Sample of the usage of the debugging API

Assume the you 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 breakpoint is hit if it did not
// already happen.
process.WaitForPause();

// The breakpoint hint 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 will produce the following messages:

- Breakpoint hit
- Current method = Program.Main
- message = Hello World! (type: String)

Fundamentals

There are two processes involved in the debugging process:

- **The debugger** - This is the process that controls the debugging and issues the debugging commands - for example, it starts the process, sets breakpoints and inspects the local variables. In this case the debugger is SharpDevelop.
- **The debuggee** - This is the process that is being debugged. It is the application you are investigating using the debugger - for example, the "Hello world" program.

The debuggee can be in two states - **running** or paused. Initially the debuggee is running, but it will pause if any significant event occurs - for example, if breakpoint is hit or if exception is thrown. Once the debuggee is paused, the debugger can investigate the program state (eg. read the local variables). It is not possible to investigate the debuggee while it is running.

To be notified when the debuggee gets paused, subscribe to the events `Process.Paused` and `Process.ExceptionThrown`.

Variables and values

The debugger and debuggee are executed in two different processes. Therefore the debugger can not obtain a direct reference to any object living in debuggee. It has to investigate the variables indirectly. The class `Value` is provided for this purpose. You can obtain an instance of the `Value` class by, for example, calling `StackFrame.GetLocalVariableValue(string name)`. Once you have the value, you can investigate it - for example, get the type of the value. If the value is a primitive type like string or integer, you can get the actual content using the `PrimitiveValue` property. If the value is a class, you can get the values of its fields or properties.

Is not possible to investigate the variables when the debuggee is running. Fist of all, the values can be changed by the running program. Even more importantly, the values can be deleted or moved by the garbage collector at any time. Therefore whenever the debuggee is resumed into the running state, all variables become invalid. It is impossible to know where the variable is in the memory anymore. This is limitation of the .NET Runtime at there is nothing the debugging API can do about it. The next time the debuggee is paused, the debugger has to obtain all values again.

To make this process of obtaining values again easier, this library provides "expressions". Expression is the strongly-typed tree that remembers how each value was obtained. For example, it would remember that you have obtained the value by calling:
`StackFrame.GetLocalVariableValue("shape").GetPropertyValue("size").GetFieldValue("width")`
You could also image this as the expression `"shape.size.width"`, although in reality the tree contains more information then just the member names (in particular, it contains the types). Each value stores the expression that was used to obtain it in `Value.Expression`. However, note that if you reevaluate the expression, you might get reference to a different object then before. For example, if the program assigns the variable `shape` a new value then you will obtain reference to this new object, not to the one you have obtained before. Make sure this is the intended behavior. In would be in theory possible, to design the `Value` class so that it would use the expressions to reobtain its value automatically, however this design was intentionally rejected because of the just mentioned behavior. The other reason for rejection of this design is performance. The programmer is forced to do the evaluations manually so that he is aware of their performance cost and side-effects.