# Plugins for cDashboard

Alexander White

March 10, 2015

## Contents



# 1 Getting Started

cDashboard is a wonderful program. However, it did not have plug-in support until recently. This means that it could not in any fashion open up external assemblies in order to acheive a specific goal. This is, fortunately, no longer the case, at this point it is possible to write plug-ins that are loaded using the MEF framework.

This tutorial is written with the assumption that you are able to read and write C# reasonably well[1], and that you are using a version of visual studio, along with the availability of the MEF on your version of .Net.

If you are unable to get any of these things, I would recommend not adventuring further into this until you do.

# 2 The Basics of cDashboard Plugining

## 2.1 Setting up a project

1. Open up Visual Studio
2. Create a new DLL project under C# or Visual Basic[2].
3. Add a reference to the System.ComponentModel.Composition Assembly[3]
4. Add a reference to the cDashboard project or executable.

[1] if not then here's a link
[2] Or C++ if you know Microsoft's super special syntax
[3] This is amongst the most important things.

5. ???

6. Profit?

### 2.1.1 Example of What ??? means.

You might have noticed that there was a spooky mysterious series of question marks in the list above. This is because you are writing the code, and far be it from me to specify what you do. So here is an example of what you might wish to do:

Create a class called Calculator

Your file should look something like this:

```
using System;
namespace CalcProject{
        class Calculator{
                public Calculator(){
                        //do things
                }
        }
}
```

Now in order to be visible as an assembly to MEF we need to make some changes.

```
using System;
using System.ComponentModel.Composition;
namespace CalcProject{
        [Export(typeof(IPlugin))]
        //This tells the Composition framework about some
            useful metadata, as defined in IPluginData.
        [ExportMetadata("name", "Calculator")]
        class Calculator{
                public Calculator(){
                        //do things
                }
        }
}
```

The Export annotation there puts the type annotation into the finished assembly. The ExportMetadata tells cDashboard the name of the plugin, which in this case seems to be "Calculator". However, an observant programmer will notice that it doesn't work. That's because it needs to implement the cDashboard.IPlugin interface. This interface is the basis of all contact between the program and the plugin, so it's very important.

```
using System;
using System.ComponentModel.Composition;
using cDashboard;
namespace CalcProject{
```

```
        [Export(typeof(IPlugin))]
        //This tells the Composition framework about some
            useful metadata, as defined in IPluginData.
        [ExportMetadata("name", "Calculator")]
        class Calculator:IPlugin{
                public bool NeedsSaving{
                        set; get;
                }
                public Calculator(){
                        //do things
                }
                public System.Windows.Forms.Form GetForm(){
                        throw new NotImplementedException();
                }
                public void SavePlugin(string
                    settingslocation){}
                public void LoadPlugin(string
                    settingslocation,cDashboard dash){}
                public bool DisposeOnClose{get;}
                Type getFormType(){
                        return typeof(CalculatorForm);
                }
        }
}
```

Hmm... It doesn't do anything still. You should also, if you have not already, add a reference to the System.Windows.Forms assembly in the project manager. Now let's look at what we have here.

1. **NeedsSaving** Returns whether or not the Plugin needs to be saved[4], and then wiped after each time it is saved(This needs to be made accessible from the form)[5].
2. **GetForm()** Retrieves an instance of the form that the plugin provides.
3. **SavePlugin(string)** Saves information[6] about the plugin so that it remains persistent across sessions, this is called regularly by the main process, saving it about once a second, so it is important to not save too much inefficiently.
4. **LoadPlugin(string,cDashboard)** Called when cDashboard is loaded, loads the forms' settings from the configuration file, there is a method callable on the dash object called AddForm which adds forms, it is also important to remember that should you have a more intensive plugin, you might want to save this to an instance variable so you can manipulate the overall gui.

[4] and then called a second later.

[5] This should be accomlished without issue by having a static bool value that is written to by the plugin's form and then returned as NeedsSaving when necessary

[6] This needs to include things like window positions.

5. **DisposeOnClose** informs cDashboard that the form should be removed after it has been closed.

6. **getFormType()** returns the type of the form that the plugin provides.

There are still a few things missing from it, such as a list of the forms provided to the main program, as you have no method for saving the plugin's settings unless you keep track of them here. So you should add something like

```
            private List<WeakReference> formsinstances;
```

It is important to make it store WeakReferences as otherwise the cDashboard program won't be able to do garbage collection on the closed forms(or anything else directly pertaining to the forms)

I won't provide a CalculatorForm class, but you should at the very minimum amend the Save and Load Plugin methods to store the location of the forms, which should look something like

```
 public void SavePlugin(string settingslocation){
        List<string> lines=new List<string>();
        //store the information in a list prior to writing to
            a file(this works better)
        foreach(var i in forminstances){//Iterate over the
            form instances
                if(i.IsAlive()){
                        //make sure it is in fact alive,
                            otherwise enjoy your null pointer
                            exceptions
                        Form a=(Form)i.Target;
                        //Cast the object to a Form
                        lines.Add(a.Location.X.ToString()+"
                            "+a.Location.Y.ToString());
                        //and store the location as a string
                }
        }
        System.IO.File.WriteAllLines(settingslocation+"calculator.cDash",lines.ToArray())
        //Write it all to a file(all at once).

        NeedsSaving=false;
        //Mark that the program no longer needs to save this.
}

public void LoadPlugin(string settingslocation,cDashboard
    dash){
        //Once again, store the information in a single list
        List<System.Drawing.Point> locations=new
            List<System.Drawing.Point>();
        //Read the file all at once
        var
            g=System.IO.File.ReadAllLines(settingslocation+"calculator.cDash");
```

```
        //iterate over each line.
        foreach(var line in g){
                //Split at the space
                var setting=line.Split(' ');
                locations.add(
                        new System.Drawing.Point(
                                int.Parse(setting[0]),
                                int.Parse(setting[1]))
                        );
        }
        foreach(var loc in locations){
                var c=GetForm();
                //Initialize the form with a location
                c.Location=loc;

                dash.AddForm(c);
        }
}
```

So there you have it, a reasonable API for reasonable tasks. Hopefully this is clear enough.