PluginInfrastructure Code Review

abeham, 2010-02-07

Summary

The PluginInfrastructure consists of 3 classes and 5 attributes to describe plugins:

- **PluginDescription**, **ApplicationDescription**, and **PluginFile**
- **PluginAttribute**, **ApplicationAttribute**, **PluginFileAttribute**, **PluginDependencyAttribute**, and **AssemblyBuildDateAttribute**

There are 5 classes for “working with” the plugins:

- **PluginManager**, **PluginValidator**, **ApplicationManager**, **InstallationManager**, and **SandboxManager**
- **ControlManager** is considered obsolete as there is no reference to it

There are 2 classes that provide a GUI:

- **StarterForm**, and **SplashScreen**

Together with the **Main** class this can probably be seen as the core of the PluginInfrastructure.

PluginManager creates a temporary app domain to load all assemblies in the reflection-only context, checks dependencies and that all necessary files are there (actually it will instantiate a **PluginValidator** in the temporary app domain that will do the checking). After it has initialized it can create new app domains for starting applications. **ApplicationManager** is a singleton in each app domain providing services to retrieve and filter types from all types loaded. **InstallationManager** is used to install, remove and update plugins. **StarterForm** is the GUI that allows users to start applications through the call to **PluginManager.Run()**.

Remarks

Code to determine why a certain plugin can’t be enabled is in **PluginValidator** and **InstallationManager**. Possibly the **PluginDescription** can be enriched with a list of error message strings that are filled by **PluginValidator**, maybe error codes could be used as well. **PluginValidator** has more ways to declare why loading a plugin failed in **CheckPluginAssemblies()** by passing on the exceptions it catches. This would probably be more helpful in determining what went wrong.

Static Code Analysis

In this section the development of the PluginInfrastructure from February 2008 to February 2010 is shown in the form of changes over time. The freeware program SourceMonitor was applied on several snapshots of the code. The PluginInfrastructure was refactored between the snapshot of November 11th, 2009 (Rev 2481) and January 5th, 2010 (Rev 2600). Actually the snapshot of November 11th, 2009 in the following charts marks the state of August 5th, 2009. Only the code in the trunk has been analyzed, branches have not been looked into. Designer generated files are excluded in the analysis.

Maximum and Average Complexity

Here is what SourceMonitor says about how these metrics are calculated:

- **Maximum Complexity:** This is the complexity value of the most complex method in a file.
- **Average Complexity:** The average complexity is computed as a simple average complexity over all complexity values measured for a file or checkpoint (snapshot).

Regarding the **complexity value** the help file of SourceMonitor says:

> *"The complexity metric is counted approximately as defined by Steve McConnell in his book Code Complete, Microsoft Press, 1993, p. 395. The complexity metric measures the number of execution paths through a function or method. Each function or method has a complexity of one plus one for each branch statement such as if, else, for, foreach, or while. Arithmetic if statements (MyBoolean ? ValueIfTrue : ValueIfFalse) each add one count to the complexity total. A complexity count is added for each "&&" and "||" in the logic within if, for, while or similar logic statements. Switch statements add complexity counts for each exit from a case (due to a break, goto, return, throw, continue, or similar statement), and one count is added for a default case even if one is not present. Each catch or except statement in a try block (but not the try or finally statements) each add one count to the complexity as well."*

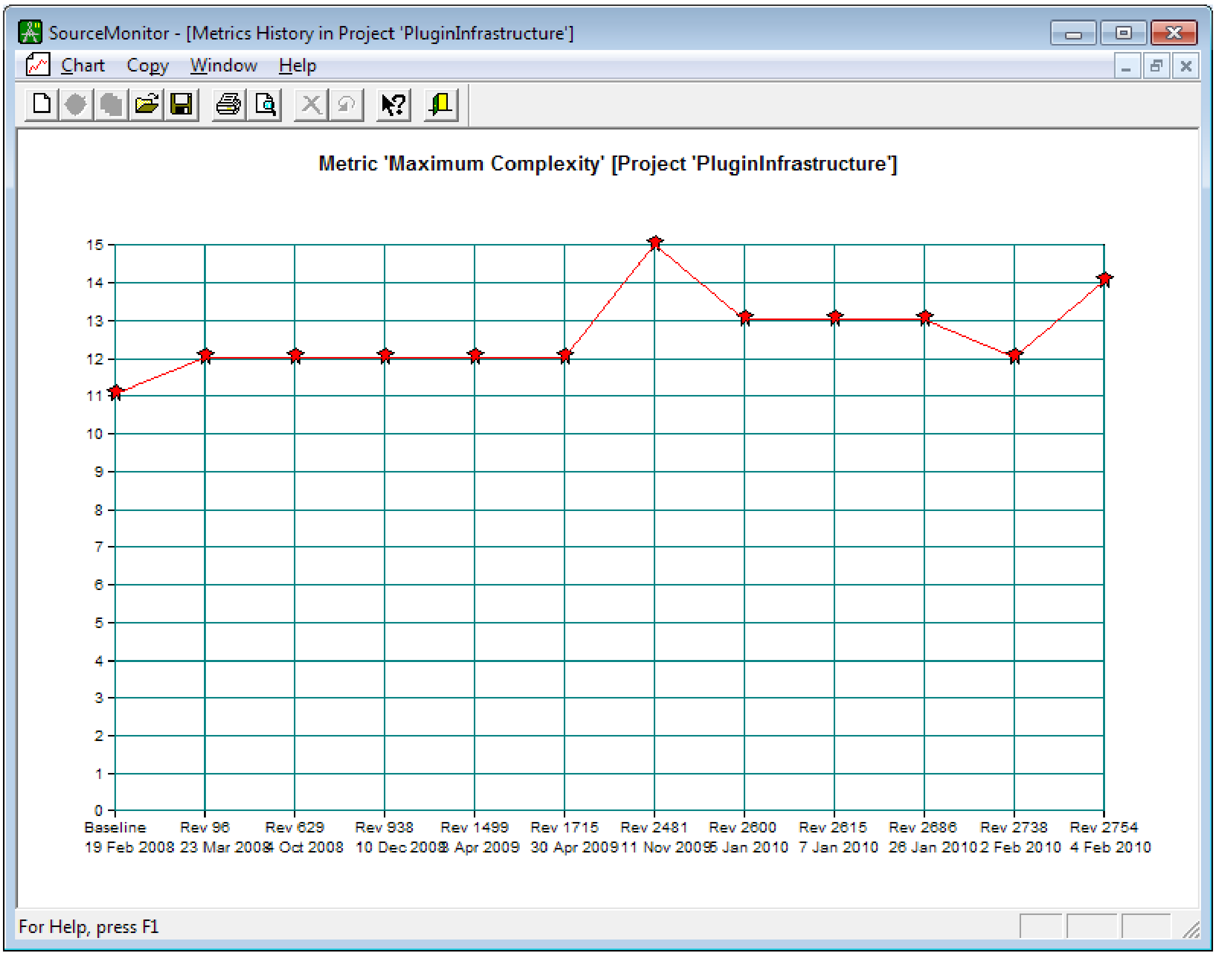

Figure 1 Changes to the Maximum Complexity metric over various snapshots

The Maximum Complexity metric displays a steep increase before the refactoring which reduced maximum complexity again. The most recent changes suggest that complexity is again rising. The rise of Maximum Complexity before the refactoring was caused by changes in **Loader.LoadPlugins()**. After the refactoring in r2600 the most complex methods were **PluginValidator.GetPluginDescription()** which stood out among all methods in **PluginValidator** in the number of Statements[1] (34 compared to the next highest: 14) and Calls[2] (32 compared to the next highest: 13) as well. The reduction of Maximum Complexity between January 2010 and February 2010 (r2686 and r2738) was achieved because of changes to **GetPluginDescription()** which reduced the Complexity to 12, the number of Statements to 30 and the number of Calls to 29. The classes **Main** and **InstallationManager** also show a Maximum Complexity of 12 in this snapshot, so further reductions in Maximum Complexity would need to involve more classes. The final rise to a Maximum Complexity of 14 is again due to **GetPluginDescription()** where the number of Statements also rose

---

[1] Computational statements are terminated with a semicolon character. Branches such as if, for and while are also counted as statements. Methods, though not terminated by semicolons, are counted as statements. All attributes are counted as statements as well, though calls inside attributes are ignored. The exception control statements try, catch, and finally are also counted as statements. Preprocessor directives #define and #undef are counted as statements. All other preprocessor directives are ignored. In addition all statements between each #else or #elif statement and its closing #endif statement are ignored, to eliminate fractured block structures. Goto labels are also counted as statements.

[2] The total number of calls to other methods or functions found inside each method or function. This is sometimes called fan out.

to 38, but the number of Calls dropped from 29 to 28. **Main** and **InstallationManager** did not change in Maximum Complexity.

It should be noted that the only “complex” method in **Main** is **Run()**. It contains an IF statement to determine if it was invoked with arguments and a switch statement for each possible argument. The most complex methods in **InstallationManager** are **DetermineProblem()** and **Update()**.

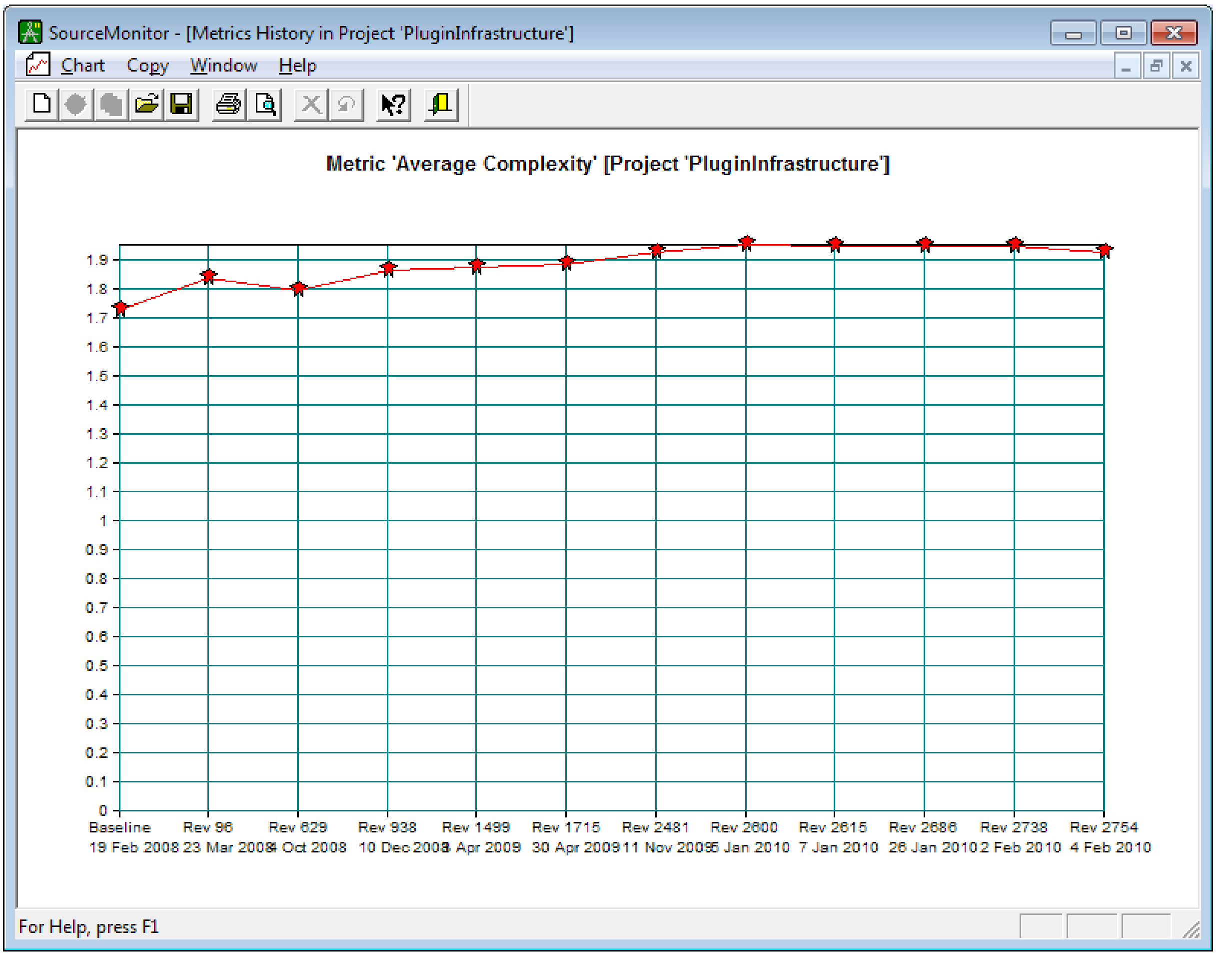

Figure 2 Changes to the Average Complexity metric over various snapshots

The rise in Average Complexity from r1715 (April 2009) to r2481 (Nov 2009) was due to an increase in Average Complexity in: **DiscoveryService** (2.67 -> 2.9) and **Loader** (2.97 -> 3.13) and a slight decrease in Average Complexity in: **PluginManager** (1.84 -> 1.74) and **Runner** (1.8 -> 1.71). Unfortunately the further increase after the refactoring is hard to explain without knowing more about which code went into which classes, but the most complex classes on average in r2600 after the refactoring were: **Main** (4.0), **PluginDescriptionIterator** (3.67), **PluginValidator** (3.6), and **InstallationManager** (3.12) . The slight reduction in Average Complexity towards the last snapshot is because of a reduction in **InstallationManager** (3.12 -> 2.89) and an increase in **PluginAttribute** (1.0 -> 1.5).

Statements and Statements/Method

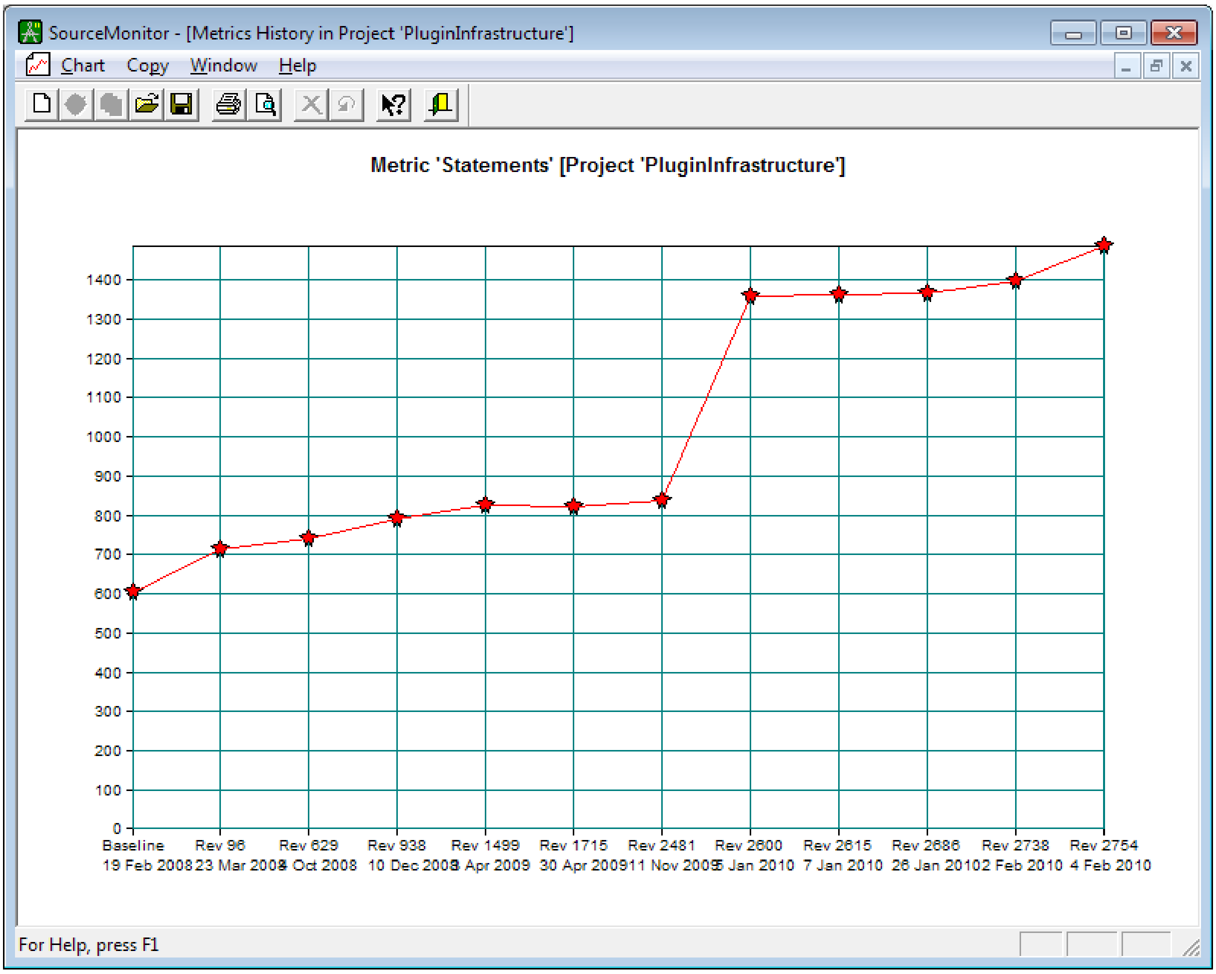

Figure 3 Changes to the number of Statements over various snapshots

There was a 62% increase in Statements after the refactoring. In the last snapshot the files with most Statements are: **PluginValidator.cs** (226), **InstallationManager.cs** (139), **ApplicationManager.cs** (106), and **StarterForm.cs** (105).

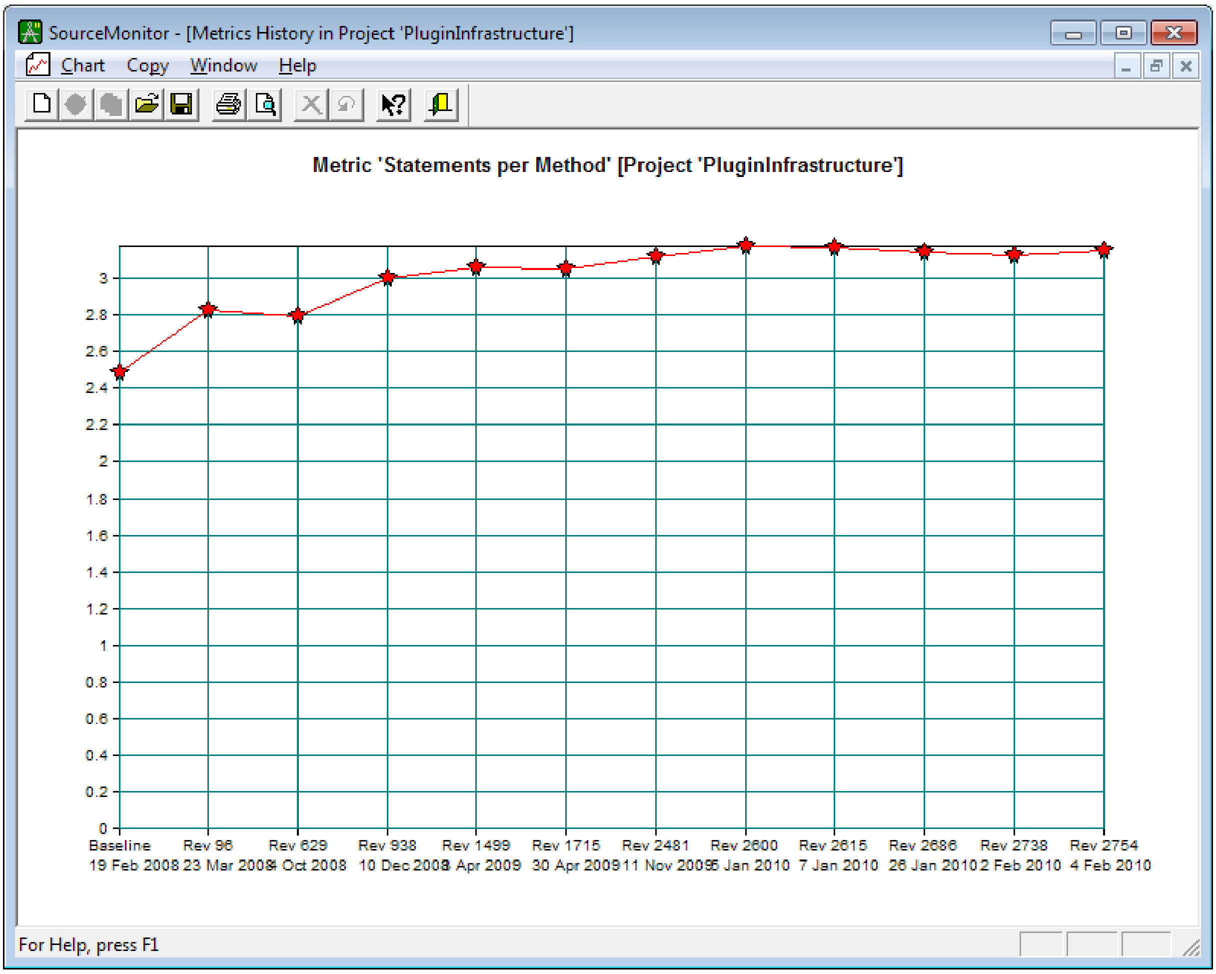

Figure 4 Changes to Statements per Method over various snapshots

Statements per Method has increased over the course of the development, but as a whole has been relatively stable over the last snapshots. In the last snapshot the files with most Statements per Method are: **Main.cs** (14.25), **SandboxManager.cs** (7.67), **DescriptionIterator.cs** (6.33), **PluginValidator.cs** (6.07), **StarterForm.cs** (5.77), **PluginManager.cs** (5.55), **InstallationManager.cs** (5.53), and **SplashScreen.cs** (5.38). This hasn't changed significantly since merging the refactored branch though **StarterForm.cs** has increased slightly (5.38 -> 5.77).

Classes, Interfaces, and Structs

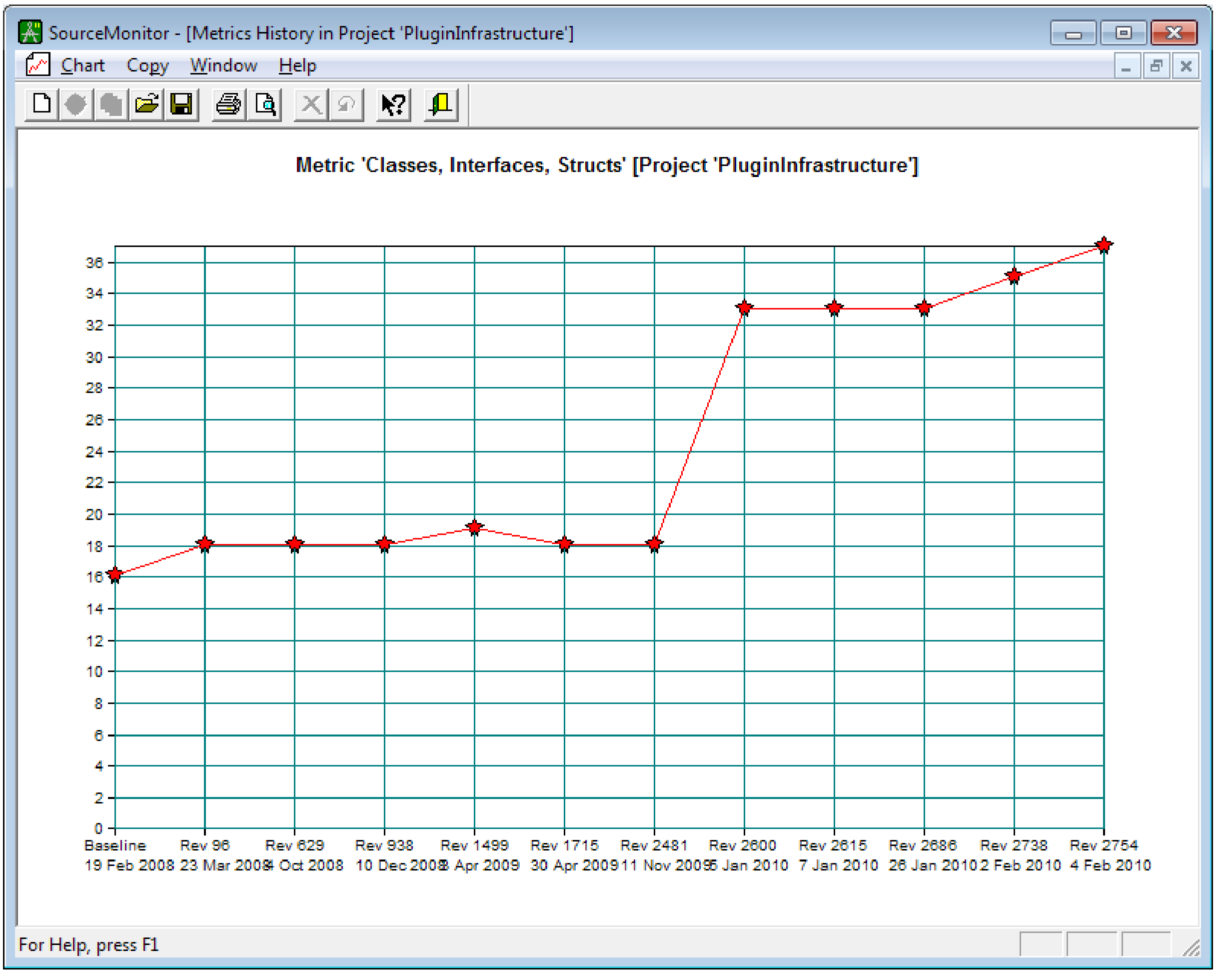

Figure 5 Changes of the number of classes, interfaces, and structs over various snapshots

Refactoring the PluginInfrastructure increased the number of classes, interfaces, and structs by 83%. Meanwhile this number has doubled to what it has been before the refactoring. The last additions since r2686 (January 2010) were: **IPluginFile**, **PluginFile**, **InstallationManagerForm**, and **PluginValidator.PluginDependency** (a private class).

Methods/Class and Calls/Method

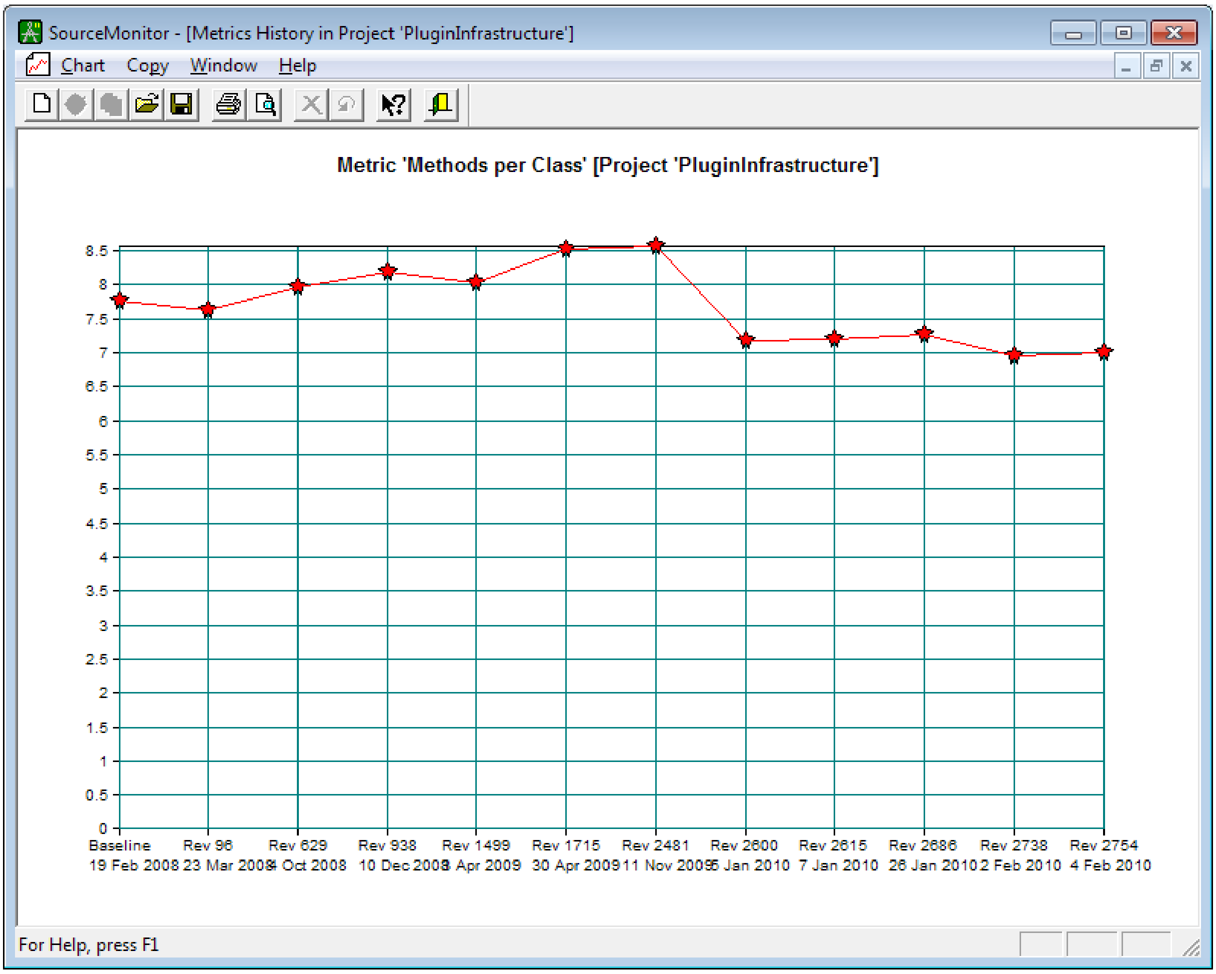

Figure 6 Changes to Methods per Class over various snapshots

After the refactoring the amount of Methods per Class[3] decreased by 16%. In the last snapshot the files with the highest amount of methods are: **ApplicationManager.cs** (27), **PluginDescription.cs** (23), and **InstallationManager.cs** (19).

---

[3] Methods per Class: This metric is an overall average for all class, interface and struct methods in a file or checkpoint, computed as the total number of methods divided by the total number of classes, interfaces and structs.

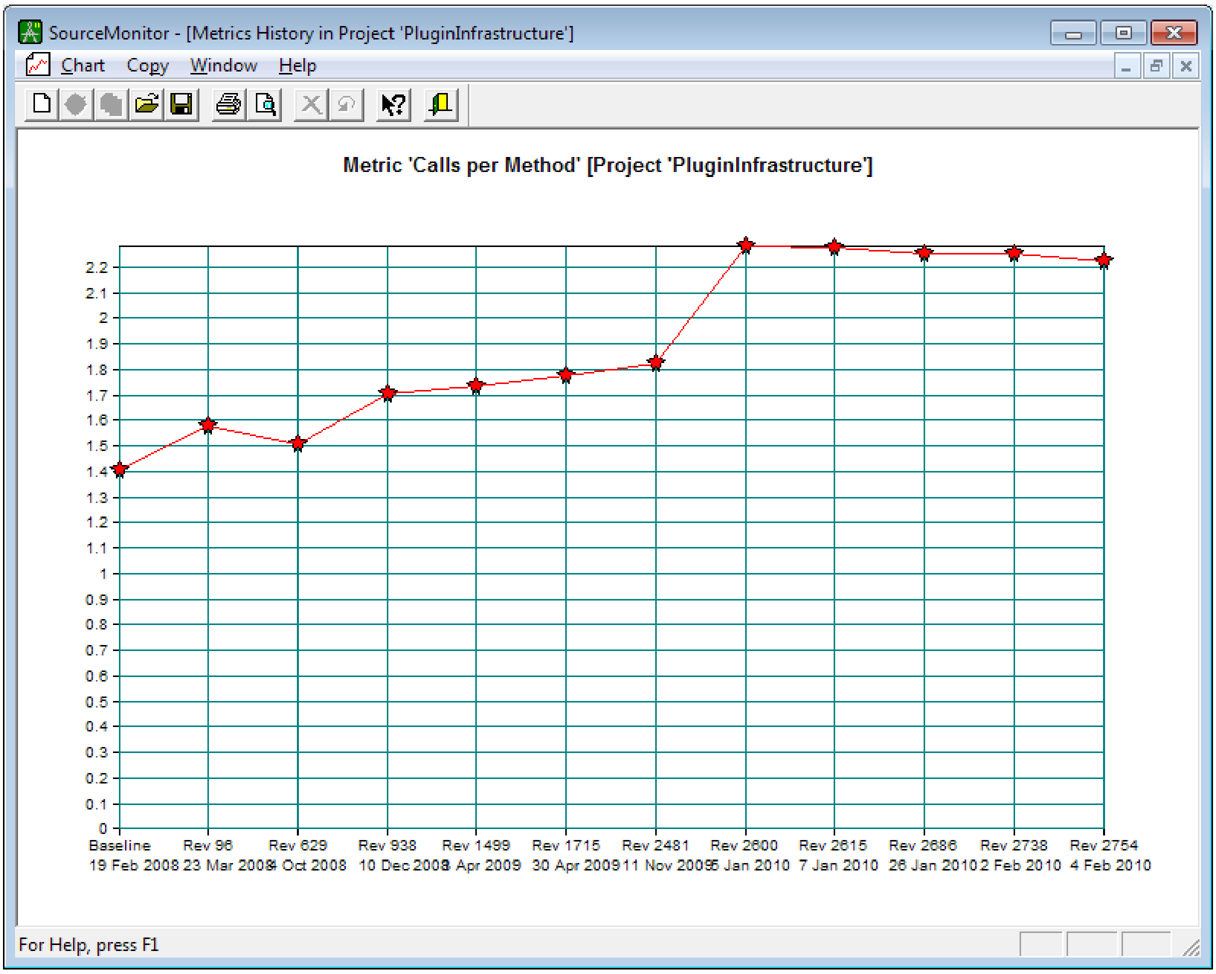

Figure 7 Changes to the amount of Calls per Method over various snapshots

There has been an overall increase of Calls per Method[4] during the time, although after the refactoring the tendency is towards a slight decrease. Unfortunately the metric doesn't seem to differ between calls to methods of the same class and calls to methods of a different class. In the last snapshot the files with the highest amount of Calls per Method are: **Main.cs** (9.75), **SandboxManager.cs** (7.0), and **InstallationManager.cs** (5.68).

---

[4] Calls per Method: The total number of calls to other methods found inside of all methods found in a file or checkpoint divided by the number of methods found in the file or checkpoint.

Code issues

StarterForm.cs

```
      public StarterForm(string appName)
      : this() {
      var appDesc = (from desc in pluginManager.Applications
                     where desc.Name == appName
                     select desc).Single();
      if (appDesc != null) {
        StartApplication(appDesc);
      } else {
        MessageBox.Show("Cannot start application " + appName + ".",
                        "HeuristicLab",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Warning);
      }
    }
```

- Unnecessary null check. Quote from MSDN:

  > The SingleTSource(IEnumerableTSource, FuncTSource, Boolean) method throws an exception if the input sequence contains no matching element. To instead return **null** when no matching element is found, use SingleOrDefault.

ControlManager.cs and IControlManager.cs

- They're probably obsolete, there exists no class implementing IControlManager

PluginDependencyAttribute.cs

```
/// <summary>
  /// This attribute can be used to declare that an plugin depends on a
another plugin.
  /// </summary>
public sealed class PluginDependencyAttribute : System.Attribute
```

- In XML comment: "an plugin" should be „a plugin"

PluginAttribute.cs

```
public PluginAttribute(string name)
      : this(name, "") {
}
```

- Should use String.Empty instead of ""

```
public PluginAttribute(string name, string description) {
      this.name = name;
      this.description = description;
}
```

- ~~Should also include sanity checks similar to the other attributes 2010-02-02~~
- Actually it doesn't need to include sanity checks (PluginValidator checks them and loads only those plugins that have a non-empty name). 2010-02-07

PluginBase.cs

```
/// <inhertitdoc>
public virtual void OnLoad() { }
/// <inhertitdoc>
public virtual void OnUnload() { }
```

- Should be “<inheritdoc />”
- PluginBase and IPlugin do not expose a Description property: What is the reason for specifying a Description in PluginAttribute?

StarterForm.cs

```
  private void applicationsListView_ItemActivate(object sender, EventArgs
e) {
   if (applicationsListView.SelectedItems.Count > 0) {
    ListViewItem selected = applicationsListView.SelectedItems[0];
    if (selected == pluginManagerListViewItem) {
     try {
      //Cursor = Cursors.AppStarting;
      //ManagerForm form = new ManagerForm();
      //this.Visible = false;
      //form.ShowDialog(this);
      //// RefreshApplicationsList();
      //this.Visible = true;
     }
     finally {
      Cursor = Cursors.Arrow;
     }
    } else {
     ApplicationDescription app =
(ApplicationDescription)applicationsListView.SelectedItems[0].Tag;
     StartApplication(app);
    }
   }
  }
```

- The else branch is the only really relevant branch as it seems

ApplicationManager.cs

```
  /// <summary>
  /// Finds all types that are subtypes or equal to the specified type if
they are part of the given
  /// <paramref name="plugin"/>.
  /// </summary>
  /// <param name="type">Most general type for which to find matching
types.</param>
  /// <param name="plugin">The plugin the subtypes must be part
of.</param>
  /// <param name="onlyInstantiable">Return only types that are
instantiable (instance, abstract... are not returned)</param>
  /// <returns>Enumerable of the discovered types.</returns>
  internal static IEnumerable<Type> GetTypes(Type type,
IPluginDescription pluginDescription, bool onlyInstantiable) {
```

- Pointless, because an internal method, but the XML comment for onlyInstantiable says […] that are instantiable (instance, abstract...)[…] “instance” should probably mean “interface”

PluginValidator.cs

```
internal void OnPluginLoaded(PluginInfrastructureEventArgs e) {
   if (PluginLoaded != null)
    PluginLoaded(this, e);
  }
```

- Should be a private method (it isn’t called from outside PluginValidator)

PluginDescription.cs

```
if (!(pluginState == PluginState.Enabled || pluginState ==
PluginState.Loaded))
        throw new InvalidOperationException("Can't loaded a plugin in state
" + pluginState);
```

- Typo: Can’t load**ed** a plugin…