 – Graph building algorithm

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: evaluating
expressions as well as enumerating fields and properties of objects is supported. What
actually makes the Object graph special is the function GetSeenNode - without this function
the algorithm would be equivalent to expanding debugger tooltips or watches recursively and
wouldn’t follow the actual structure of the object graph. The main question therefore is
whether and how 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 instance,
or null if such node doesn’t exist in the graph yet.

To be able to distinguish which Debugger.Values have been seen from those which haven’t,
the algorithm will have to keep some unique identifiers of the Debugger.Values. One 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. Not being able to
add unique identifiers to the instances in the debuggee process means that unique identifiers
have to be added to the Values living in the debugger process. Permanent references
represent a possible unique identification. Expressions represent another possibility because
when evaluated they also uniquely identify Values. As it will be shown, Permanent references
and Expressions are mostly interchangeable for the purposes of building the Object graph.

A first solution using Expressions follows. The algorithm works with Expressions and keeps
an Expression for every node in the graph. GetReferencedExpressions creates Expressions

1 It would be possible to create a Dictionary directly in the debuggee by evaluating an expression
"dict=new Dictionary<object, int>()" and track instances using this Dictionary by evaluating dict[o] and
dict.Add(o, new_id). However, a Dictionary uses user defined instance equality (overridden
GetHashCode and Equals methods) while in the Object graph visualizer we are interested in instance
equality.
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 property name FooProp to object foo.bar we get an expression foo.bar.FooProp.

BuildGraph(expression):
   If maximum recursion depth has been 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))

Listing 2 - Graph building algorithm using Expressions

In order to determine whether an Expression identifies an already seen instance, GetSeenNode 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\(^2\). Pseudocode for GetSeenNode follows:

GetSeenNode(Expression exprToTest)
   foreach node in graphNodesSoFar
      if Evaluate(BinaryOpExpression(op.Equals, node.Expression, exprToTest))
         return node
   return null

Listing 3 – GetSeenNode has to compare given Expression to every other known Expression

As can be seen from the pseudocode, every call of GetSeenNode has to perform many Evaluate calls, which is the most significant problem of this algorithm because GetSeenNode will be called once per edge and it needs do up to \(n\) Evaluate calls, the total number of Evaluate calls in O(\(nE\)) where \(n\) is the size of the resulting graph and \(E\) is the number of edges in the graph. We implemented this algorithm 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 unacceptable.

The algorithm using Permanent references would be exactly the same, including the needed calls to the debugger in a loop, expect it would work with Permanent references. Permanent references provide instance addresses (offsets in the memory space of the debuggee) but these are not usable as they can be changed by the garbage collector which moves the 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 and the only way to use Permanent references is the same way as using Expressions.

\(^2\) Evaluating a==b does not execute a==b directly in the debuggee, but is evaluated recursively using the ExpressionEvaluator (see section Error! Reference source not found.), which must be implemented correctly so that the expected behavior is met.
The conclusion is 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 solution to this problem would be some different form of unique identification of instances in the debuggee such that GetSeenNode could run faster than in $O(n)$, ideally in $O(1)$ in average case.

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

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. In the Object graph visualizer we are interested in instance equality to display actual state of objects in memory, not user defined equality. Therefore the default hash codes (accessible through RuntimeHelpers.GetHashCode) will be used. The default 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.

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 (on a 32-bit system), 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 in a given small object graph. 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:
Hashtable hashCodesSeen = new Hashtable();
LinkedList<object> l = new LinkedList<object>();
int n = 0;
while (true)
{
    object o = new object();
    // remember instances to be sure they don't get collected
    l.AddFirst(o);
    int hashCode = o.GetHashCode();
n++;
    if (hashCodesSeen.ContainsKey(hashCode))
    {
        // same hashCode seen twice for DIFFERENT instances
        Console.WriteLine("Hashcode seen twice after ", n , " steps");
        break;
    }
    hashCodesSeen.Add(hashCode, null);
}

Listing 41 – Generating instances until a hash code collision is found

During our tests, the output of the program was Hashcode seen twice after 5322 steps, which means that the possibility of different objects having same hash codes is very real and must definitely be accounted for. The article [19], which was the original source for this experiment, gives very similar results.

Interestingly, during our tests, the output was always the same on a given machine. This shows that the hash code generation on the current implementation of the CLR is deterministic.

The final algorithm

The final Object graph building algorithm takes advantage of the .NET hash codes while correctly accounting for different instances having same hash codes. This is done by looking up graph Nodes by hash codes and comparing instance addresses\(^3\) in case of hash code collisions to be sure the hash code collision was not a coincidence. The main BuildGraph algorithm stays almost the same as in the first version, with the exception that graph Nodes are being added into a hascode->Nodes hashtable. Ideally, when there are no hash code collisions, the hashtable will contain exactly one Node for every hash code.

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

BuildGraph(value):
    If a maximum recursion depth is reached, return null.
    node = create a Node for this value (holding a Permanent reference)
    add the node to the hashtable,
        key=value.HashCode  // HashCode being the in-debuggee hash code
```

\(^3\) Each graph Node holds a Permanent reference to be able to access the address of the in-debuggee instance it represents.
foreach referencedValue in GetReferencedValues(value)
    existingNode = GetSeenNode(referencedValue)
    if existingNode != null then MakeEdge(node, existingNode)
    else MakeEdge(node, BuildGraph(referencedValue))

Listing 5 – The final Object graph building algorithm

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

GetSeenNode(value) {
    if no candidates, the object is new
    if some candidates, compare their addresses to o.Address
        if no address equal, o is new // hashcode collision was a coincidence
        if some address equal, o already seen
}

Listing 6 – The final version of GetSeenNode, using .NET hash codes for quick lookup

GetSeenNode only works with addresses of Permanent references, which is safe because getting addresses of Debugger.Values does not resume the debuggee and therefore the addresses are guaranteed to be fixed during the execution of GetSeenNode.

Note: As said in the previous section (.NET hash codes), the algorithm always obtains hash codes by invoking RuntimeHelpers.GetHashCode() in the debuggee. If the algorithm instead used Object.GetHashCode() and user code would override GetHashCode to always return zero (for example), the algorithm would still work. The whole algorithm would, however, run in O(n.E) (n being the number of vertices and E being the number of edges in the object graph) – the same as the original slow algorithm.

---

**Edge routing**

*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):
As discussed in section, after researching available solutions it was decided that a custom solution would be implemented for the purposes of the Object graph visualizer. The implementation presented in this section is not WPF-dependent and can be reused as-is in other applications; this is achieved by following a common practice of programming against interfaces.

**Our algorithm**

The first decision that has to be made when designing an edge routing algorithm is whether the edges will be routed globally or independently (one by one):

- Global routing tries to minimize the total number of edge overlaps while making the individual edge paths look natural.
- One-by-one routing calculates each edge path independently from others, trying to make each edge path look natural.

Like the edge routing algorithm used by Graphviz [28] [29], our algorithm processes edges one by one, treating each edge as separate input. The Graphviz paper [28] also mentions a possibility of a global algorithm that would try to make edges avoid each other or emphasis routing edges in parallel but does not provide any ideas for such algorithm.

When designing the algorithm, two important observations were made:

- The most natural path is a straight line and when there is a rectangle blocking the straight line, the path must be broken around some corners of the rectangle. There is no need to break paths in open space. Therefore, the interesting points in the plane are the corners of the rectangles.
- To make edge paths look natural to users, the edge paths should be relatively close to closest possible paths, while not bending too sharply.

The algorithm works in the following way: First a visibility graph is built, where vertices represent the corners of all the rectangles and edges connect pairs of vertices which can be connected by straight lines without intersecting any rectangles (each edge endpoint is **visible**...
from the opposite edge endpoint). The pairs of vertices where the edge would be blocked by a rectangle are not connected. Then, to find routes for edges, shortest path are found in the visibility graph.

Determine edge start and end points in the plane:

For each edge e in G:
  Determine edge start and end point of edge e:
    take 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 e_s, analogically end point e_t

Build the visibility graph:

Build the following graph $G_v$ (visibility graph):

$V = \{\text{every 4 corners of all rectangles on input}^4\} \cup \{\text{all start and end points of all edges}\}$

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

Find edge routes (each edge independently):

For each edge e in G:
  In $G_v$ find shortest path from $e_s$ to $e_t$ (using A* or Dijkstra’s algorithm)
  Smoothen the path by replacing each sharp join by a bend (join smoothing)

Our implementation has time complexity of $O(n^3)$ where $n$ is the number of rectangles due to the construction of visibility graph: $O(n^2)$ vertex pairs are tested for visibility and each test needs $O(n)$ line-rectangle intersections. The time complexity could be probably reduced but for our needs this is sufficient (note that Graphviz also uses an $O(n^3)$ implementation).

The following figure shows an example of a result of our implementation:

---

$^4$ Note: the vertices of the visibility graph do not lie exactly in the corners of the rectangles but further from the rectangles by a small offset so that the paths go around the rectangles, not directly through their corners.
The implementation is completely reusable and can be found in the namespace Debugger.AddIn.Visualizers.Graph.SplineRouting. The reusability is achieved by defining simple interfaces such as IPoint, IRect, IEdge and programming against them. Any representation of a graph implementing these simple interfaces is then a suitable input to the algorithm.

**Join smoothing**

The last step of the algorithm, join smoothing, is used to replace sharp joins on shortest paths in the visibility graph by smooth bends. As a result, the output is visually much more appealing. The following two figures show a comparison of an edge path with and without join smoothing used:
The principle of the join smoothing method is the following: Given two consecutive line segments, replace the segments by a Bezier curve of order 3 (extended by a straight line segment at each end):

In Figure 5, the tangents of the Bezier curve are aligned with the original lines of the edge path. The distance of the control points (circles in the figure) from the join point controls the smoothness of the curve. The curve continues by a straight line on both ends, following the original lines.

---

**Fundamental problem of debugging**

There are several important facts to realize about debugging in general when object properties are involved. The first fact is that the debugger is actually changing the state of the debuggee by observing it. This is because the only correct way to obtain a value of a property is to invoke its getter, and if the getter has some side effects it can change the state of the program.

Another fact to realize is that it is not always easy to correctly determine the values of all properties of an object. Take, for example, the following class:
```csharp
public class Tricky
{
    int a;
    int b;
    public int A { get { b++; return a; } }
    public int B { get { a++; return b; } }
}
```

**Listing 2 – Class with two property getters incrementing each other**

After creating an instance of this class, debugger tooltips in Visual Studio 2010 show an incorrect state of the instance:

```csharp
public static void Main(string[] args)
{
    Tricky t = new Tricky();
    // Debugger tooltip showing incorrect state
    // A: 0, B: 1
}
```

**Figure 6 – Debugger tooltip in Visual Studio 2010 showing „incorrect“ state of an instance**

The state displayed in Figure 6 definitely does not correspond to reality – the debugger read both of the properties A and B so the actual values of the fields are a=1, b=1. But what are the actual values of properties A and B?

If a value of a property is defined by calling a getter of the property and reading the return value, then reading the value of A changes the value of B and vice versa, which means values of both A and B cannot be determined at the same time. The debugger could get around the problem with values of fields by first evaluating all properties and then all fields (here the evaluation was apparently done in alphabetical order) but the debugger can never display „correct“ values for both properties A and B because such pair of values does not exist.

The problem might seem a little theoretical, but there are very practical scenarios which exhibit similar problems:
The situation in Figure 7 could be solved by first evaluating properties and then fields. But what if there is another property which only reads cachedInstance? Then the value of such property would be shown incorrectly if it were evaluated before evaluating Instance. Next solution would then be evaluating all properties twice, which would incur a large performance overhead (especially concerning the expensiveness of the debugger API) and still would not solve all cases – when there are more instances shown at the same time (watch window, or multiple levels of a debugger tooltip expanded), all of them would have to be updated because a property getter could also change other instances.

Probably the only relatively reasonable solution is evaluating properties first and then fields, which would solve some common scenarios like the one in Figure 7, but such solution would have to evaluate all properties even when they would be out of view (scrolled away) which would again incur mostly unnecessary performance overhead. As seen in the screenshot, current debuggers do not try to solve these problems – the user has better understanding of the code and can reevaluate relevant parts as needed.

**NRefactory**

NRefactory is a standalone library providing object model for C# and VB code. NRefactory is important to this thesis because the debugger is using NRefactory extensively to represent expressions to be evaluated.
NRefactory contains a lexer and a parser of C# and VB languages. The main use of NRefactory is parsing of source code and providing a Syntax tree of the code. There are many possible types of nodes in the syntax tree, for example ForStatement, SwitchStatement, MemberReferenceExpression, BinaryOperatorExpression etc. For example the following code snippet is parsed into the following syntax tree:

```csharp
var foo = Foo.Size + 2;
```

**Listing 1 - Sample code to be parsed by NRefactory**

```csharp
VariableDeclaration(Name=foo)
  Initializer=
    BinaryOperatorExpression(op=Add)
      MemberReferenceExpression(MemberName=Size)
        IdentifierExpression(Identifier=Foo)
        PrimitiveExpression(Value=2)
```

**Figure 8 – Parse tree for the the code snippet from Listing 9**

The important fact to realize is that the syntax tree does not contain any semantic information – for example it is not known whether `Foo` is a class, a local variable, or a property.

**ICSharpCode.SharpDevelop.Dom**

In the previous section it was stated that the NRefactory syntax tree does not contain any semantic information – in Figure 8 it can be seen that `Foo` is an Identifier but no more information is provided – there is no information about whether it is a class, a property, a field or a local variable.

Of course, the IDE needs a lot of semantic information about the code to provide features such as Go to definition, Find references and Code completion. The semantic understanding of the code is provided by ISharpCode.SharpDevelop.Dom. The Dom provides an object model for representation of classes, methods, types, parameters etc. An important part of the Dom are resolvers (that is implementations of IResolver). Resolvers accept the identifier `Foo` from our example plus context (position in the source code file) and return information telling whether the symbol is a class, a property or other symbol, where to find its definition etc.

In order for the resolvers to be able to work correctly, the Dom representation of all the symbols in the currently open solution is needed. When a solution is opened, SharpDevelop reads and parses all the solution files to build an initial Dom representation. When files are being edited, SharpDevelop is reparsing the files in background and updating the Dom representation.

ICSharpCode.SharpDevelop.Dom depends only on NRefactory and is therefore reusable outside of SharpDevelop. The principle of programming against interfaces is strictly followed, so for example PythonClass (coming from the IronPython backend binding) implements Dom.IClass and the code in Dom can work with it without knowing anything about Python.

---

The lexer and parser are kept up to date with most recent language specifications changes.
Similarly, concrete implementations of IResolver provide resolvers implementing the semantic rules of C#, VB, Xaml and the other supported languages.

**Future of NRefactory and Dom**

In retrospect, the decision to split the syntactic and semantic representation of code into NRefactory and Dom was not the best one – there could be a unified representation bringing the facilities of NRefactory and Dom together. Also, NRefactory was designed when C# and VB were identical languages differing only in syntax. However, the features of C# and VB cannot be mapped 1:1 anymore. As a result, new version of NRefactory has been designed to be used in the future versions of SharpDevelop.

There is also a very interesting initiative from Microsoft to open the internals of their compiler which could be very useful for SharpDevelop and many other tools dealing with source code. The problem with the current C# compiler is that it acts as a black box, consuming source code and producing binary output. In the process of turning the source code into bytecode, the compiler builds large amount of very useful semantic information about the code, and then throws it away.

IDEs and other tools have to re-implement the logic that is already present in the black-box compiler – NRefactory contains a full C# parser, the resolvers implement type inference and overload resolution exactly according to the C# language specification etc. All these features are present in the compiler but not accessible (yet). In the next version of NRefactory, the parser is shared with the Mono compiler.

Note that the Scala programming language already has an open compiler (called Presentation compiler) exposing all its understanding of the code so for example the Scala IDE for Eclipse and Ensime IDE for Emacs are both using the Scala Presentation compiler.