JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

# Metrix - A Measuring Tool for Run-time Figures in Plug-in based .NET Applications

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Rainer Pichler, 0555853

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Mag. Reinhard Wolfinger

Hagenberg im Mühlkreis, Oktober 2009

**Metrix**
**A Measuring Tool for Run-time Figures in Plug-in based .NET Applications**

**Abstract**

*Advocates of plug-in based architectures claim that plug-in components make applications customizable. They argue that customizable applications are smaller and easier to use. This bachelor thesis presents a system of metrics to evaluate those claims for the composition framework Plux.NET. The plug-in component Metrix integrates into the Plux.NET composition framework and provides those metrics. It measures how many of an application's components are active and how this number changes when the application adapts to a new working context. Metrix visualizes metrics at run-time and can log metrics for off-line analysis.*

# Table of Contents

1 Overview....................1
2 Using Metrix....................2
  2.1 Installing Metrix....................2
  2.2 Querying metrics....................3
  2.3 Logging metrics....................5
  2.4 Metrix Documentation....................8
3 Metrix Design Goals....................11
  3.1 Structure of Metrix....................11
  3.2 Behavior of Metrix....................13
4 Developing with Metrix....................14
  4.1 Interfaces....................14
  4.2 Using Metrix in Applications....................16
  4.3 Extending Metrix....................19
  4.4 UI Controls....................28
5 Further Work and Discussion....................33
  5.1 Derived metrics must be compiled....................33
  5.2 Limited analysis of memory usage....................33
6 References....................34

# 1 Overview

The composition framework Plux.NET is designed for Microsoft .NET and aims at building lightweight applications. It consists of a minimal core and is extensible through plug-ins. Whether such architectures allow to build more lightweight applications than traditional approaches, is an open question.

This bachelor thesis discusses the Plux.NET plug-in Metrix. Metrix collects run-time metrics about Plux.NET applications. Researchers use these metrics to evaluate the claim of being lightweight. This work is about how to aquire the data with Metrix but not about the interpretation of those data. Further information about the plug-in platform Plux.NET can be obtained from http://ase.jku.at/plux.

To achieve the task of collecting metrics, Metrix consists of four parts: Firstly, Metrix includes a set of so-called figures, which provide more than 40 metrics at different granularity levels. The *Filesize* metric for example is collected separately for each assembly as well as for the Plux.NET application as a whole. Metrix can generate a HTML documentation with an overview of available metrics. Secondly, Metrix integrates in a command line tool and allows the user to query metrics in a text-based manner. Thirdly, Metrix includes four user controls to display metrics in Windows Forms applications. Finally, the plug-in includes an extensible logger for saving observed metrics.

The remainder of this bachelor thesis consists of the following parts: Firstly, section 2 gives an overview on how to work with Metrix. Then, section 3 explains the structure and behavior of Metrix and therefore gives a deeper understanding of the features explained in the second section. After that, section 4 discusses how developers can extend Metrix further through new plug-ins. It also shows how they can use metrics in their applications. Finally, section 5 summarizes and outlines further work.

# 2 Using Metrix

This section shows how non-developers can use Metrix. Section 4.2 (Using Metrix in Applications) discusses how developers can integrate Metrix in other applications.

Firstly, this section explains how to install Metrix. Secondly, it discusses how users can control Metrix through a command line interface. Thirdly, it describes how users can log metrics to a file. Fourthly, it gives an overview over the available metrics.

## 2.1 Installing Metrix

Plux.NET can be downloaded from http://ase.jku.at/plux/downloads. To use Metrix, a minimal Plux.NET environment must be set up by copying the following files in an arbitrary folder:

| File | Description |
|---|---|
| Metrix.Contracts.dll<br>Plux.Contracts.dll<br>Scripting.Contracts.dll | These assemblies contain interface definitions which describe the capabilities of components. |
| IsolatedStoragePersistor.dll<br>Plux.Framework.dll<br>Preferences.dll | These components allow to persist settings. They are used by the Plux.NET command line interface and the Metrix logger. |
| LogWriter.dll<br>Metrix.dll<br>MetrixCmdlets.dll | The Metrix core components. |
| Plux.Client.dll<br>Plux.dll<br>Runtime.exe | The core components which make up the Plux.NET composition framework. |
| ConsoleNG.dll<br>Scripting.dll | The Plux.NET command line interface and its script interpreter back-end. |

*Table 1: A minimal Plux.NET environment for Metrix.*

Launching *Runtime.exe* starts up the Plux.NET environment. At startup, the software component infrastructure scans the working directory for components and interface definitions. When it encounters a component which implements the *IStartup* interface, it instantiates the component and calls the component's *Run* method. The component *ConsoleNG.dll* implements *IStartup* and displays a command line interface at startup. This component is also referred to as Plux.NET console.

The Plux.NET console allows the user to control the Plux.NET composition framework by typing in commands. Since the Plux.NET console is based on Microsoft PowerShell (see also *Windows PowerShell*), Microsoft PowerShell must be installed in order to use the Plux.NET

console. The Plux.NET console commands allow to control the composition of components and to query information about components. Furthermore it can be extended with additional commands through other components like Metrix. For familiarity, the naming scheme of the commands follows the verb-noun pattern of Microsoft PowerShell (see also *Learning Windows PowerShell Names*).

## 2.2 Querying metrics

Metrix integrates into the Plux.NET console which supports the Microsoft PowerShell object pipeline. The object pipeline allows commands to pass on information to other commands as objects instead as in text (see also *Piping and the Pipeline in Windows PowerShell*). Therefore, Metrix commands can be combined with existing commands.

The *get-measurement* command retrieves metrics:

```
get-measurement [-Scope] [-FigureName] [-Item]
```

The optional parameters narrow the set of returned metrics: Firstly, a *Scope*, which filters the measured elements' type, can be set. Possible values for *Scope* are: *ExtensionInfo*, *SlotInfo*, *PlugInfo*, *ExtensionTypeInfo*, *SlotTypeInfo*, *PlugTypeInfo*, *ContractInfo*, *SlotDefinition* and *Runtime*.

Secondly, *FigureName* can be used to filter metrics of interest. Please consider subsection 2.4 (Metrix Documentation) for available metrics. Both parameters support wildcards. Thirdly, an *Item* for which to print metrics can be given. The *Item* parameter can be assigned through the object pipeline.

The following statement gives an overview of the current state of the Plux.NET application:

```
get-measurement -Scope Runtime | format-measurement
```

Above statement will return all metrics for the Plux.NET application, like its uptime or the number of loaded extensions. The result is passed on to *format-measurement*, which creates a table representation. For each metric, one *Current* value is provided. Many metrics also have a *Total* counter which increases whenever *Current* increases. *Total* counters never decrease. For example, the *Current* value of zero for the metric *queueitems* indicates that the Plux.NET task queue is idle at the moment. Additionally, the *Total* counter shows that Plux.NET has processed 204 tasks since Metrix has started to monitor. Whenever a task is enqueued, *Current* and *Total* increase. When a task is dequeued, *Current* decreases but *Total* stays the same.

```
Item    FigureName                      Current         Total
Runtime bytesinuse                      7093036
Runtime closedslotinfos                 0               8
Runtime contractinfos                   5               5
Runtime extensioninfos                  62              62
Runtime extensiontypeinfos              163             163
Runtime filesize                        819712
```

```
Runtime maxqueueitems                110
Runtime openslotinfos                35              35
Runtime pluggedextensioninfos        82              82
Runtime pluggedplugs                 62              123
Runtime pluginfos                    64              64
Runtime plugininfos                  19              19
Runtime plugtypeinfos                172             172
Runtime queueitems                   0               204
Runtime registeredextensiontypeinfos 147             147
Runtime shared                       59              59
Runtime slotdefinitions              29              29
Runtime slotinfos                    35              35
Runtime uptime                       00:00:22.0817520
```

*Listing 1: Querying application-wide metrics (shortened output).*

The query above also shows that 62 extensions are loaded. However, it does not tell which plug-ins they belong to. To answer this question, the following statement changes the *Scope* to *PluginInfo*. Additionally, it just selects the metric *extensioninfos* by specifying the *FigureName*:

```
get-measurement -Scope PluginInfo -FigureName extensioninfos | format-
measurement
```

Now, the output shows that Metrix alone loaded 47 extensions. Note that the total number of extensions increased to 63. This happened because Metrix created a new extension which provides the *extensioninfos* metric for each plug-in:

```
Item                                    FigureName     Current Total
PluginInfo{1:Plux}                      extensioninfos 1       1
PluginInfo{2:Plux.Framework.dll}        extensioninfos 1       1
PluginInfo{3:Workbench.dll}             extensioninfos 1       1
PluginInfo{4:Cerberus.dll}              extensioninfos 1       1
PluginInfo{5:Console.dll}               extensioninfos 0       0
PluginInfo{6:ScriptRunner.dll}          extensioninfos 1       1
PluginInfo{7:ConsoleDiscovery.dll}      extensioninfos 0       0
PluginInfo{8:ConsoleNG.dll}             extensioninfos 1       1
PluginInfo{9:ControlsApp.dll}           extensioninfos 0       0
PluginInfo{10:HotViz.dll}               extensioninfos 0       0
PluginInfo{11:IsolatedStoragePersistor.dll} extensioninfos 1   1
PluginInfo{12:LayoutManager.dll}        extensioninfos 0       0
PluginInfo{13:LogWriter.dll}            extensioninfos 0       0
PluginInfo{14:MenuStrip.dll}            extensioninfos 1       1
PluginInfo{15:Metrix.dll}               extensioninfos 47      47
PluginInfo{16:MetrixCmdlets.dll}        extensioninfos 3       3
PluginInfo{17:Preferences.dll}          extensioninfos 1       1
PluginInfo{18:PropertyDialog.dll}       extensioninfos 0       0
PluginInfo{19:Scripting.dll}            extensioninfos 4       4
```

*Listing 2: Querying the number of extensions per plug-in.*

Alternatively, the command *display-measurement* can be used too. It generates a table itself, therefore the *format-measurement* command can be omitted. Because it is a convenience function with fixed parameters, unused parameters must be omitted with the '*' wildcard. Therefore, the previous statements can be written in a shorter form:

```
get-measurement -Scope Runtime | format-measurement
display-measurement Runtime *

get-measurement -Scope PluginInfo -FigureName extensioninfos | format-
measurement
display-measurement PluginInfo extensioninfos
```

Finally, *get-measurement* can also display measurements for a specific item. The following statement prints all metrics for the extension *Workbench*:

```
plux> get-extension Workbench | get-measurement | format-measurement
Item                        FigureName         Current Total
ExtensionInfo{3:Workbench} closedslotinfos    0       0
ExtensionInfo{3:Workbench} openslotinfos      6       6
ExtensionInfo{3:Workbench} pluggedinbound     3       3
ExtensionInfo{3:Workbench} pluggedoutbound    1       1
ExtensionInfo{3:Workbench} pluginfos          1
ExtensionInfo{3:Workbench} registeredinbound 26       26
ExtensionInfo{3:Workbench} slotinfos          6
```

*Listing 3: Querying all metrics for a specific extension.*

This is accomplished by utilizing the object pipeline. The *get-extension* command passes on the according extension which is implicitly assigned to the *Item* parameter of *get-measurement*.

As seen before, most loaded extensions belong to Metrix. Metrix keeps alive all figure extensions which have a *Total* counter unless the user decides to release them. This is necessary because the total counter would reset with each query otherwise. The command *free-figure* can be used to release all figures held back by Metrix.

## 2.3 Logging metrics

To analyze metrics at a later time, Metrix includes a logger extension called *MetrixLogger*. It can log metrics at intervals as well as when an observed metric changes. For flexibility, the logger is not bound to a specific output format. Instead, it allows extensions which implement the *LogWriter* plug to conduct the further processing themselves. Metrix ships with three *LogWriter* extensions: Firstly, *CSVLogWriter* saves the data comma separated into a text file. The output file can then be further processed with third party software, for example a spreadsheet application. Secondly, *HTMLLogWriter* can be used to view the logged metrics in a browser. Thirdly, *DebugLogWriter* helps developers to understand when which methods of a *LogWriter* are called.

The Plux.NET console command *start-metrixlogger* starts the logger. It will create and configure the *MetrixLogger* and a *LogWriter* extension. It supports the parameters described in table 2. Note that at least one of the options *OnEvents* or *Interval* must be set because otherwise no output will be generated.

| Parameter name | Description |
|---|---|
| LogWriter | The parameter *LogWriter* is mandatory and represents the name of the *LogWriter* extension. |
| Measurements | The parameter *Measurements* is mandatory and describes the metrics to log. |
| File | The parameter *File* is optional and specifies the file the *LogWriter* writes into. |
| OnEvents | This switch tells the log writer to log whenever an observed metric changes. |
| Interval | The parameter *Interval* is optional and sets the interval (in milliseconds) at which values will be logged. |

*Table 2: Parameters for the start-metrixlogger command.*

One problem of expressing what to log is, that the measurement providing the wanted metrics for a specific meta element may not exist yet. Therefore, *MetrixLogger* uses so-called measurement descriptors which represent a specific metric as string. Their syntax looks like this:

```
Scope ["[" (Name|Id)[";parent="string] "]"] "." FigureName ["."
("Current"|"Total")]
```

Basically, this representation allows to refer to a meta element via name or id. The indexer brackets are omitted for runtime scoped measurements. Also, if *Current* or *Total* is not given, *Current* is assumed. Therefore, a user can describe the number of open slots of the extension *MetrixLogger* in several ways (assuming that the id of the extension is 7) :

```
ExtensionInfo[MetrixLogger].Openslotinfos.Current
ExtensionInfo[MetrixLogger].Openslotinfos
ExtensionInfo[7].Openslotinfos.Current
ExtensionInfo[7].Openslotinfos
```

To distinguish between several meta elements with the same name, the element's id can be used. The actual ids of all *MetrixLogger* instances can be found out with the following command:

```
get-extension MetrixLogger
```

Additionally, the option *parent* can be set after the name of a meta element. This option allows to distinguish same-named elements by specifying their parent element. It is supported for *SlotInfo*, *PlugInfo*, *SlotTypeInfo* and *PlugTypeInfo*. The parent element is an *ExtensionInfo* for the first two and an *ExtensionTypeInfo* for the last two types, which can be referred to by name. Table 3 explains several measurement descriptors.

| Measurement descriptor | Description |
|---|---|
| Runtime.Extensioninfos<br>Runtime.Extensioninfos.Current | Current number of extensions in the extension store. |
| Runtime.Extensioninfos.Total | Total number of extensions in the extension store since Metrix started monitoring. |
| ExtensionInfo[Workbench].Pluggedinbound | Current number of plugs plugged into the extension with name Workbench. |
| SlotInfo[Plux.View;Parent=Workbench].Pluggedinbound | Current number of plugs plugged into the *Plux.View* slot of the extension with name Workbench. |

*Table 3: Examples for measurement descriptors which allow to refer to not yet existing measurements.*

The following example logs several characteristics of the Plux.NET composition framework to a CSV file whenever a metric changes:

```
start-metrixlogger -File out.csv -OnEvents -LogWriter CSVLogWriter
-Measurements Runtime.Extensioninfos,Runtime.Bytesinuse,
Runtime.Maxextensioninfos, Runtime.Maxqueueitems, Runtime.Queueitems.Total,
PluginInfo[Metrix.dll].Shared
```

The log file looks like this:

```
Time;Runtime.Extensioninfos;Runtime.Bytesinuse;Runtime.Maxextensioninfos;Runtime.Maxqu
eueitems;Runtime.Queueitems.Total;PluginInfo[Metrix.dll].Shared
04.09.2009 14:42:34;24;12943400;n/a;121;42;n/a
04.09.2009 14:42:34;24;12951592;n/a;121;43;n/a
04.09.2009 14:42:34;24;12967976;n/a;121;44;n/a
04.09.2009 14:42:34;24;12984360;n/a;121;45;n/a
[…]
04.09.2009 14:42:34;25;12884156;n/a;217;266;n/a
04.09.2009 14:42:34;25;13056188;n/a;217;267;7
04.09.2009 14:42:34;25;13088956;25;217;268;7
[…]
04.09.2009 14:45:08;20;5565132;25;217;270;7
04.09.2009 14:45:08;20;5565132;25;217;270;7
04.09.2009 14:45:08;20;5589708;25;217;271;7
04.09.2009 14:45:08;19;5606092;25;217;271;7
04.09.2009 14:45:08;19;5614284;25;217;271;7
04.09.2009 14:45:08;18;5643444;25;217;271;7
04.09.2009 14:45:08;18;5643444;25;217;271;7
```

*Listing 4: An excerpt from the log file.*

For this example, the Plux.NET environment was started by launching *Runtime.exe*. After that, above console command started the logger. In the first two blocks of output, the number of extensions steadily increases up to 25 and the memory usage rises. This happens because the logger creates several figure extensions which measure the logged values. Because not all figure extensions are instantiated at the beginning, two of the columns contain the value *n/a*. Later on, the two missing measurements also return values. The metric *PluginInfo[Metrix.dll].Shared* indicates the number of shared extensions which come from the Metrix plug-in. A value of 7 is reasonable because the logger (one extension) logged mea-

surements from six different figures. The logger itself belongs to the plug-in *Metrix.dll* whereas the *CSVLogWriter* is a separate plug-in. The third block shows the shutdown of the Plux.NET application: When the logger itself was released by the composition framework, 18 extensions still were alive.

## 2.4 Metrix Documentation

Metrix itself is plug-in based and therefore extensible. It includes the extension *Metrix-DocBuilder*, which creates HTML documentation pages for all available metrics. This extension extracts the information from the figure extensions which provide the metrics.

To create a current documentation, it suffices to create the extension *MetrixDocBuilder*. The user can achieve this in the Plux.NET console through the following command:

```
create-extension MetrixDocBuilder
```

After that, a set of HTML files will be created in a folder named *metrixdoc* in the Plux.NET base directory. The file *index.html* gives an overview of all available metrics and includes references to detail pages for each metric. It is organized as a matrix: The columns represent the different scopes (like *Runtime*, *ExtensionInfo*, *SlotInfo*, …) whereas the rows contain the *FigureNames*. Therefore, it is possible to see at which granularity levels a metric is available.

The following table shows important runtime scoped metrics:

| FigureName | Description |
|---|---|
| avgslotinfos | The average number of slots per extension. |
| bytesinuse | The memory used by the application (in bytes). |
| closedslotinfos | The number of closed slots. A closed slot means that certain types of features (provided by another extension) can not be added at this time. |
| extensioninfos | The number of instantiated extensions. This metric describes the number of active components in the application. |
| filesize | The filesize of plug-ins and contracts (in bytes). It does not change until assemblies are installed into or removed from the type store. |
| maxextensioninfos | The maximum number of extensions instantiated so far. |
| maxqueueitems | The maximum number of tasks enqueued in the Plux.NET task queue so far. |
| openslotinfos | The number of open slots. Open slots can be used to enrich an extension with further functionality provided by other extensions. |
| pluggedplugs | The number of plug to slot connections. This metric describes to which extent the single extensions make use of each other. |
| queueitems | The number of tasks enqueued in the Plux.NET task queue. Tasks for composing applications through registering and plugging extensions |

| | |
|---|---|
| | are enqueued in the task queue and carried out one by one by the Plux.NET composition framework. |
| shared | The number of extensions that can be used by multiple other extensions at a time. |
| unique | The number of extensions which are supposed to be used by a single hosting extension. |
| uptime | The uptime of Plux.NET. |

*Table 4: Several runtime scoped metrics.*

Metrix can observe the following types of Plux.NET meta elements, which are all derived from the class *RepositoryElement*:

| Type (Scope) | Description | Important metrics |
|---|---|---|
| ExtensionTypeInfo | An extension type contains the meta data for specific extensions. | *extensioninfos*: The number of instantiated extensions.<br>*plugtypeinfos*: The number of plug types this extension type provides.<br>*registeredoutbound*: The number of plug types registered at slots of other extensions.<br>*shared*: The number of shared extension instances.<br>*unique*: The number of unique extension instances. |
| ExtensionInfo | An extension is an instance of an extension type. Multiple extensions of the same extension type can exist. | *closedslotinfos*: The number of closed slots belonging to this extension.<br>*creationtime*: The time when this extension was created.<br>*openslotinfos*: The number of open slots belonging to this extension. More open slots suggest a higher extensibility because this extension can use more contributing extensions.<br>*pluggedinbound*: The number of plugs plugged into this extension's slots. Indicates, to what extent the extension uses other extensions.<br>*pluggedoutbound*: The number of plugged plugs. Indicates, to what extent other extensions use this extension.<br>*pluginfos*: The number of provided plugs. Indicates, to what extent this extension can contribute to other extensions.<br>*registeredinbound*: The number of plug types registered at this extension's slots.<br>*slotinfos*: The number of slots. Indicates, to what extent this extension can be extended by other extensions. |
| PluginInfo | A plug-in is an | *extensiontypeinfos*: The number of extension types |

| | | |
|---|---|---|
| | assembly containing at least one extension type. | included in this assembly.<br>*filesize*: The size of the assembly file in bytes.<br>*plugincontribution*: The number of plug-ins using this plug-in.<br>*pluginusage*: The number of plug-ins this plug-in uses. |
| SlotDefinition | A slot definition is an interface definition that an extension must implement to support a specific plug. | *paramdefinitions*: The number of parameter definitions which this slot definition provides.<br>*slotinfos*: The number of slots using this slot definition. |
| ContractInfo | A contract is an assembly containing at least one slot definition. | *filesize*: The size of the assembly file in bytes.<br>*paramdefinitions*: The number of parameter definitions included in this assembly.<br>*slotdefinitions*: The number of slot definitions included in this assembly. |
| PlugTypeInfo | A plug type belongs to an extension type and represents a slot definition which this extension type implements. It holds the meta data for plugs. | *paramvalues*: The number of parameter values used to describe this plug type.<br>*pluggedoutbound*: The number of this plug type's plugs which are plugged into slots. Describes, how often this plug type's functionality is used.<br>*pluginfos*: The number of plugs belonging to this plug type.<br>*registeredoutbound*: The number of slots this plug type is registered at. |
| PlugInfo | A plug belongs to an extension. It can be plugged into a compatible slot of another extension. | *pluggedoutbound*: The number of slots the plug is plugged into. Indicates, how often this plug's functionality is used. |
| SlotTypeInfo | A slot type belongs to an extension type and holds meta data for slots. | *slotinfos*: The number of slots belonging to this slot type. |
| SlotInfo | A slot belongs to an extension. Slots can be opened or closed. A plug of another extension can be plugged into an open slot. | *opentime*: The last time this slot was opened.<br>*pluggedinbound*: The number of plugged plugs.<br>*registeredinbound*: The number of plug types registered to this slot. |

*Table 5: Plux.NET meta elements which Metrix can observe.*

# 3 Metrix Design Goals

## 3.1 Structure of Metrix

Metrix contains a set of metrics which describe various aspects of the Plux.NET composition framework. As stated earlier, metrics are provided for different meta elements of the Plux.NET composition framework or for the application as a whole. This is described through the *Scope*. When a metric has the scope *ExtensionInfo* for example, then this metric is calculated for each single extension. Another categorization is the *FigureName* which describes the semantics of a metric. When several metrics share the same *FigureName*, but have different scopes, then those metrics express basically the same, but at different granularity levels. The combination of the properties *FigureName* and *Scope* is referred to as *Figure*. Two distinct figures with the same *FigureName* provide the number of loaded extensions for each plug-in and for the whole application. In contrast, the amount of memory used by the application is also runtime scoped, but expresses something completely different. Therefore this figure also has another *FigureName*:

| Figure Description | FigureName | Scope | Number of values |
|---|---|---|---|
| Number of loaded extensions for each plug-in | extensioninfos | PluginInfo | 1 current value and 1 total counter for each plug-in |
| Number of loaded extensions in the whole application | extensioninfos | Runtime | 1 current value and 1 total counter |
| Memory used by the application | bytesinuse | Runtime | 1 current value |

*Table 6: Each figure represents an unique combination of FigureName and Scope.*

As shown above, a figure may also calculate multiple values: At plug-in level for example, a figure returns one value for each plug-in. Moreover, there can be also a second value for each meta element, which represents a total counter.

In Plux.NET, applications are constructed by composing extensions. Extensions can be connected with others through the concept of plugs and slots. An extension is an instance of an extension type. An assembly containing one or more extension types is called a plug-in.

In the plug-in *Metrix.dll*, each figure is represented as an extension: This enables the user to just load those figures actually needed, keeping the memory footprint lean. Once a figure is unplugged from all other extensions and thus not used currently, there is no need to keep it alive any longer: Therefore it will be released and later recreated, if necessary. To keep it simple, Metrix figures use the shared instance mode of Plux.NET: This means that the figure's extension type will be instantiated at most once. Other extensions just have to ask Plux.NET for

a shared instance of the requested figure: If the figure is already in use, Plux.NET returns its reference. If not, it creates the shared instance.

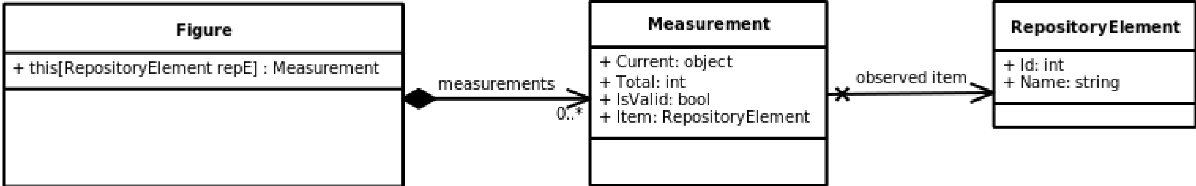


*Figure 1: Metrix figure extension with measurement collection.*

However, as figures keep track of all items belonging to their scope, there is a finer unit of organization: Each figure provides a collection of so-called measurements, which shrinks and grows as observed items get released or are created. One such measurement belongs to a specific Plux.NET meta element and represents the actual measured metric for it. A measurement provides a *Current* value, and a *Total* value if feasible. Figures like the number of loaded extensions, which count something, expose a *Total* value which increases whenever the *Current* value increases. Therefore, the user knows how many extensions were loaded since Metrix started monitoring and how many are loaded currently. In contrast, figures like the creation time of an extension instance do not expose a *Total* value since they do not count anything. The *Current* value may be of any arbitrary type, the *Total* value is a number (of type int32) always. Section 4.1.1 (page 14) provides more information about figure extensions and their measurements.

Metrix is packaged in six assemblies: One contract assembly contains the interface definitions for Metrix, whereas five plug-in assemblies contribute the functionality:

| Assembly | Description |
| --- | --- |
| Metrix.Contracts.dll | This contract contains the slot definitions for Metrix extensions. Applications reference this assembly when they use Metrix. |
| Metrix.dll | This plug-in contains the Metrix core extensions with more than 40 figures, a logger and a documentation tool. |
| LogWriter.dll | This plug-in contains three *LogWriter* extensions which can be used in conjunction with the Metrix logger. |
| Controls.dll | This plug-in contains four Windows Forms controls for visualizing metrics. |
| MetrixCmdlets.dll | This plug-in integrates Metrix into the new Microsoft Powershell based Plux.NET console. If Microsoft Powershell is not installed, Metrix can still be used without command line access by removing this plug-in. |
| MetrixExamples.dll | This plug-in contains several example applications using Metrix. |

*Table 7:The plug-ins Metrix consists of.*

## 3.2 Behavior of Metrix

This subsection introduces the life-cycle of Metrix figures and measurements.

When Plux.NET creates a new meta element the figure wants to observe, the figure creates a new measurement. After that, the figure notifies its users that a new measurement is available. Now, the measurement can be used to query the desired metric values.

However, at some point in time, Plux.NET will free the meta element. This happens for example, when an extension gets released. Metrix takes care of this by notifying clients when their measurement of interest gets invalid. Alternatively, Clients can ask a measurement whether it is still valid. However, for performance reasons, measurements will not check themselves for validity when accessing a metric value. Also, measurements never become valid again once they turn invalid.

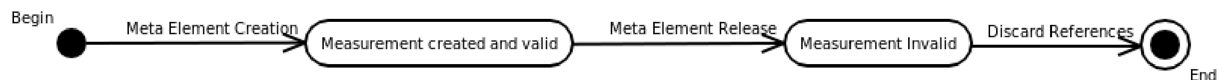


*Figure 2: Measurement life-cycle.*

Metrix is designed to update metrics only when needed. This works by observing Plux.NET runtime events. However, sometimes metrics are recalculated although they have not changed: This is the case, when no fitting events propagating these metrics' changes are available. Of course, the *Total* counters must be increased too whenever the *Current* counters rise.

Summing up, Metrix tries to avoid work which may not be needed. Also, there is no need to poll for new measurements and values. Instead, Metrix provides a push model: Metrix will notify its clients when new measurements are available or metric values might have changed.

# 4 Developing with Metrix

This section describes how a developer can extend Metrix with custom figures.

## 4.1 Interfaces

This subsection discusses three interfaces for extending Metrix: Firstly, the slot definition *Plux.Figure* allows to create custom figure extensions. Secondly, *IMeasurement* encapsulates the metric values for one specific item. Thirdly, *ILogWriter* allows to extend the logger.

### 4.1.1 IFigure and IMeasurement

A Metrix figure is an own extension type that can be instantiated as shared extension. Due to the nature of Plux.NET, the capabilities of Metrix are defined by a contract. This facilitates that figures from different authors can be used in a uniform manner. The following slot definition lists the properties and events every Metrix figure exposes:

```
[SlotDefinition("Plux.Figure")]
[Param("Scope", typeof(string))]
[Param("FigureName", typeof(string))]
[Param("Type", typeof(string))]
[Param("SupportsTotal", typeof(bool))]
[Param("Description", typeof (string))]
public interface IFigure {
  event EventHandler<FigureEventArgs> MeasurementAdded;
  event EventHandler<FigureEventArgs> MeasurementRemoved;
  IEnumerable<IMeasurement> Measurements { get; }
  IMeasurement this[RepositoryElement repositoryElement] { get; }
}
```

*Listing 5: Slot definition for Plux.Figure slots (see Metrix.Contracts\IFigure.cs).*

The attributes for the interface *IFigure* describe structured meta data which must be reported by a specific figure. A user of a figure can then interpret these meta data. The most important fields are *Scope* and *FigureName*: The first one describes to which type of meta element this figure applies, whereas the second one defines the actual name of the figure. Several figures supporting different scopes can have the same *FigureName* as long as they express the same semantics: This pattern is used frequently when metrics are cumulated for different scopes. For example, the number of extensions of a specific extension type and the number of extensions for the whole application have a different *Scope*, but the same *FigureName*. However, the scope in conjunction with the figure name is always unique among all figures.

The *Type* field defines the data type of the actual metric. *SupportsTotal* states whether the figure maintains a totals counter (of type int32). Finally, the *Description* field is used to make documentation easy and uniform within the code: This free-text field is accessed by various components to give further information about the figure.

The actual interface that a figure implements is narrow: Firstly, it allows to retrieve the measurement for a specific meta element or all measurements at once. Secondly, a user can observe the measurement collection for changes: When the measurement of interest does not exist yet, the client can subscribe to the *MeasurementAdded* event. It will notify the client whenever a new measurement is created.

Measurements encapsulate the actual metric values. Each measurement implements the interface *IMeasurement*:

```
public interface IMeasurement : INotifyPropertyChanged {
  RepositoryElement Item { get; }
  object Current { get; }
  int Total { get; }
  bool IsValid { get; }
}
```

*Listing 6: Interface definition for measurements (see Metrix.Contracts\IFigure.cs).*

Like *IFigure*, *IMeasurement* also exposes several properties. However, measurements are plain objects and not represented by an extension. Therefore, some static parameters which affect the measurements too (for example *Type* and *SupportsTotal*) are attached to the figure extension type. Because measurements implement *INotifyPropertyChanged*, a measurement notifies its clients when the metrics change or the measurement becomes invalid.

The *Item* property never changes since a measurement belongs to one specific meta element. This property can be null because no meta element exists for the runtime scope. This also applies to runtime scoped figures: To query the measurement, supply *null* as index.

Due to the life-cycle described earlier, *IsValid* can only change from true to false. As values must not be accessed once the measurement is invalid, clients should discard references to invalid measurements to facilitate garbage collection. Likewise, also the *Total* property must not be accessed when the figure indicates that it does not support totals.

### 4.1.2 ILogWriter

The Metrix logger can be extended by *LogWriter* extensions to support different data sinks (see page 5). To create a new *LogWriter* extension, the extension must implement the interface *ILogWriter*. The interface definition looks like this:

```
[SlotDefinition("LogWriter")]
public interface ILogWriter
{
     void Setup(Dictionary<string, string> configuration);
     void Open();
     void SetCaption(String[] caption);
     void Log(Object[] data);
     void Close();
}
```

*Listing 7: The slot definition for LogWriter slots (see Metrix.Contracts\IFigure.cs).*

Basically, the logger is table-oriented. It has a fixed set of measurement values which are all logged at specific points in time. Therefore, the number of columns in the table never changes, but the number of rows grows when new values are logged.

For familiarity, the methods of a *LogWriter* are similar to file operations: Firstly, the *Setup* method configures the log writer: For example, a file-based log writer could take a filename and a text encoding as parameter. For logging to a database, the connection string might be set this way. Secondly, the *Open* method prepares the data sink for writing data. A file based log writer would open the file whereas a database log writer would establish the connection to the database. Thirdly, the method *SetCaption* informs the log writer about the names of the metrics. Then, the logger calls the method *Log* whenever values should be written to the data sink. Finally, the logger calls the method *Close* to finish pending writing operations. After that, the log writer will be unplugged from the logger. Therefore, a *LogWriter* extension should have set *AutoRelease* to *true* in order to get released after logging.

## 4.2 Using Metrix in Applications

This section uses examples to explain how other extensions can use Metrix figures.

To use Metrix figures, the application's extension needs to have a *Plux.Figure* slot. Normally, the Plux.NET composition framework would plug all extensions having a *Plux.Figure* plug into that slot automatically. To prevent that and let the application choose which figures should be plugged, the property *AutoPlug* must be set to *false* for this slot. Because the application wants to select from multiple figures, the property *Multiple* must be set to *true*. By setting the properties *OnRegistered* and *OnPlugged*, the application can register event handlers for these events of the *Plux.Figure* slot.

```
[Extension]
[Plug("Startup")]
[Slot("Plux.Figure", AutoPlug = false, Multiple = true, OnRegistered =
"Figure_Registered", OnPlugged = "Figure_Plugged")]
internal class SampleApp : IStartup
{
    public void Run() { }
```

*Listing 8: Declaration of the SampleApp extension (see MetrixExamples\SampleApp.cs).*

To automatically instantiate the application extension at startup, it has a *Startup* slot and implements *IStartup* with the *Run* method. However, in the following sample application, nothing is actually done in this method.

The sample application's goal is to print the number of loaded extensions for the whole application and for its own extension type. Therefore, the application needs to access two figures with the same *FigureName* (*extensioninfos*), but a different *Scope* (*Runtime* and *ExtensionTypeInfo*).

The application connects these figure extensions in two steps: Firstly, the application chooses the figures' extension types. This happens at the registration phase. According to above slot configuration, the method *Figure_Registered* will be called for each *Plux.Figure* plug type. By inspecting the parameters *FigureName* and *Scope*, the application extension can decide whether to use an registered figure extension type or not. When the application wants to use a registered figure, it must obtain a reference to an instance of that extension type. This is done by calling the *GetSharedExtension* method of the wanted figure extension type. This method returns the shared instance of the extension. If the shared instance does not exist and the parameter *createOnDemand* is set to true, the method creates the shared extension. Then, the figure extension is plugged into the application's extension.

```
 public void Figure_Registered(object sender, RegisterEventArgs args)
 {
     // extract parameter values
     string figureName = (string)args.GetParamValue("FigureName");
     string scope = (string)args.GetParamValue("Scope");

     if (figureName == "extensioninfos" && (scope == "Runtime" || scope ==
"ExtensionTypeInfo"))
     {
       ExtensionTypeInfo figureExtensionType = args.PlugTypeInfo.ExtensionTypeInfo;
       // create the figure extension on demand, set this extension as creator
       ExtensionInfo figureExtension = figureExtensionType.GetSharedExtension(true,
myExtension);
       // plug figure extension into this extension
       figureExtension.PlugPlugs(args.SlotInfo.ExtensionInfo);
     }
 }
```

*Listing 9: This method is called whenever a figure is discovered. Based on the figure's parameters, the application decides whether the figure will be used.*

In the second step (see Listing 10), the wanted figure is already plugged into the application's extension. According to the *Plux.Figure* slot configuration, the method *Figure_Plugged* will be called each time a figure is plugged. To distinguish figures, the application can access the figure extension's parameters. This time however, it suffices to inspect the *Scope*. Because *AutoPlug* was set to false, only wanted figure extensions will be plugged. Since both of them have the *FigureName extensioninfos*, it is not inspected again. The application uses the property *Extension* of the figure extension to obtain the actual figure object which implements the *IFigure* interface. Now, the wanted measurement can be accessed via the indexer brackets. Because runtime scoped figures have just one measurement belonging to no specific meta element, the application supplies *null* as index.

```
public void Figure_Plugged(object sender, PlugEventArgs args)
{
    IFigure figure = (IFigure)args.Extension;

    // extract parameter values
    string scope = (string)args.GetParamValue("Scope");

    if (scope == "Runtime")
      Console.WriteLine("Number of extensions: " + figure[null].Current);
    else if (scope == "ExtensionTypeInfo")
      Console.WriteLine("Number of SampleApp extensions: " +
figure[args.SlotInfo.ExtensionInfo.ExtensionTypeInfo].Current);
}
```

*Listing 10: This method is called for each figure the application decided to use.*

The sample application uses the default way of composing Plux.NET extensions. For Metrix however, this yields in tedious code since the application needs to distinguish the figures by parameter values at two points. Therefore, Metrix ships with the *MetrixHelper* class, which generalizes those registration and plugging tasks and allows to access Metrix with more structured code. Instead of above steps, the extension has *IFigure* properties which are annotated with the *SetFigure* attribute. When a required figure extension is available, the corresponding property is set by *MetrixHelper*. Therefore it suffices to describe the wanted figure once in the *SetFigure* attribute with no further distinction. Of course, the registration and plugging events must be forwarded to *MetrixHelper*. There, another object different from the extension can be specified: This is useful when developing UI applications, because a form implementing the IFigure properties can be specified. In this way, the extension itself must neither pass on figures to the form, nor must the form export controls to which the extension can assign measurement values.

Furthermore, *MetrixHelper* provides convenient access to measurements. Often, measurements of interest are not available immediately. Therefore, the application needs to observe the figure's measurement collection until the wanted measurement is available. With *MetrixHelper*, the application does not need to care about the fact whether a measurement is available or not: The *MeasurementAvailable* and *MeasurementAvailableByName* methods call a method implementing the *MeasurementActionDelegate* as soon as the measurement is available. The delegate's signature looks like the following:

```
public delegate void MeasurementActionDelegate(IMeasurement measurement);
```

The following code is an equivalent implementation of above sample application using *MetrixHelper*:

```
[Extension]
[Plug("Startup")]
[Slot("Plux.Figure", AutoPlug = false, Multiple = true, OnRegistered = "Figure_Registered",
OnPlugged = "Figure_Plugged")]
class SimplyfiedSampleApp : IStartup
{
    private MetrixHelper helper = new MetrixHelper();

    public void Run() {}
```

```
    private IFigure runtimeExtensions;
    [SetFigure("Runtime","ExtensionInfos")]
    public IFigure RuntimeExtensions
    {
        set
        {
            runtimeExtensions = value;
            if (runtimeExtensions == null)
                return;
            helper.MeasurementAvailable(runtimeExtensions,null,m => Console.WriteLine("Number
of extensions: " + m.Current));
        }
    }

    private IFigure appExtensions;
    [SetFigure("ExtensionTypeInfo", "ExtensionInfos")]
    public IFigure AppExtensions
    {
        set
        {
            appExtensions = value;
            if (appExtensions == null)
                return;
            helper.MeasurementAvailableByName(appExtensions,"SimplyfiedSampleApp",m =>
Console.WriteLine("Number of SimplyfiedSampleApp extensions:" + m.Current));
        }
    }

    public void Figure_Registered(object sender, RegisterEventArgs args)
    {
        MetrixHelper.Register(Runtime.GetExtensionInfo(this), this, args);
    }

    public void Figure_Plugged(object sender, PlugEventArgs args)
    {
        MetrixHelper.Plug(Runtime.GetExtensionInfo(this), this, args);
    }
}
```

*Listing 11: Equivalent implementation of the former SampleApp extension using the class MetrixHelper (see MetrixExamples/SampleApp.cs).*

Note, that the mechanisms behind are still the same as in the previous example application.

## 4.3 Extending Metrix

Metrix already includes a collection of built-in figures as described in section 2.4 (Metrix Documentation). However, sometimes a user may want to define custom metrics. To do so, the user writes an extension which has a *Plux.Figure* plug and therefore implements *IFigure*.

For the discovery of figures, Metrix uses the Plux.NET mechanisms. Therefore, only a few points must be considered to accomplish the expected behavior: Firstly, the extension should be declared as singleton to prevent the creation of more than one instance: This is done by setting the property *Singleton* to *true* in the *Extension* attribute. Secondly, the extension name should obey the naming scheme "Metrix.<*Scope*>.<*FigureName*>" to meet users' expectations. The *FigureName* parameter value is always lowercase, whereas the first character of the figure name in the extension name must be uppercase. Then, a *Description* (free text) is needed, which is used by the Metrix documentation generator for example. The parameter *Type* represents the type of the *Current* value of the measurement. Finally, *SupportsTotal* indicates whether the figure's measurements have a total value.

As an example, a figure extension recording the time that the Plux.NET task queue was emptied last will be shown. The *Scope* is set to *Runtime*. Therefore it has only one permanent measurement and does not use the *MeasurementAdded* and *MeasurementRemoved* event definitions. The runtime scoped figure object returns its measurement when *null* is passed as index. Alternatively, it is also accessible via the *Measurements* property which returns a collection of all measurements (containing one measurement in this case). The type is set to *DateTime* and no total value is supported (indicated by setting parameter *SupportsTotal* to *false*). The figure updates its measurement by observing the *QueueEmptied* event of the Plux.NET task queue. The measurement itself is defined in the inner class *M* which implements *IMeasurement*. When the measurement's *Current* value is updated, the measurement informs its clients via the *PropertyChanged* event. The *Item* property is *null* because the measurement does not belong to a specific meta element:

```
[Extension("Metrix.Runtime.Taskqueueemptiedtime", Singleton = true, OnCreated =
"SampleFigure_Created", OnReleased = "SampleFigure_Released")]
[Plug("Plux.Figure")]
[ParamValue("Description", "Timestamp, when the task queue was emptied last.")]
[ParamValue("Scope", "Runtime")]
[ParamValue("Type", "DateTime")]
[ParamValue("FigureName", "taskqueueemptiedtime")]
[ParamValue("SupportsTotal", false)]
class SampleFigure : IFigure
{
    private M measurement;
    class M : IMeasurement
    {
        private DateTime lastAction = DateTime.Now;
        public DateTime LastAction
        {
            set
            {
                lastAction = value;
                if (PropertyChanged != null)
                  PropertyChanged(this,new PropertyChangedEventArgs("Current"));
            }
        }

        public object Current { get { return lastAction; } }

        public bool IsValid   { get { return true; } }

        public RepositoryElement Item { get { return null; } }

        public int Total { get { return 0; } }

        public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
    }

    public void SampleFigure_Created(object sender, ExtensionEventArgs args)
    {
        measurement = new M();
        Runtime.TaskQueue.QueueEmptied += TaskQueue_QueueEmptied;
    }

    public void SampleFigure_Released(object sender, ExtensionEventArgs args)
    {
        Runtime.TaskQueue.QueueEmptied -= TaskQueue_QueueEmptied;
    }

    void TaskQueue_QueueEmptied(object sender, TaskEventArgs args)
    {
        measurement.LastAction = DateTime.Now;
```

```
    }

    // unused
    public event EventHandler<FigureEventArgs> MeasurementAdded;
    public event EventHandler<FigureEventArgs> MeasurementRemoved;

    public IEnumerable<IMeasurement> Measurements
    {
        get { return new IMeasurement[] { measurement }; }
    }

    public IMeasurement this[RepositoryElement repositoryElement]
    {
        get
        {
            if (repositoryElement == null)
                return measurement;
            return null;
        }
    }
}
```

*Listing 12: A sample figure extension implementation without using helper classes. An inner measurement class is used to store the actual values (see MetrixExamples\SampleFigure.cs).*

To ease the development of new figures, Metrix offers two base classes a new figure may inherit from. Firstly, *GroupingFigureBase* allows to group the measurements of another figure. Secondly, *FigureBase* is more flexible than *GroupingFigureBase* but also requires more code.

### 4.3.1 Grouping Figures

The class *GroupingFigureBase* groups measurements of another figure by applying a function on their values. Therefore, a grouping figure always depends on another figure called *ChildFigure*. A grouping figure has less or equal measurements than its *ChildFigure*. A typical task for a grouping figure would be to sum up the number of extensions per extension type to obtain the number of extensions in the whole application. Therefore, the new grouping figure would be *Runtime* scoped and use the *ExtensionTypeInfo* scoped figure *Metrix.ExtensionTypeInfo.Extensioninfos* as *ChildFigure*. Note, that the real figure implementing this metric is actually not a *GroupingFigure* for performance reasons: For using a *GroupingFigure*, its *ChildFigure* must be instantiated too. Therefore, for often used and easy computable *Runtime* scoped figures like the number of extensions, it pays off to implement them as normal figure extension (see page 25).

A grouping figure can be configured by overriding the get-accessor of at most three properties:

| Property | Description |
|---|---|
| ChildFigureName | The name of the figure extension the new figure is based on. This property is mandatory. |
| GroupingRule | The method which is used to determine whether a measurement belongs to a certain group of measurements which make up a new measurement. The default value is *GroupingRules.CheckJoin*. |
| GroupingFunction | The method which is used to actually calculate the *Current* value of the new measurement. The default value is *GroupingFunctions.Summation*. |

*Table 8: Properties, which allow to configure the GroupingFigure base class.*

A method which is used as a grouping rule must conform to the *GroupingRule* delegate:

```
public delegate bool GroupingRule(RepositoryElement parent,
RepositoryElement child);
```

The grouping rule must state for two meta elements (of type *RepositoryElement*), whether they should form a relation. If so, it returns *true*, otherwise *false*. All child measurements which belong to the same parent will be used to calculate the new value of the parent measurement. The following code excerpt from the default grouping rules set (*GroupingRules.cs* in *Metrix.dll*) shows the rule which groups all extension measurements by their extension type:

```
private static bool CheckJoin(ExtensionTypeInfo extType, ExtensionInfo ext)
{
        return (ext.ExtensionTypeInfo == extType);
}

public static bool CheckJoin(RepositoryElement parent, RepositoryElement child)
{
        if (parent == null) // Runtime
        return true; // accumulate all child measurements
        if (parent.GetType() == child.GetType())
            return (parent == child); // generic one-to-one grouping rule
        [...]
        if (parent is ExtensionTypeInfo)
        {
                ExtensionTypeInfo p = (ExtensionTypeInfo) parent;
                [...]
                if (child is ExtensionInfo)
                return CheckJoin(p, (ExtensionInfo) child);
        }
        [...]
}
```

*Listing 13: This join rule defines a parent-child relation between an extension type and its extension instances (see Metrix\JoinRules.cs).*

Metrix already ships with two grouping rules in the *GroupingRules* class:

| Grouping Rule | Description |
|---|---|
| CheckJoin | This rule applies to all types of meta elements and represents the relations between them. For example, it defines which extension type belongs to which plug-in. It defines the following child-parent relations:<br>PlugInfo, SlotInfo → ExtensionInfo<br>PlugInfo → PlugTypeInfo<br>SlotInfo → SlotTypeInfo<br>ExtensionInfo, PlugTypeInfo, SlotTypeInfo, PlugInfo, SlotInfo → ExtensionTypeInfo<br>ExtensionTypeInfo, ExtensionInfo, PlugTypeInfo, SlotTypeInfo, PlugInfo, SlotInfo → PluginInfo<br>PlugTypeInfo, SlotTypeInfo, PlugInfo, SlotInfo → SlotDefinition<br>SlotDefinition → ContractInfo |
| CheckJoinNoMetrix | This rule is based on *CheckJoin*. However, it tries to exclude meta elements from Metrix. It is used for some runtime scoped figures to prevent them from measuring the effects of Metrix figures. Those figures' names end with the suffix *_nm*. |

*Table 9: The two join rules included in Metrix are defined in Metrix/JoinRules.cs.*

A method which is used as *GroupingFunction* must comply with the *GroupingFunction* delegate:

```
public delegate object GroupingFunction(IMeasurement[] childMeasurements,
ref object store, string type);
```

The method must return a value for the new measurement which is based on a set of *childMeasurements*. The *store* object (initialized with *null*) allows the *GroupingFunction* to store an arbitrary object between two calculations for each measurement. This is used by the *HistoricMaximum* function to save the highest ever value for example. Finally, it also gets the type of the *childMeasurements*' values.

Metrix defines several grouping functions in the *GroupingFunctions* class:

| Grouping Function | Description |
|---|---|
| CountNonZero | This function counts the number of current child measurements where the *Current* value is not 0. |
| Summation | This function sums up the *Current* values of all current child measurements. |
| Minimum | This function calculates the current minimum of the *Current* values of all child measurements. |
| HistoricMinimum | Like Minimum, but considers all measurements so far, not only the current minimum. |

| | |
|---|---|
| Maximum | This function calculates the current maximum of the *Current* values of all child measurements. |
| HistoricMaximum | Like Maximum, but considers all measurements so far, not only the current maximum. |
| Average | This function calculates the average value of all *Current* values of the current child measurements. |

*Table 10: The grouping functions included in Metrix (see Metrix\GroupingFunctions.cs).*

Note, that *GroupingFigure* does not support grouping functions for *Total* values: It only has a built-in summation functionality for *Total* values. Therefore, when using another *GroupingFunction* than *Summation*, the parameter *SupportsTotal* must be set to *false*.

As an example, the *CountNonZero* grouping function listed above is implemented in the following way:

```
public static object CountNonZero(IMeasurement[] childMeasurements, ref object store, string
type)
{
    int count = 0;
    object cur;
    long val=0;

    foreach (IMeasurement m in childMeasurements)
    {
          cur = m.Current;
          if (cur is int)
             val = (long)(int)cur;
          else if (cur is long)
             val = (long)cur;
          if (val != 0)
             count++;
    }
    return count;
}
```

*Listing 14: This function counts the number of measurements with a Current value other than zero (see Metrix\GroupingFunctions.cs).*

Note, that the above function does not use the *store* parameter. Evaluating the *type* parameter instead of analyzing the individual measurements would also be possible.

When deriving from *GroupingFigureBase*, an *OnReleased* event handler called “OnReleased” must be registered for the extension so that the base class can properly shut down.

Finally, as an example for a *GroupingFigure* declaration, the code of the figure *Metrix.Runtime.Maxextensioninfos* is given. It calculates the maximum number of loaded extensions so far by observing the figure *Metrix.Runtime.Extensioninfos*. Therefore it actually does not group multiple measurements into one but transforms a given one into a new one by utilizing the *HistoricMaximum* grouping function:

```
[Extension("Metrix.Runtime.Maxextensioninfos", Singleton = true,
  OnReleased = FigureBase.ON_RELEASED)]
[Plug("Plux.Figure")]
[Slot("Plux.Figure", AutoPlug = false, LazyLoad = true, Multiple = true, OnRegistered =
"Figure_Registered", OnPlugged = "Figure_Plugged")]
```

```
[ParamValue("Description", "max(#ExtensionInfos)")]
[ParamValue("Scope", "Runtime")]
[ParamValue("Type", "int")]
[ParamValue("FigureName", "maxextensioninfos")]
[ParamValue("SupportsTotal", false)]
public class RtMaxextensioninfos : GroupingFigureBase
{
    public override string ChildFigureName
    {
        get { return "Metrix.Runtime.Extensioninfos"; }
    }
    public override GroupingFunction GroupingFunction
    {
        get { return GroupingFunctions.HistoricMaximum; }
    }
}
```

*Listing 15: This figure is implemented in Metrix\Figures\GroupingFigures.cs.*

In contrast, the figure counting all unique extensions in the whole application actually aggregates all *ExtensionTypeInfo* scoped measurements into one value. Because the property *GroupingFunction* is not overridden, it is assumed as *Summation*:

```
[Extension("Metrix.Runtime.Unique", Singleton = true,
  OnReleased = FigureBase.ON_RELEASED)]
[Plug("Plux.Figure")]
[Slot("Plux.Figure", AutoPlug = false, LazyLoad = true, Multiple = true, OnRegistered =
"Figure_Registered", OnPlugged = "Figure_Plugged")]
[ParamValue("Description", "#unique ExtensionInfos")]
[ParamValue("Scope", "Runtime")]
[ParamValue("Type", "int")]
[ParamValue("FigureName", "unique")]
[ParamValue("SupportsTotal", true)]
public class RtUnique : GroupingFigureBase
{
    public override string ChildFigureName
    {
        get { return "Metrix.ExtensionTypeInfo.Unique"; }
    }
}
```

*Listing 16: This figure is implemented in Metrix\Figures\GroupingFigures.cs.*

### 4.3.2 FigureBase

For more flexibility, developers can derive their figure extension from *FigureBase*. This base class also requires the registration of an *OnReleased* event handler called “OnReleased”.

The class *FigureBase* collects measurements and notifies clients about new or removed measurements. To add or remove a measurement, the methods *AddMeasurement* and *RemoveMeasurement* should be used. For creating runtime scoped figures, the figure extension should derive from *RuntimeFigureBase*, which in turn inherits from *FigureBase*. Then, the field *runtimeM* should be used to store the runtime wide measurement.

Additionally, a (nested) measurement class must be declared which should inherit from *MeasurementBase*. When the figure does not support *Total* counters, the measurement class should derive from *CurrentOnlyMeasurementBase*. The get-accessors of the remaining properties *Current* and *Item* must also be implemented. The *MeasurementBase* class contains multiple methods: Firstly, *FireCurrentChanged* notifies clients about a new *Current* value of the

measurement. Secondly, *IncreaseTotal* increases the *Total* counter of the measurement by a given value or by one, if no parameter is set. Note that the parameter of *IncreaseTotal* must be greater or equal than one, otherwise an exception will be thrown. Finally, the method *Invalidate* marks the measurement as invalid.

To conveniently create or remove measurements, a nested class can be derived from *RepositoryElementRegistrar*. Then, the developer can overwrite the *Register<Scope>(...)* and *Unregister<Scope>(...)* methods. This way, the developer must not distinguish between monitored meta elements which already exist at the time of figure creation and those which appear later on. The *Register<Scope>(...)* method will be called for every meta element. The class actually calling these methods is *EventMonitor*. Therefore, the constructor of the registrar class creates an *EventMonitor* instance and assigns it to the *monitor* field. After that, the constructor sets several *Enable** properties to configure which hook methods of the registrar class should be called. Finally, the constructor calls the *Start* method of the monitor instance. Since the registrar class is derived from *RepositoryElementRegistrar*, the base class will shutdown the monitor instance properly.

The following example measures the number of open slots per extension and illustrates the usage of those base classes:

```
[Extension("Metrix.ExtensionInfo.Openslotinfos", Singleton = true,
    OnReleased = FigureBase.ON_RELEASED)]
[Plug("Plux.Figure")]
[ParamValue("Description", "#Slotinfos | isopen=true")]
[ParamValue("Scope", "ExtensionInfo")]
[ParamValue("Type", "int")]
[ParamValue("FigureName", "openslotinfos")]
[ParamValue("SupportsTotal", true)]
internal class EInfOpenslotinfos : FigureBase
{
    public EInfOpenslotinfos()
    {
        registrar = new Registrar(this);
    }

    private class M : MeasurementBase
    {
        private ExtensionInfo ext;

        public M(ExtensionInfo ext)
        {
            this.ext = ext;
        }

        public override RepositoryElement Item
        {
            get { return ext; }
        }

        public override object Current
        {
            get
            {
                int sum = 0;

                if (IsValid)
                {
                    foreach (SlotInfo slot in ext.SlotInfos)
```

```
                    sum += slot.IsOpen ? 1 : 0;
                }
                return sum;
            }
        }
    }

    private class Registrar : RepositoryElementRegistrar
    {
        private EInfOpenslotinfos figure;

        public Registrar(EInfOpenslotinfos figure)
        {
            this.figure = figure;
            monitor = new EventMonitor(this);
            monitor.EnableExtensionInfo = true;
            monitor.EnableSlotInfo = true;
            monitor.Start();
        }

        public override void RegisterExtensionInfo(ExtensionInfo ext)
        {
            figure.AddMeasurement(ext, new M(ext));
            MeasurementBase m = figure.MeasurementOf(ext);
            m.Total = (int) m.Current;
        }

        public override void UnregisterExtensionInfo(ExtensionInfo ext)
        {
            figure.RemoveMeasurement(ext);
        }

        public override void RegisterSlotInfo(SlotInfo slot)
        {
            slot.Opened += sInf_Opened;
            slot.Closed += sInf_Closed;
        }

        public override void UnregisterSlotInfo(SlotInfo slot)
        {
            slot.Opened -= sInf_Opened;
            slot.Closed -= sInf_Closed;
        }

        private void sInf_Opened(object sender, SlotEventArgs args)
        {
            MeasurementBase m = figure.MeasurementOf(args.SlotInfo.ExtensionInfo);
            m.OnCurrentChanged();
            m.IncreaseTotal();
        }

        private void sInf_Closed(object sender, SlotEventArgs args)
        {
            MeasurementBase m = figure.MeasurementOf(args.SlotInfo.ExtensionInfo);
            m.OnCurrentChanged();
        }
    }
}
```

*Listing 17: The implementation of Metrix.ExtensionInfo.Openslotinfos which uses the FigureBase, MeasurementBase and RepositoryElementRegistrar base classes (see Metrix\Figures\EInfOpenslotinfos.cs).*

The figure class derives from *FigureBase*. It uses an inner measurement class that derives from *MeasurementBase* since the figure supports *Total* counters. Also, an inner registrar class based on *RepositoryElementRegistrar* takes care of attaching a measurement to every extension. Therefore, it sets the *EnableExtensionInfo* property of its monitor instance to *true*, so that the *RegisterExtensionInfo* and *UnregisterExtensionInfo* methods will be called. Note that

the term *register* in this context has nothing in common with *register* in terms of Plux.NET. Additionally, *EnableSlotInfo* is also set to true because the figure extension wants to bind event handlers to each slot.

Therefore, this pattern keeps the code clean of non-measurement related event handlers and ensures a clean removal of them: The *Unregister* methods will also be called for each slot or extension when the figure extension is released. For each measurement, the *Total* counter is initially set to the *Current* value. From then on, the *Total* counter is increased each time a slot is opened via event observation. When a slot is opened or closed, the figure extension does not recalculate the *Current* value of the according measurement. Instead, it just notifies its clients about the change of this property: The figure calculates the new value, when the *Current* property of the measurement is actually queried.

## 4.4 UI Controls

Metrix ships with four Windows Forms controls for visualizing metrics. The seven segment digit display and the bar control are single value controls. The other two, the plotter and the pie chart control, can visualize multiple values. The following subsections give a brief overview on how to use these *UserControl* based controls in conjunction with Metrix.

### 4.4.1 Single Value Controls

The digit display and the bar control can take advantage of Windows Forms data binding to display metrics. The following code snippet from the UI panel of the example application *MetrixSidebar* demonstrates this:

```
        // Variable declarations by Windows Forms designer
         private Bar runtimeExtensioninfosBar;
         private DigitDisplay taskQueueDigit;

         private IFigure runtimeExtensioninfos = null;
        [SetFigure("Runtime","Extensioninfos")]
        public IFigure RuntimeExtensioninfos
        {
            set
            {
                runtimeExtensioninfos = value;
                if (runtimeExtensioninfos != null)
                {
                    helper.MeasurementAvailable(runtimeExtensioninfos,null, m =>
runtimeExtensioninfosBar.DataBindings.Add("Value", m, "Current");
                    );
                }
            }
        }

        private IFigure runtimeMaxExt;
        [SetFigure("Runtime", "Maxextensioninfos")]
        public IFigure RuntimeMaxExt
        {
            set
            {
                runtimeMaxExt = value;
                if (runtimeMaxExt != null)
                {
                    helper.MeasurementAvailable(runtimeMaxExt,null,m =>
```

```
runtimeExtensioninfosBar.DataBindings.Add("MaxRange", m, "Current"));
                }
            }
        }

        private IFigure queueItems = null;
        [SetFigure("Runtime", "Queueitems")]
        public IFigure QueueItems
        {
            set
            {
                queueItems = value;
                if (queueItems != null)
                {
                    helper.MeasurementAvailable(queueItems, null, m =>
taskQueueDigit.DataBindings.Add("Value", m, "Total"));
                }
            }
        }
```

*Listing 18: Measurement values are bound to a bar control and to a seven segment digit display (see MetrixExamples\SidebarUC.cs).*

The corresponding figures are assigned by the *MetrixHelper* class which is explained in section 4.2 (Using Metrix in Applications). When the measurement for the application-wide number of extensions is available, the measurement's *Current* value is bound to the *Value* property of the bar control. Likewise, the measurement describing the maximum number of application-wide extensions is bound to the *MaxRange* property. Finally, the total number of enqueued tasks is bound to the *Value* property of the digit display. Whenever the value of a measurement changes, the control updates itself through data binding because measurements implement the *INotifyPropertyChanged* mechanism. The only exception is the figure extension *Metrix.Runtime.Bytesinuse* since it does not observe events and therefore lacks change notification. To display this metric, a Windows Forms timer which assigns the current value to a control at a specific interval can be used.

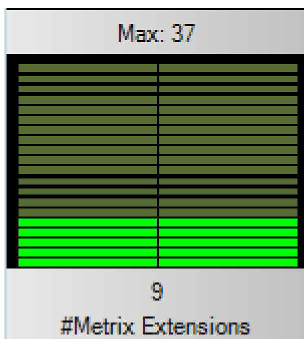


*Figure 3: The bar control.*

*Figure 4: The seven segment digit display.*

The bar control exposes the following properties:

| Property | Type | Description |
|---|---|---|
| AutoScale | bool | If set, the *MaxRange* value will increase to the maximum value of the *Value* property. |

| DarkColor | Color | The dark color used to paint the bar's stripes. |
|---|---|---|
| Description | string | The description displayed near the bar. |
| HorizontalAlignment | bool | If set, the bar is drawn horizontally instead of vertically. |
| LegendColor | Color | The color of the area containing the *Description* and the numeric *Value*. |
| LightColor | Color | The light color used to paint the bar's stripes. |
| MaxRange | long | The maximum value the bar visualizes. Must be greater or equal the *Value* property. The bar displays the fraction *Value*/*MaxRange*. |
| Value | long | The displayed value. Must be less or equal the *MaxRange* property unless *AutoScale* is not set. |

*Table 11:Properties of the bar control (see Controls\Bar.cs).*

The seven segment digit display exposes the following properties:

| Property | Type | Description |
|---|---|---|
| Description | string | The description displayed above the seven segment digit display. |
| DisplayColor | Color | The color the digits are painted in. |
| EnableKPostfix | bool | If set, the symbol *K* is appended to the value. This is used for displaying memory usage. |
| LegendColor | Color | The color of the area containing the *Description*. |
| NumberOfDigits | byte | The number of digits the maximum displayed *Value* will have. Used to ensure consistent layout when values with less digits are displayed first. However, if necessary, the number of digits will automatically increase. |
| Value | long | The value which will be displayed in digits. |

*Table 12:Properties of the seven segment digit display (see Controls\DigitDisplay.cs).*

### 4.4.2 Multi-Value Controls

The plotter and the pie chart control can display several values at once. They use an array of *IDataItem* objects as data source. The definition of the interface *IDataItem* looks like this:

```
public interface IDataItem : INotifyPropertyChanged
{
    string Description { get; }
    bool IsValid { get; }
    object Value { get; }
}
```

*Listing 19: Objects which should be displayed in the multiple value controls must implement this interface (see Metrix.Contracts\IDataItem.cs).*

An *IDataItem* provides a value and a description, which are both displayed by the multi-value controls. Furthermore it has an *IsValid* property: When this property returns false, the control knows that the value is not valid from now on and does not query it again. This is useful for example when a measurement turns invalid and should not be accessed. An *IDataItem* must report the change of its properties via *INotifyPropertyChanged*. Metrix already ships with *MetrixDataItem*, an *IDataItem* implementation which wraps *IMeasurement*. Its constructor takes an *IMeasurement*, a description and a boolean indicating whether it should return the measurement's *Current* or *Total* value as *Value*:

```
public MetrixDataItem(IMeasurement m, string description, bool total);
```

The following snippet (adapted from the *ControlsView* example application) demonstrates the assignment of data to the pie chart control:

```
List<MetrixDataItem> list = new List<MetrixDataItem>();
int count = 0;
foreach (IMeasurement m in figure.Measurements) {
   if(++count > 5) break;
   MetrixDataItem item = new MetrixDataItem(m, m.Item.Name, false);
   list.Add(item);
}
PieChart1.DataItems = list.ToArray();
```

*Listing 20: Configuration of a PieChart control (see MetrixExamples\ControlsView.cs).*

In the above example, the first five measurements' *Current* values are assigned to a pie chart control named *PieChart1*. In order to display the *Current* value, the application sets the third parameter in the constructor of *MetrixDataItem* to *false*. The pie chart also displays the name of the measured items because it was set as description. Assigning measurements to the plotter control works in the same way.

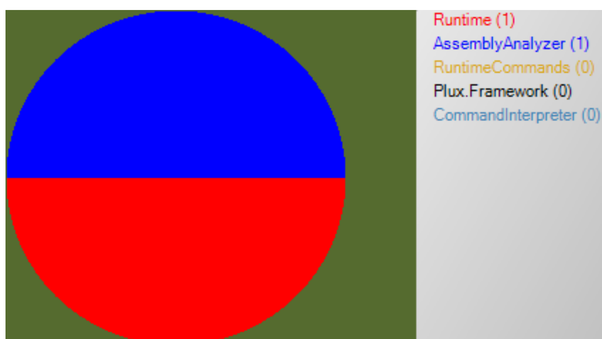


*Figure 5: The pie chart control.*

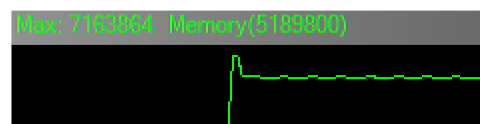


*Figure 6: The plotter control.*

The pie chart control exposes the following properties:

| Property | Type | Description |
|---|---|---|
| DataItems | IDataItem[] | The data source of the displayed values. |
| LegendColor | Color | The color of the area containing the *Description* and the numeric *Value* of the *DataItems*. |
| PieColors | Color[] | Array of colors used to visualize the *DataItems*. |

*Table 13:Properties of the pie chart control (see Controls\PieChart.cs).*

The plotter control exposes the following controls:

| Property | Type | Description |
|---|---|---|
| AutoScale | bool | If set, the *MaxValue* will be automatically increased to the highest *Value* of the *DataItems*. |
| DataItems | IDataItem[] | The data source of the displayed values. |
| GraphColors | Color[] | The colors used to visualize the values. |
| Interval | int | The interval in milliseconds at which the control will visualize new values. |
| LegendColor | Color | The color of the area containing the *Description* and the numeric *Value* of the *DataItems*. |
| MaxValue | float | The maximum value which can be displayed. Adjusts itself to the highest peak encountered so far when *AutoScale* is set. Other values scale in relation to the *MaxValue*. |
| QueryOnUpdate | bool | Normally, the plotter updates its values when the *DataItems* report a change. If set, the plotter queries all *DataItems* for new values at the set *Interval*. Useful, when measurements which do not report changes (for example memory usage) should be displayed. |

*Table 14: Properties of the plotter control (see Controls\Plotter.cs).*

# 5 Further Work and Discussion

As shown in this paper, Metrix allows to work already with more than 40 metrics. However, there is desirable functionality which is not implemented yet. This section explains the barriers encountered when developing these features and makes suggestions on how to overcome them.

## 5.1 Derived metrics must be compiled

Deriving new metrics based on existing ones without compiling code would be a useful feature: For example, declaring the metric *Metrix.Runtime.Averageslotinfos* as a division of *Metrix.Runtime.SlotInfos* by *Metrix.Runtime.Extensioninfos* on the command line would allow users to calculate relevant metrics quickly. One approach would be to define a special figure extension type whose extension instances can be configured by the user. However, without breaking the concepts of Metrix, this is difficult to achieve: Metrix expects that each single figure is represented by a single extension type with at most one extension instance. Therefore, extensions using Metrix can rely on finding the desired figure by just inspecting the extension types' parameters. When a configurable extension type as described above would be developed, this would not work anymore: Firstly, there would be as many extension instances as declared metrics, thus violating the rule of having one shared instance per figure extension type. Secondly, extensions using Metrix would also have to inspect the properties of those instances to find the metric of interest. To overcome this problem in a clean way, Plux.NET needs to provide a way of declaring extension types without code. Because this is not possible yet, Metrix introduced the concept of *GroupingFigures*: It allows to define some simple operations like summation or average calculation in compact code.

## 5.2 Limited analysis of memory usage

The second problem encountered when developing Metrix is the measurement of memory usage. Originally, Metrix wanted to measure the memory usage for each extension. However, the .NET framework just allows obtaining the memory usage per application domain. Since the size of base types and code blocks can be obtained, this information could be used to calculate an approximation of the memory usage by examining all referenced objects recursively. Plux.NET meta elements could act as a border for this summation of object sizes. How to deal with data that is shared by multiple extensions (like Metrix measurements used in other extensions or strings), is an open question. However, this method would allow a qualitative analysis of two extensions' memory usage.

# 6 References

All web references were retrieved on October 5, 2009.

*Learning Windows PowerShell Names*. Microsoft TechNet. http://technet.microsoft.com/en-us/library/dd315315.aspx

*Piping and the Pipeline in Windows PowerShell*. Microsoft TechNet. http://www.microsoft.com/technet/scriptcenter/topics/winpsh/manual/pipe.mspx

*Windows PowerShell*. Microsoft Windows PowerShell Website. http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx