Advanced Customization Tutorial

Table of Contents

Getting Started

Before we get started on how to build your own custom components and drives, there is some terminology you must know first:

- *Component*. The internal (i.e., functional) objects that define how the bottom and top levels of the subsystems within an agent operate. All components (and component templates) extend from the `ClarionComponent` class (located in the *Clarion.Framework.Templates* namespace).

- *Implicit Component*. The bottom level components. All bottom level components extend from the `ImplicitComponent` class (located in the *Clarion.Framework.Templates* namespace).
- *Rule*. The top level components. All top level components extend from the `Rule` class (located in the *Clarion.Framework.Templates* namespace).
- *Drive*. A special component for the bottom level of the MS (more details to follow). All components intended for the bottom level of the MS extend from the `Drive` class (located in the *Clarion.Framework.Templates* namespace).
- *Module*. The meta-cognitive subsystem, itself, is not actually a subsystem (per say) and so it does not technically have a top and bottom level. Instead, the meta-cognitive subsystem is a container for meta-cognitive "modules", which themselves contain a top and bottom level. The meta-cognitive modules act like "mini" ACSs, containing much of the same capabilities and requirements as the actual ACS.
    - All meta-cognitive modules extend from the `MetaCognitiveModule` class (located in the *Clarion.Framework.Templates* namespace).[1]

Next, you need to be aware of the specific requirements for each of the subsystems. The top and bottom levels of the subsystems are not entirely generic. Each subsystem has its own requirements when it comes to the type of components it will accept at the bottom and top levels:

ACS Structure

The ACS is defined by the `ActionCenteredSubsystem` class and has the following structure:

1. The bottom level of the ACS expects components that extend from the `ImplicitComponent` class, therefore any component that is intended for the bottom level of the ACS **MUST** extend this class.
2. The top level of the ACS will only accept 3 types of rules: refineable action rules, IRL rules, and fixed rules. Therefore, any component that is intended for the top level of the ACS **MUST** extend either the `RefineableActionRule`, `IRLRule`, or `FixedRule` classes (located in the *Clarion.Framework* namespace)

[1] While it is technically possible to implement a custom meta-cognitive module, this is a **VERY** advanced (i.e., developer level) customization and requires a deep knowledge of the interworking of the system. A "*Developer Tutorial*" may be made available upon request by contacting clarion.support@gmail.com.

NACS Structure

The NACS is defined by the `NonActionCenteredSubsystem` class and has the following structure:

1. Like the bottom level of the ACS, the bottom level of the NACS also expects components that extend from the `ImplicitComponent` class, therefore any component that is intended for the bottom level of the NACS **MUST** also extend this class.
2. The top level of the NACS expects components that extend from the `AssociativeRule` class (located in the *Clarion.Framework.Templates* namespace), therefore any component that is intended for the top level of the NACS **MUST** also extend this class.

MS Structure

The MS is defined by the `MotivationalSubsystem` class and has the following structure:

1. As was stated earlier, the bottom level of the MS expects components that derive from the `Drive` class. The `Drive` class is a somewhat special component since it is really just a wrapper for an implicit component. In fact, the Drive class itself expects a component that extends from the `ImplicitComponent` class.
2. Unlike the other subsystems, the top level of the MS is special in that it does not actually contain components. Instead, the top level of the MS contains goals (See the "*Setting Up & Using the Goal Structure*" tutorial in the "*Basic Tutorials*" section of the "*Tutorials*" folder for details concerning how to setup and use goals).

MCS Structure

The MCS is defined by the `MetaCognitiveSubsystem` class, however, all of the functionality for the MCS is defined by the `MetaCognitiveModule` classes[2] that are contained within it. These modules have the following structure:

1. The bottom level of a meta-cognitive module is essentially the same as the bottom level of the ACS in that it expects components that extend from the `ImplicitComponent` class. Therefore, any component that is intended for the bottom level of a meta-cognitive module **MUST** extend this class.
2. The top level of a meta-cognitive module is similar to the top level of the ACS, except that it only accepts one type of rule, refineable action rules. Therefore, and component that is intended for the top level of a meta-cognitive module **MUST** extend the `RefineableActionRule` class.

[2] All currently implemented meta-cognitive modules can be found in the *Clarion.Framework.Extensions* namespace

Interfaces and Templates

In addition to what we have laid-out so far, you should also be aware of the various interfaces that are available to you (and located in the *Clarion.Framework.Templates* namespace). These interfaces inform the system about the capabilities of your component. For example:

- *Is your component trainable?* If so, it needs to implement the `ITrainable` interface.
- *Can your component be trained using reinforcement learning?* If so, it needs to implement the `IReinforcementTrainable` interface.
- *Does your component use Q-learning?* If so, it needs to implement the `IUsesQLearning` interface.
- *Will your component track positive and negative match statistics?* If so, it needs to implement the `ITracksMatchStatistics` interface.
- *Can your component be deleted by the system (say, via density considerations)?* If so, it needs to implement the `IDeletable` interface.
- *Does your component require the input for the following state to perform learning?* If so, it needs to implement the `IHandlesNewInput` interface.
- *Is your component able to extract rules?* If so, it needs to implement the `IExtractsRules` interface.
- *Can your component be refined (using the RER algorithm)?* If so, it needs to implement the `IRefineable` interface.
- *Etc.*

As is the convention for C#, all interfaces in the Clarion Library begin with "I" and are followed by a brief description of the capabilities they provide. The specifics about how to implement an interface can be found in the documentation for that interface.

Furthermore, several "template classes" have been provided that implement parts of (or even most of) certain interfaces. For example:

- The `Rule` class fully implements the `ITracksMatchStatistics` interface.
- The `TrainableImplicitComponent` class implements parts of the `ITrainable` interface.
- The `ReinforcementTrainableImplicitComponent` and `ReinforcementTrainableBPNetwork` classes implement parts of the `ITracksMatchStatistics`, `IReinforcementTrainable`, and `IExtractsRules` interfaces.
- Etc.

Hopefully, at this point, you are becoming excited about all of the possibilities for customization that are available. One of the advantages of working with the Clarion

Library, is that you do not need to feel restrained by what has been provided to you "in-the-box." For example, if you don't want to use a 3-layer neural network that implements backpropagation, then you can write your own 3-layer neural network by extending the `NeuralNetwork` class (in the *Clarion.Framework.Templates* class). Then you could use whatever learning algorithm you would like by implementing the `ITrainable` interface. Suppose you don't need a hidden layer or you want to implement a 2-layer recurrent neural network. You could do this by extending the `ImplicitComponent` class and implementing whatever capabilities (i.e., interfaces) you wish for your network to have. The possibilities for customization are almost limitless!

So now that we have laid-out the groundwork, we are ready to start building some custom components (and drives).

How to Implement a Custom Component

The Clarion Library has been designed to allow for a maximal amount of customization while still maintaining an interaction that is straightforward and congruent with the conception of the Clarion theory. This being said, we encourage users to explore and expand-upon the foundation that has been developed and made available within the Clarion Library.

As a means of aiding customization, we have created several "template classes" (found within the *Clarion.Framework.Templates* namespace) that can you can build-upon to create your own customized internal (i.e., function) components.

Requirements for Implementing a Custom Component

First, there are three things that **ALL** components **MUST** have:

1. A "Factory" class that can be used by the `AgentInitializer` class to generate the component
2. A "Parameters" class that stores any parameters that may be "fine tuned" to improve the performance of your component
3. A "Commit" method (which "wires-in" the component after it has been setup and makes it "immutable" to an extent) and a "Retract" method (which removes the component from operation and makes it editable again).

So let's start by looking at how to build a factory for your component.

Implementing a "Factory"

To begin, your factory class **MUST** implement one of the following generic factory interfaces:

- `IimplicitComponentFactory<T>` if you intend for your component to be an `ImplicitComponent`
- `IActionRuleFactory<T>` if you intend for your component to be an `ActionRule`

- `IAssociativeRuleFactory<T>` if you intend for your component to an `AssociativeRule`
- `IDriveFactory<T>` if you intend for your component to be a `Drive`

For all of the above interfaces, the generic `<T>` indicator specifies the type of component that the factory will create (i.e., the class name of your component).

As a standard practice, it is usually a good idea to simply define your factory class as a "nested class" inside of your component and to create a single `static` instance of that factory that can be called statically from your component. This is how the factory class is implemented for the "built-in" components in the Clarion Library.

Let's suppose, for the sake of demonstration, that you wanted to create a custom implicit component. If you were to follow the standard convention, your code should look something like this:

```
public class SomeCustomComponent : ImplicitComponent
{
    public class SomeCustomComponentFactory :
        IImplicitComponentFactory<SomeCustomComponent>
    {
        public SomeCustomComponent Generate(params dynamic[] parameters)
        {

            //Elided code for parsing-out the parameters

            return new SomeCustomComponent();
        }
    }

    protected SomeCustomComponent()
        : base (new
            ImplicitComponentParameters(ImplicitComponent.GlobalParameters)) { }

    private static SomeCustomComponentFactory factory =
        new SomeCustomComponentFactory();

    public static SomeCustomComponentFactory Factory
    {
        get
        {
            return factory;
        }
    }
}
```

As a rule of thumb, the name of your factory class should simply be the name of your component with the word "Factory" appended to the end it.

By using the standard convention, if a user wanted to use your component, they would be able to easily initialize it without having to also know where to find the factory for your component (and without needing to instantiate an instance of that factory). For example, suppose someone wanted to initialize your component in the bottom level of the ACS, the following line would accomplish this:

```
SomeCustomComponent comp =
           AgentInitializer.InitializeImplicitDecisionNetwork
           (SomeAgent, SomeCustomComponent.Factory);
```

The agent initializer calls the `Generate` method from your factory class to generate an instance of your component and it places that component within the specified agent at its intended location (for instance, as in the example above, as an implicit decision network).

Generating a component can be as simple a process as the code above has just demonstrated, but suppose you need to require that some basic information be provided in order for your component to be successfully initialized. By leveraging the combination of the `params` and `dynamic` keywords, the `Generate` method can also take an arbitrary number of parameters as input. Within your `Generate` method, all you have to do is "parse" the parameters list, and "extract" the information needed by your component (or throw an exception if the required information was not specified). For example, lets assume that our `SomeCustomComponent` class contains a construct called "nodes" and that a user must specify how many of these "nodes" need to be setup as part of initializing the component. This requirement can be setup within the `Generate` method as follows:

```
public SomeCustomComponent Generate(params dynamic[] parameters)
{

    int numNodes = 0;

    foreach (dynamic p in parameters)
    {
        if (p is int)
            numNodes = (int)p;
    }

    if (numNodes <= 0)
        throw new ArgumentException("You must specify the number of nodes " +
            "(greater than 0) that are to be created in order to initialize " +
            "this component");

    return new SomeCustomComponent(numNodes);
}

...

//The custom component constructor
protected SomeCustomComponent(int numNodes)
    : base (new ImplicitComponentParameters(ImplicitComponent.GlobalParameters))
{
    //Elided initialization code using numNodes
}
```

Since the initialization parameters, passed into the `Generate` method, are `dynamic`, your `Generate` method will be responsible for "parsing out" the parameters that are needed for initializing your component. The method demonstrated in the example above uses the `is` operator to correctly assign the parameters based on their type. However, this method only works if no two parameters have the same type. If your component needs more than one parameters of a specific type, then you should consider using a different method for "parsing" the parameters (e.g., require that the parameters be specified in a certain order).

Your `Generate` method should always either successfully return a new instance of your component or throw an exception if any of your required parameters are missing. By throwing these sorts of exceptions in the `Generate` method, you can clearly convey to other users the mistakes they have made while trying to initialize your component. This will also cut down on unforeseen problems that might occur at runtime due to mistakes during initialization.

In addition to required parameters, you can also specify optional initialization parameters. For example, by default, all components automatically inherit the `IsEligible` method from the `ClarionComponent` base class. However, the base constructor for this class has the option for specifying an `EligibilityChecker` delegate (located in the *Clarion.Framework.Templates* namespace) during initialization. If that delegate is specified, the `IsEligible` method will use the delegate method to check the component's eligibility in lieu of using the default method. The following code would allow the `SomeCustomComponent` class to have that same option:

```
public SomeCustomComponent Generate(params dynamic[] parameters)
{

    int numNodes = 0;
    EligibilityChecker el = null;

    foreach (dynamic p in parameters)
    {
        if (p is int)
            numNodes = (int)p;
        if (p is EligibilityChecker)
            el = (EligibilityChecker)p;
    }

    if (numNodes <= 0)
        throw new ArgumentException("You must specify the number of nodes " +
            "(greater than 0) that are to be created in order to initialize " +
            "this component");

    return new SomeCustomComponent(numNodes, el);
}

...

//The custom component constructor
protected SomeCustomComponent(int numNodes, EligibilityChecker elChecker = null)
    : base (new ImplicitComponentParameters(ImplicitComponent.GlobalParameters),
    elChecker)
{
    //Elided initialization code
}
```

The only real difference between optional and required initialization parameters is whether an exception is thrown. Obviously, your component should operate correctly regardless of whether a user specifies an optional parameter during initialization, so there is no need to throw an exception for them.

Expanding on our earlier example, initializing your component (with parameters) would now look something like this:

```
SomeCustomComponent comp =
    AgentInitializer.InitializeImplicitDecisionNetwork
    (SomeAgent, SomeCustomComponent.Factory,
    SomeNumberOfNodes, (EligibilityChecker)SomeEligibilityMethod);
```

Remember, depending on which "template" (or other "built-in" component) you extend, you will probably want to "parse out" all of the parameters (both required and optional) for your base class as well. To determine which parameters your base class needs/wants, consult the documentation of the constructor for that class. In general, the parameters that have been specified with a default value (in the signature for the constructor) are optional, and the ones without a default value are required (with regard to the `Generate` method as well as in standard C# terms).

The final thing you will need to do in your `Generate` method is collect together all of the "factory parameters" that are being used to initialize your component and store them in `FactoryParameters` (located in the `ClarionComponent` base class). Doing this will allow the system to automatically generate new instances of your component based off of the configuration of a single other instance. The following code demonstrates how this would look in the `Generate` method of our custom component example:

```
public SomeCustomComponent Generate(params dynamic[] parameters)
{

    int numNodes = 0;
    EligibilityChecker el = null;

    foreach (dynamic p in parameters)
    {
        if (p is int)
            numNodes = (int)p;
        if (p is EligibilityChecker)
            el = (EligibilityChecker)p;
    }

    if (numNodes <= 0)
        throw new ArgumentException("You must specify the number of nodes " +
            "(greater than 0) that are to be created in order to initialize " +
            "this component");

    List<dynamic> fpars = new List<dynamic>();
    fpars.Add(numNodes);
    if (el != null)
        fpars.Add(el);
    result.FactoryParameters = fpars.ToArray();
    return new SomeCustomComponent(numNodes, el);
}
```

A good example of where the "factory parameters" are used by the system is in generating "rule variations" based on a given "refineable rule." So, for instance, let's suppose that our custom component extended `RefineableActionRule`. By setting the `FactoryParameters` in our `Generate` method, when an instance of our custom rule is added to the rule store, the system will automatically be able to perform the refinement process using that rule (by auto-generating variation instances of our custom component using the specified factory parameters).

At this point, we have covered everything you need to know with regard to setting up a "factory" for your custom component. Therefore, let's now turn our attention to considering how a custom component should handle the "non-initialization" parameters (i.e., those parameters that can be "fine tuned" during runtime in order to improve the overall functioning of a component).

Implementing a "Parameters" class

Recall that, back in the "*Intermediate ACS Setup*" tutorial, we discussed how parameters can be accessed and manipulated in the Clarion Library either globally (by statically calling the `GlobalParameters` property) or locally (by calling the `Parameters` property on a per-instance basis). This feature is accomplished through the implementation of a "parameters" class, which houses all of the "tunable" (non-initialization) parameters and provides properties for getting and setting them. By default, all of the "built-in" components (and subsystems, etc.) contain two instances of this sort class: a global (`static`) instance, and a local instance.

The global parameters act as the "default" settings for all local instances of a component (or subsystem, etc.). In other words, whenever a local instance is initialized, the parameters for that instance are set using the values from the global (`static`) instance.

It is your prerogative to decide whether you want to implement the global (`static`) instance for your custom component (although it recommended). However, your component **MUST** at least have a local instance of a parameters class. We decided to make this a requirement for two reasons:

1. It retains consistency with the rest of the system, which makes it easier for users to "fine tune" your component
2. It is necessary to do so if you want the manipulation of parameters via actions[3] to operate correctly[4]

[3] See the "*Advanced ACS Setup*" tutorial for more details on this feature
[4] If we didn't require this then you would have to write your own event handler method for parameter changes using action events (plus override the `ParameterChangeRequestedDelegate` property). This is NOT a simple task and would likely result in your component operating in a way

Of course, if your component does not add any new parameters to those that are inherited from the base class, then you will not need to implement a new parameters class. In this case, you can simply instantiate a new instance of the base class's parameters class during the initialization of your custom component. In the example we presented previously, this is accomplished by the following code:

```
public SomeCustomComponent()
        : base (new
            ImplicitComponentParameters(ImplicitComponent.GlobalParameters)) { }
```

In all likelihood, however, your component is probably going to have at least a few "tunable" parameters, so you will probably need to create a new parameters class for your custom component. So let's look at the simplest way to create a parameters class (i.e., "local" only).

Local (per instance) Parameters

As was the convention for the "factory" class, we recommend you implement your parameters class as a "nested class" within your custom component. This is the standard convention used throughout the Clarion Library. Using this convention, the `SomeCustomComponent` class (from our previous examples), would look something like this:

```
public class SomeCustomComponent : ImplicitComponent
{
    public class SomeCustomComponentParameters : ImplicitComponentParameters
    {
        ...
    }

    ... //Elided factory class, etc.
}
```

As was suggested for the factory class, the name of your parameters class should simply be the name of your component with the word "Parameters" appended at the end of it. In addition, your parameters class **MUST ALWAYS** extend from the parameters class of the base class of your component (for the example above, this would be the `ImplicitComponentParameters` class). This is necessary to ensure that your parameters class inherits all of the parameters that are needed for the base class of your component. In addition, as you will see in a moment, the local instance of your parameters class is stored generically at the bottom of the inheritance hierarchy (a.k.a., within the `ClarionComponent` class), so it **MUST**, at least, extend from the `ClarionComponentParameters` class.

Now we can begin filling our parameters class with parameters. In general, it is a good idea to declare your parameters as `private` fields within the class and then use properties to get and set the parameter publicly. We recommend this for two reasons:

other than was intended whenever action events are used to manipulate the parameters of your component.

1. All parameters **MUST** be accessible via a property for the manipulation of parameters through action events to operate correctly
2. If you are going to implement the global parameters instance, you will need to use properties
3. It is the recommended convention for accessing fields (as suggested by Microsoft) when programming in C#

Let's assume that our `SomeCustomComponent` class uses a "learning rate" parameter as part of its learning operation.[5] To implement this parameter, we would do the following:

```
public class SomeCustomComponentParameters : ImplicitComponentParameters
{
    private double lr = .1;

    public double LEARNING_RATE
    {
        get
        {
            return lr;
        }
        set
        {
            lr = value;
        }
    }
}
```

First, notice that we set a value at the top for the learning rate when it is initialized. This is the "default" value for the parameter (i.e., its value if it is never manipulated). Second, notice that the property is in all caps and fully names the parameter. This is done as a matter of style. It is a common practice in programming to specify "constants" using unambiguous names in all capital letters, but feel free to use whatever style you would like for naming your parameters. Note, however, that you will probably want to stick with the common practice here since your parameters class will also inherit parameters from its base class, and those parameters are named using this style.

A local instance of your parameters class must be initialized at the same time as your component. This gives you two options for where you can perform this initialization: in the `Generate` method, or in the constructor. The latter is the easier of the two methods, and would be accomplished as follows (for the simplest `SomeCustomComponent` example):

```
public SomeCustomComponent(): base (new SomeCustomComponentParameters()) { }
```

Although this options works perfectly well for correctly initializing the local instance of your parameters class, you will probably prefer to initialize it in the

[5] This also means that it would have to implement the `ITrainable` interface

`Generate` method. This way, you can add the ability for users to specify their own instance of your parameters class as an optional initialization parameter for your component. This capability would be especially useful to a user who has already tuned an instance of your component and now wants to use the parameter settings for that instance to initialize additional instances.

Note, however, that in the above case, it would be better to **COPY** the settings from the instance of the parameters class specified by the user instead of actually using that instance as the local parameters instance for the new component. Otherwise, there is a very good chance that the user could end up with 2 or more instances of your component that are sharing the same local parameters instance. This would mean that if the user were to change a parameter for one of those instances, it will change that parameter for **ALL** of the instances (which could create problems or unintended consequences for that user).

To address the above consideration, all of the parameters classes for the "template" classes (and other "built-in" components) in the Clarion Library contain 2 constructors; one of which takes an instance of a parameters class as input and sets the value of its parameters using that instance. The following code shows how you would setup these constructors for your own component's parameters class:

```
public class SomeCustomComponentParameters : ImplicitComponentParameters
{
    ...

    //The constructors for the custom component's parameters class
    public SomeCustomComponentParameters() : base() { }

    public SomeCustomComponentParameters(SomeCustomComponentParameters p)
        : base(p)
    {
        //Set parameters here based on the values of p
        LEARNING_RATE = p.LEARNING_RATE;
    }

    ... //Elided parameter properties
}
```

The second constructor in the above example will not only copy the values for all of your parameters, it will also copy the values for all of the parameters inherited from the parameters class of your component's base class. You can now use this constructor in your `Generate` method to copy the settings from the instance of the parameters class specified by the user. The following code would accomplishes this:

```
public SomeCustomComponent Generate(params dynamic[] parameters)
{
    SomeCustomComponentParameters pars = null;

    foreach (dynamic p in parameters)
    {
        if (p is SomeCustomComponentParameters)
            pars = (SomeCustomComponentParameters)p;
    }

    if (pars == null)
        pars = new SomeCustomComponentParameters();
    else
        pars = new SomeCustomComponentParameters(pars);

    List<dynamic> fpars = new List<dynamic>();
    fpars.Add(pars);
    result.FactoryParameters = fpars.ToArray();

    return new SomeCustomComponent(pars);
}

...

//The custom component constructor
public SomeCustomComponent(SomeCustomComponentParameters pars) : base (pars) { }
```

After your component has been initialized, its "local" parameters instance can be accessed via the `Parameters` property. However, by default, this property will return your component's parameters class in a "down casted" state. For example, calling the "Parameters" property for an instance of `SomeCustomComponent` will return a parameters class instance that is specified as being type `ImplicitComponentParameters`, even though it is really of the type `SomeCustomComponentParameters`.

As a result, a user would be forced to always manually cast the instance returned by the `Parameters` property. This is **VERY** inconvenient, so to get around the problem, your component should override the `Parameters` property and provide the correct casting for your component's parameters class. In the `SomeCustomComponent` class, this is accomplished by implementing the following:

```
public class SomeCustomComponentParameters : ImplicitComponentParameters
{
    ... //Elided parameters class, factory class, constructors, etc.

    public new SomeCustomComponentParameters Parameters
    {
        get
        {
            return (SomeCustomComponentParameters) base.Parameters;
        }
    }
}
```

Now users of your component will be able to access your parameters without constantly needing to cast the instance returned by the `Parameters` property. Your component should also access its parameters using the `Parameters` property whenever it is performing calculations that make use of them. The following code demonstrates how a user could modify the `LEARNING_RATE` parameter from our previous examples:

```
comp.Parameters.LEARNING_RATE = .5;
```

At this point, you have learned how to setup a parameters class in its simplest form (i.e., "local" only). However, we can also setup a parameters class such that it can act as either a local instance **OR** a global instance.

Global (`static`) Parameters

In order to implement global parameters, you need to be aware of some key factors:

1. Every "template" class (and "built-in" component) contains a global parameters instance
2. Global parameters are implemented using a static (singleton) instance of a parameters class and are accessed statically using a static property
3. The global parameters instance is specified as being global during the initialization (i.e., in the constructor) of that instance
4. The global parameters instance is **ONLY** used during the initialization of a component as a means for setting the local parameters of that component
5. Global parameter changes can be made at any point along the inheritance hierarchy
6. Changing a parameter globally at any point along the inheritance hierarchy will change the value of that parameter for all classes that are "downstream" from (i.e., the subclasses of) the class where the global parameter change was initiated
7. The parameters class of a component class should override the properties of all of the parameters that are inherited from its base classes (i.e., those classes that are higher-up on the inheritance hierarchy) so that those parameters can be changed from that point on the inheritance hierarchy without affecting the "upstream" (or adjacent) classes

Now that we have laid out these factors, we will break each one down to show how it is implemented.

Factor # 1

Every "template" class (and "built-in" component) contains a global parameters instance that can be accessed by calling statically calling the `GlobalParameters` property. For example, the global parameters for all implicit components can be accessed as follows:

```
ImplicitComponent.GlobalParameters
```

Factor # 2

Global parameters are implemented using a static (singleton) instance of a parameters class and are accessed statically using a static property. In other words, to implement global parameters for your custom component, you need to: first, setup a static instance of your component's parameters class as a static field within your component class; and second, setup a static property to access that static instance. The following code demonstrates how this might look:

```
public class SomeCustomComponent : ImplicitComponent
{
    ... //Elided code for the parameters and factory classes

    private static SomeCustomComponentParameters g_p =
        new SomeCustomComponentParameters();

    ... //Elided fields, constructors, methods, properties, etc.

    public static new SomeCustomComponentParameters GlobalParameters
    {
        get
        {
            return g_p;
        }
    }
}
```

Note that the `GlobalParameters` property has been specified using the "`new`" qualifier. This is done so that your component's `GlobalParameters` property hides the one that is "inherited" from the base class.[6]

Factor # 3

The global parameters instance is specified as being global during the initialization (i.e., in the constructor) of that instance. This is done because, as some of the later factors attest, there are special considerations that need to be taken into account regarding how parameters are changed globally. Therefore, an instance of a parameters class needs to know whether it is being used as a global instance. This can be implemented by updating your parameters class's constructors as follows:

```
//The constructors for the custom component's parameters class
public SomeCustomComponentParameters(bool isGlobal = false) : base(isGlobal)
{
    //Elided
}

public SomeCustomComponentParameters(SomeCustomComponentParameters p,
    bool isGlobal = false) : base(p, isGlobal)
{
    //Elided
}
```

The specification as to whether the instance is global gets stored by the bottom-most base class (i.e., in the `ClarionComponentParameters` class). However, it can

[6] Technically, classes don't "inherit" static members from their base classes, even though static base class members can be called statically from a subclass. As a result of this, if we didn't create a new `GlobalParameters` property, then calling that property statically from `SomeCustomComponent` would actually return the global parameters instance from `ImplicitComponent`.

be retrieved using the `IsGlobal` property that is inherited from `ClarionComponentParameters`. With the above changes, initializing the global instance of your parameters class actually changes to:

```
private static SomeCustomComponentParameters g_p =
    new SomeCustomComponentParameters(true);
```

Factor # 4

The global parameters instance is **ONLY** used during the initialization of a component as a means for setting the local parameters of that component. The global parameters instance should **NEVER** be used as part of the actual operation of the component. If you were to use a global parameter instead of a local parameter to perform calculations in your component, then it would make irrelevant any parameter changes that a user may wish to perform for a specific instance of your component. You should never assume that a user of your component will only want to make parameters changes for **ALL** instances of your component at the same time. Therefore, do **NOT** use the global parameter instance except to initialize the local parameters for an instance of your component.

In order to initialize a local parameters instance using the global parameters, you must first recall that there are two options for initializing local parameters class instances. The following lines of code demonstrate how you could initialize a local parameters instance using the global parameters instance via either option:

```
//Option #1 – In the constructor of the custom component
public SomeCustomComponent():
    base (new
        SomeCustomComponentParameters(SomeCustomComponent.GlobalParameters)) { }


//Option #2 – In the "Generate" method of the custom component factory
public SomeCustomComponent Generate(params dynamic[] parameters)
{
    //Elided code for parsing the initialization parameters

    if (pars == null)
        pars = new
            SomeCustomComponentParameters(SomeCustomComponent.GlobalParameters);
    else
        pars = new SomeCustomComponentParameters(pars);

    List<dynamic> fpars = new List<dynamic>();
    fpars.Add(pars);
    result.FactoryParameters = fpars.ToArray();

    return new SomeCustomComponent(pars);
}
```

Factor # 5

Global parameter changes can be made at any point along the inheritance hierarchy. Suppose, for instance, that you wanted to change the eligibility of all components.

The code below would change the eligibility for anything that extend from `ClarionComponent` (i.e., all components):

```
ClarionComponent.GlobalParameters.ELIGIBILITY = false;
```

This feature is made possible by event handlers that are located within the parameters classes at every point within the inheritance hierarchy. These event handlers handle `ParameterChangeRequestedEventArgs`. When a global parameters instance is initialized, its event handler method (which is inherited from `ClarionComponentParameters`) is added to the event handlers for all classes that are above that parameters class in the inheritance hierarchy.

Every parameters class needs to have its own even handler (although it does not need its own event handler method). The event handler is instantiated as a `static` field in your parameters class. For example, the following code sets up the global parameter change event handler for the `SomeCustomComponentParameters` class:

```
public class SomeCustomComponentParameters : ImplicitComponentParameters
{
    private static event EventHandler<ParameterChangeRequestedEventArgs>
        g_SomeCustomComponent_pEvent;

    //Elided fields, constructors, properties, etc.
}
```

Feel free to name the event handler whatever you want. However, the standard convention that is used in the Clarion Library is: *g_ClassName_pEvent*[7].

Now, you can simply add the event handler method (`Global_ParameterChanged`, inherited from `ClarionComponentParameters`) to your event handler. This is performed in the constructors of your parameters classes. For our `SomeCustomComponentParameters` example, the result might look like this:

```
//The constructors for the custom component's parameters class
public SomeCustomComponentParameters(bool isGlobal = false) : base(isGlobal)
{
    if (IsGlobal)
        g_SomeCustomComponent_pEvent += Global_ParameterChanged;

    //Elided
}

public SomeCustomComponentParameters(SomeCustomComponentParameters p,
    bool isGlobal = false) : base(p, isGlobal)
{
    if (IsGlobal)
        g_SomeCustomComponent_pEvent += Global_ParameterChanged;

    //Elided
}
```

[7] "*ClassName*" refers to the name of your component, **NOT** the name of the parameters class

Don't forget to add the event handler method to your even handler in **BOTH** constructors. Otherwise, your event handler could fail to register all of the global parameter instances.

Factor # 6

Changing a parameter globally at any point along the inheritance hierarchy will change the value of that parameter for all classes that are "downstream" from (i.e., the subclasses of) the class where the global parameter change was initiated. Whenever a global parameter change occurs, the property not only changes the value of that parameter for the class where the static parameter change occurred, but it also initiates a global parameter change event that propagates the parameter change to every class that derives from that class. For example, in the `LEARNING_RATE` property of our `SomeCustomComponentParameters` class, the following code would handle both the setting of the parameter as well as the initiation of the global parameter change event:

```
public virtual double LEARNING_RATE
{
    get { return lr; }
    set
    {
        if (IsGlobal && !ParameterChange_EventInvoked)
        {
            ParameterChange_EventInvoked = true;
            g_SomeCustomComponent_pEvent(this,
                new ParameterChangeRequestedEventArgs(
                    parameters:new ParameterTuple("LEARNING_RATE", value))
            lr = value;
            ParameterChange_EventInvoked = false;
        }
        else
            lr = value;
    }
}
```

It is important to know a few key things about this code. First, the "`if`" statement in the property's setter introduces a new term: `ParameterChange_EventInvoked`. To clarify, this is a `static` "flag" that is "inherited" from `ClarionComponentParameters` and is used to prevent recursion as the parameter change is propagated downward along the inheritance hierarchy. You **MUST** set this flag to `true` before your event handler's `Invoke` method is called and then set it back to `false` afterwards. Second, notice that we create something called a `ParameterTuple` as part of creating a new `ParameterChangeRequestedEventArgs`. The `ParameterTuple` is a core construct that we use to couple parameters and values within a single object. Finally, notice that the string, which is used as part of invoking the global parameter change event, is the same as the name of the property in which the event is invoked. This string is used to inform the event handler method as to which parameter is supposed to be updated for the other (downstream) parameters classes. The event handler method propagates the parameter change by using Reflection to "lookup" the appropriate property in the other (downstream) parameters classes. Therefore, it is essential that this string matches the name of the property **EXACTLY.**[8]

Factor # 7

The parameters class of a component class should override the properties of all of the parameters that are inherited from its base classes (i.e., those classes that are higher-up on the inheritance hierarchy) so that those parameters can be changed from that point on the inheritance hierarchy without affecting the "upstream" (or adjacent) classes. If you choose not to override the properties of the parameters that are inherited from your component's base classes, then any global changes to those parameters will initiate a global parameter change event at the level of the base class.

As a result of this, any other classes that also derived from that base class, would have their parameter updated as well. This could lead to unintended consequences, so, in general, it is usually a good idea to override **ALL** of the properties that are inherited from the base classes of your parameters class. For example, the `SomeCustomComponentParameters` class inherits the `ELIBIGILITY` property. The following code demonstrates how to override this property:

```
public new bool ELIGIBILITY
{
    get { return base.ELIGIBILITY; }
    set
    {
        if (IsGlobal && !ParameterChange_EventInvoked)
        {
            ParameterChange_EventInvoked = true;
            g_SomeCustomComponent_pEvent.Invoke(this,
                new ParameterChangeRequestedEventArgs(
                    parameters:new ParameterTuple("ELIGIBILITY", value)));
            base.ELIGIBILITY = value;
            ParameterChange_EventInvoked = false;
        }
        else
            base.ELIGIBILITY = value;
    }
}
```

Notice that, for the most part, the process for overriding a property is essentially the same as in your own parameters. This makes sense if your recall that the primary reason for overriding these properties in the first place is so that your event handler is invoked instead of the base class's.

That is everything you need to know in order to setup your parameters class to be able to act as either a global parameters instance or a local parameters instance. You should now be able to setup your parameters class, so we will move onto the final requirement for setting up a custom component.

[8] Third, you should specify your properties as being `virtual`. This way, if someone decides to derive a new custom component from your component, they will be able to override your properties as well.

Commiting and Retracting

By this point, you should have everything you need in order to implement both the functionality of your component as well as the "tuning" parameters that are used to optimize that functionality. However, an agent cannot begin using your component until it is committed. Once your component has been generated by the agent initializer, it is placed in a special "initializing" state until it is committed. This is done in order to help ensure that an agent remains operable once it begins interacting with the world.

To maintain this operability, your component **MUST** be put into a "read-only" (i.e., immutable) state during the commit process. More specifically, you must "lock down" those aspects of your component that would break the proper operation of your component if they were to be altered during runtime (i.e., after the agent is initialized and starts interacting with the world). For example, adding or removing nodes from the input, hidden, or output layers of a backpropagation neural network (i.e., a `BPNetwork` in the Clarion Library) would break the correct operation of that network. Therefore, the `Commit` method of the `BPNetwork` class locks down (i.e., makes "read-only") those layers.[9] You should consult the documentation of the `Commit` methods of your component's base classes to determine what functionality has already been provided for you.

Note that the "tuning" parameters (i.e., those parameters located in the parameters class of your component) do not need to be committed (i.e., made immutable) since changing these parameters will not affect your component's ability to maintain its operability.[10]

By default, your component inherits `Commit` and `Retract` methods from the "template" class (or other component class) from which it is derived. Therefore, if your component does not add any new functionality that could be broken if something within your component were to be manipulated during runtime, then you do not need to implement new `Commit` and `Retract` methods. Furthermore, your component also inherits the `IsReadOnly` property (from the `ClarionComponent` class), so if your component can be made immutable using only that property, then you may also not need to implement your own `Commit` and `Retract` methods. Otherwise, you **MUST** override the base class's `Commit` (and possibly `Retract`) method(s) and add whatever additional commit operations are needed to make your component immutable.

In addition to handling the "lock down" operations, the `Commit` method can also be used to "wire-in" those aspects of a component that cannot be initialized until after everything else has been setup. In fact, this is actually the more common use of the `Commit` method. For example, the weights and thresholds of a 3-layer neural network cannot be initialized until after the input, hidden, and output layers of that

[9] Actually, the `Commit` method of the `ImplicitComponent` class provides all of the functionality needed to "lock" the input & output layers for implicit components

[10] Although changing the parameters of your component will alter **HOW** it operates

network have been completely setup. Therefore, the NeuralNetwork “template” class puts off the initialization of these pieces until its Commit method is called.

Below is the general outline to use when implementing a Commit method in your component:

```
public override void Commit()
{
    if (!CommitLock.IsWriteLockHeld)
    {
        CommitLock.EnterWriteLock();

        //Call the base class’s “Commit” method
        base.Commit();

        //Perform whatever “lock downs” and “wire-ins” are needed here

        CommitLock.ExitWriteLock();
    }
    else
    {
        //Call the base class’s “Commit” method
        base.Commit();

        //Perform whatever “lock downs” and “wire-ins” are needed here
    }
}
```

There are two important points to be made regarding the above outline. First, since the Commit method may be called asynchronously, you should **ALWAYS** use the CommitLock (which is inherited from ClarionComponent) to enter a “write lock” before performing the operations needed to commit your component.[11] In addition, your Commit method should also make sure to check that the “write lock” is not already open. If it is, then you can assume that your Commit method is being called from within the Commit method of a subclass of your component (see details regarding this below) and that it is safe for your commit operations to be performed without having to enter (or exit) the write lock.

Second, you should **ALWAYS** call the base class’s Commit method **BEFORE** performing the operations to commit your component. This way, you can make sure that all of the functionality inherited by your component is made immutable first. Furthermore, if there is any functionality in your component that cannot be initialized until after other functionality (that is inherited from the base class) is initialized, then calling the base class’s Commit method first will ensure that the base class’s functionality is initialized before your component’s functionality is initialized.

Moving over to “retracting”, this process is essentially just the reverse operation as “committing.” In other words, retracting your component will take the component out of “runtime” operation and place it back in the “initializing” state. Afterward,

[11] Don’t forget to exit the “write lock” after your commit operations are finished as well

your component should once again be "editable" (i.e., mutable). The `Retract` method allows users to make changes to components "on-the-fly" without damaging the operability of the overall agent.

It must be noted, however, that the retract feature comes with some drawbacks. For example, retracting a `BPNetwork` will allow its layers to be edited. However, since editing the layers of the network will affect the connections (i.e., the weights and thresholds) between these layers, the `BPNetwork` **MUST** be reinitialized when it is recommitted. This means that, when a `BPNetwork` is retracted, **ALL** learning that has been performed on that network will be lost. You component should try, as much as is possible, to preserve its state between retract and recommit phases. However, if this is not possible, then you need to warn users (in the documentation of your component's `Retract` method) about the consequences of retracting your component.

The general outline for implementing the `Retract` method is basically the same as for the `Commit` method:

```
public override void Retract()
{
    if (!CommitLock.IsWriteLockHeld)
    {
        CommitLock.EnterWriteLock();

        //Call the base class's "Retract" method
        base.Retract();

        //Perform whatever "unlocks" are needed here

        CommitLock.ExitWriteLock();
    }
    else
    {
        //Call the base class's "Retract" method
        base.Retract();

        //Perform whatever "unlocks" are needed here
    }
}
```

Using the `InitializeOnCommit` Property

While retracting usually requires that a component be reinitialized when it is recommitted, it is possible (and sometimes even necessary) to set up a component to "overlook" certain parts of the commit process by making use of the `InitializeOnCommit` property flag. When this flag is set to `true`, the `Commit` method will perform its initializations as normal. However, if the flag is set to `false`, then the process will skip over those initialization steps that are contained within any `if` statements that make use of that flag. The following code demonstrates how the `InitializeOnCommit` flag could be applied to our general `Commit` method outline:

```
public override void Commit()
{
    if (!CommitLock.IsWriteLockHeld)
    {
        CommitLock.EnterWriteLock();

        //Call the base class's "Commit" method
        base.Commit();

        //Perform whatever "lock downs" are needed here

        if (InitializeOnCommit)
            Initialize();    //Perform whatever initializations are needed here

        CommitLock.ExitWriteLock();
    }
    else
    {
        //Call the base class's "Commit" method
        base.Commit();

        //Perform whatever "lock downs" are needed here

        if (InitializeOnCommit)
            Initialize();    //Perform whatever initializations are needed here

    }
}
```

The `Commit` method still needs to perform all of the appropriate operations to lock the component (i.e., make it read-only). However, by using the `InitializeOnCommit` property, the users of your custom component will have the choice of whether or not they want the component to be initialized when it is committed. Note that a user **MUST** initially commit a component with the `InitializeOnCommit` flag turned on. Otherwise, the component will **NOT** operate correctly. Additionally, if a component is retracted and any of the inputs or outputs (etc.) are altered, then the `InitializeOnCommit` flag **MUST** also be turned on (for the same reason as before).

Furthermore, as you will see later, using the `InitializeOnCommit` flag is necessary to correctly reload our component using the `SerializationPlugin` (but we'll get into this later).

How to Implement a Custom (Secondary) Drive

In addition to defining the primary drives, the Clarion theory also specifies a thing called "secondary" (or derived) drives. Conceptually, these drives are the result of a combination of various primary drives that are typically not an inherently derived (or evolutionarily evolved) motivation. However, we contend that one could reasonably argue that, over time, these sorts of drives become independent motivators of behavior. For example, humans do not inherently have a desire to smoke cigarettes. Most people choose to smoke in order to attend to certain primary motivations, such as: affiliation & belongingness (to fit in), similance (because others are doing it), or possibly avoiding unpleasant stimulus (e.g., to manage stress). However, as people continue to smoke and their brains become "chemically addicted", the "drive to maintain nicotine levels" may, in and of itself, replace the other (primary) drives as the fundamental motivation for the "smoke a cigarette" behavior.

The Clarion Library comes prepackaged with all of the primary drives. These drives will likely be sufficient for most tasks. However, if you find that you need to specify a secondary drive as part of the setup of your agent, the library provides the abstract `Drive` class as a template for creating your own custom (secondary) drive.

Implementing a custom (secondary) drive is very similar to setting up a custom component (in fact, in many ways, it may even be simpler). To walk you through the process, we will use the "maintain nicotine levels" example from earlier. Below is a demonstration of how we could declare the `NicotineDrive` class:

```
public class NicotineDrive : Drive
{
    //The custom drive constructor
    protected NicotineDrive(Guid agentID, DriveParameters pars,
        double initialDeficit) : base(agentID, pars, initialDeficit) { }

    ...
}
```

The first thing to note about setting up a secondary (derived) drive is that we do **NOT** need to override any of the methods from the base `Drive` class. Since the `Drive` class is basically just a wrapper for an `ImplicitComponent`, its primary function is simply to use that component to calculate the drive strength. Therefore, if you want to customize the functionality of your secondary (derived) drive, you will need to implement a custom component for your drive.

Implementing the Nested "Factory" Class

For the next step, as was the case for implementing a custom component, we need to setup a "factory" class for initializing our custom drive. Note that, in addition to specifying the agent's world ID, a parameters class, and the initial deficit for the drive, the base `Drive` class's constructor can also accept two optional parameters: the drive's group[12], and a `DeficitChangeProcessor` delegate. The following code demonstrates how we could setup the `NicotineDriveFactory` as a "nested class" within the `NicotineDrive`:

```
public class NicotineDrive : Drive
{
    public class NicotineDriveFactory : IDriveFactory<NicotineDrive>
    {
        public NicotineDrive Generate(params dynamic[] parameters)
```

[12] Using the `MotivationalSubsystem.DriveGroupSpecifications` enumerator

```
        {
            ... //Elided code for parsing-out the parameters

        }
    }

    protected NicotineDrive(Guid agentID, NicotineDriveParameters pars, double
            initialDeficit, DeficitChangeProcessor deficitChangeMethod = null)
            : base(agentID, pars, initialDeficit, deficitChangeMethod){ }

    ... //Elided drive class code
}
```

To handle all of the parameters (both optional and required) for initializing the `NicotineDrive`, we could set up the `Generate` method (within the `NicotineDriveFactory`) as follows:

```
public NicotineDrive Generate(params dynamic[] parameters)
{
    Guid aID = Guid.Empty;
    double iD = -1;
    DeficitChangeProcessor d = null;
    NicotineDriveParameters dp = null;
    MotivationalSubsystem.DriveSystemSpecifications ds =
        MotivationalSubsystem.DriveSystemSpecifications.BOTH;
    foreach (dynamic p in parameters)
    {
        if (p is double)
            iD = p;
        else if (p is DeficitChangeProcessor)
            d = p;
        else if (p is NicotineDriveParameters)
            dp = p;
        else if (p is MotivationalSubsystem.DriveSystemSpecifications)
            ds = p;
        else if (p is Guid)
            aID = p;
    }

    if (aID == Guid.Empty)
        throw new ArgumentException("You must specify the agent to which this
            drive is being attached in order to generate the drive");

    if (iD == -1)
        throw new ArgumentException("To initialize a drive, you must specify an
            initial deficit.");

    if (dp == null)
        dp = new NicotineDriveParameters(g_p);
    else
        dp = new NicotineDriveParameters(dp);

    if(dp.DRIVE_SYSTEM == MotivationalSubsystem.
        DriveSystemSpecifications.UNSPECIFIED)
        dp.DRIVE_SYSTEM = ds;

    return new FoodDrive(aID, dp, iD, d);
 }
```

Once the factory (including the `Generate` method) class has been setup, we need to specify the `static` factory instance and the `static` `Factory` property for accessing it within the `NicotineDrive`:

```
public class NicotineDrive : Drive
{
    private static NicotineDriveFactory factory =
        new NicotineDriveFactory();

    ... //Elided factory and parameters classes, constructors, etc.

    public static NicotineDriveFactory Factory
    {
        get
        {
            return factory;
        }
    }
}
```

Implementing the Nested "Parameters" Class

Just like we did for the custom component, we also need to setup a parameters class for our custom drive. The following code demonstrates how we would declare the `NicotineDriveParameters` class within our `NicotineDrive`:

```
public class NicotineDrive : Drive
{
    public class NicotineDriveParameters : DriveParameters
    {
        private static event
            EventHandler<GlobalParameterChangedEventArgs> g_NicotineDrive_pEvent;

        public NicotineDriveParameters(bool isGlobal = false) : base(isGlobal)
        {
            if (IsGlobal)
                g_NicotineDrive_pEvent += Global_ParameterChanged;
        }

        public NicotineDriveParameters(NicotineDriveParameters p,
            bool isGlobal = false) : base(p, isGlobal)
        {
            if (IsGlobal)
                g_NicotineDrive_pEvent += Global_ParameterChanged;
        }

        ... //Elided parameter properties
    }

    ... //Elided factory class, etc.
}
```

Unlike a custom component, however, we will likely **NOT** need to define any new parameters for our custom drive. This is the case because we do not alter the functionality of the drive itself (by overriding its methods). Therefore, we also do not need to define new parameters for our drive. This being said, though, you will still probably want to create a parameters class for your custom drive in order to handle global (`static`) parameter changes for all of the parameters that are inherited from the base `Drive` class. For our `NicotineDriveParameters` example, the following code demonstrates how we might accomplish creating `new` parameter properties at the `NicotineDrive` level of the inheritance hierarchy:

```
public new double DEFICIT_CHANGE_RATE
{
    get { return base.DEFICIT_CHANGE_RATE; }
    set
    {
        if (IsGlobal && !ParameterChange_EventInvoked)
        {
            ParameterChange_EventInvoked = true;
            g_NicotineDrive_pEvent.Invoke(this,
                new ParameterChangeRequestedEventArgs(
                    parameters:new ParameterTuple("DEFICIT_CHANGE_RATE", value)));
            base.DEFICIT_CHANGE_RATE = value;
            ParameterChange_EventInvoked = false;
        }
        else
            base.DEFICIT_CHANGE_RATE = value;
    }
}

public new double DRIVE_GAIN
{
    ... //Same as above, except for the DRIVE_GAIN parameter
}

public new double BASELINE
{
    ... //Same as above, except for the BASELINE parameter
}
```

Once the parameters classes is set up, our final step is to specify properties within the `NicotineDrive` for the global (`static`) parameters and local parameters instances. This can be accomplish by doing the following:

```
public class NicotineDrive : Drive
{
    private static NicotineDriveParameters g_p =
        new NicotineDriveParameters(true);

    ... //Elided factory and parameters classes, constructors, etc.
    public static new NicotineDriveParameters GlobalParameters
    {
        get { return g_p; }
    }

    public new NicotineDriveParameters Parameters
    {
        get { return (NicotineDriveParameters) base.Parameters; }
    }
}
```

You should now have all of the information you need to implement a custom (secondary) drive within the Clarion Library. In the following (final) section of this guide, we will talk about how you can setup your drive (or custom component) so that it can be loaded and unloaded using serialization.

Serializing a Custom Component (or Drive)

The final step when implementing a custom component (or drive for that matter) is to make your component serializable. This step is optional, however, you should note that all of the "built-in" objects (including both descriptive and functional objects) throughout the Clarion Library are serializable. This has been done in order to provide you with a means for loading and unloading both descriptive objects (i.e., those objects contained within the `World`) as well as functional objects (i.e., all of the agents' internals). This feature is implemented by leveraging attributes (and in particular the `DataContract` and `DataMember` serialization attributes). By making your component serializable, users will be able to use the library's built-in `SerializationPlugin`[13] (or C#'s `DataContractSerializer`) to load and unload your custom component (or drive).

The process of implementing the `DataContract` and `DataMember` serialization attributes is actually fairly simple and straightforward. In fact, Microsoft's MSDN API resource for the `DataContractSerializer` already provides an excellent explanation for how and when to make use of these attributes, so we will forgo the particulars in this tutorial. Instead, in the following subsections, we will use the `SomeCustomComponent` class that we set up earlier to demonstrate how these attributes can be applied to a component (or drive).

Specifying the *System.Runtime.Serialization* Resource

Before we describe the process for making our components serializable, you should be aware that all of the serialization mechanisms that we discuss in this document are defined within C#'s *System.Runtime.Serialization* assembly. This assembly is usually not included as part of the default libraries that get loaded when a project is created. Therefore, you will likely need to manually specify this assembly as a resource in order to setup the serialization capabilities for your component.

[13] See the "*Using Plugins*" tutorial in the "*Features & Plugins*" section of the "*Tutorials*" folder

To use the *System.Runtime.Serialization* assembly, we must add it as a resource to our project. Accomplishing this tends to vary based on the development environment, so you should consult the guides for your particular one if you need help with how to do this. However, in general, the process usually involves something like the following:

- Under your project (in the solution explorer), there is a “folder” named something like “resources” (or possibly “references”). Right-click on that folder and choose the “add” menu item from the drop-down.
- In the window that comes up, navigate to the “built-in libraries” section and select the “*System.Runtime.Serialization*” assembly.

Once you have completed these steps, the *System.Runtime.Serialization* assembly should appear in the “resources” (or “references”) section under your project in the solution explorer. If it is listed there, then you have successfully specified the *System.Runtime.Serialization* assembly for your project. The only other step you will need to do in order to use it is to specify the serialization namespace[14] at the top of the file containing your custom component (with a `using` clause):

```
using System.Runtime.Serialization;
```

The `DataContract` Attribute

The first thing we need to do is specify that our component is serializable. This is done by adding the `DataContract` attribute above the class declaration:

```
[DataContract(Namespace = "ClarionLibrary")]
public class SomeCustomComponent : ImplicitComponent
{

    ... //Elided class code

}
```

Note that the `Namespace` parameter for the `DataContract` attribute has been assigned the `"ClarionLibrary"` value. You can feel free to rename this if you’d like, however, by convention, all of the classes in the Clarion Library are serialized using this namespace.

The `DataContract` attribute needs to be specified for all of the classes that we want to be serialized. Therefore, since we will likely want to serialize our component’s parameters, we are going to need to specify this attribute for the `SomeCustomComponentParameters` inner class that we setup inside of our `SomeCustomComponent`:

[14] Also named *System.Runtime.Serialization*

```
[DataContract(Namespace = "ClarionLibrary")]
public class SomeCustomComponent : ImplicitComponent
{
    [DataContract(Namespace = "ClarionLibrary")]
    public class SomeCustomComponentParameters : ImplicitComponentParameters
    {
        ...
    }

    ... //Elided class code

}
```

Recall that we also setup a “factory” class (`SomeCustomComponentFactory`) inside of our component, however, since this class only contains a single method (`Generate`) and is only initialized statically, it does not need to be serialized. Therefore, we do not need to specify the `DataContract` attribute for this class.

Specifying the `DataContract` attribute for the `SomeCustomComponent` and `SomeCustomComponentParameters` classes will indicate to the system that these classes can be serialized. However, we still need to specify which parts of the class will be serialized. We do this using the `DataMember` attribute.

The `DataMember` Attribute

The next thing we need to do to make our component serializable is to specify the `DataMember` attribute for all of the fields whose settings are important for making sure the component runs correctly when it is “re-serialized” by the system. The decision as to which fields to include depends on the specifics of the component. However, in general, you will normally want to serialize any fields that are either required as part of the initialization process, or are “locked-down” during the commit process. For instance, in our `SomeCustomComponent` example, we will want to serialize the “nodes” that were generated during the initialization of the component.

Let’s suppose that these “nodes” are of a special `Node` type and that we store these `nodes` using C#’s built-in generic `List<T>` collection. To serialize our `nodes`, we need to add the `DataMember` attribute above the line where they are declared:

```
[DataContract(Namespace = "ClarionLibrary")]
public class SomeCustomComponent : ImplicitComponent
{
    [DataMember(Name = "Nodes")]
    private List<Node> nodes;

    ... //Elided additional class code
}
```

This code will indicate to the system that the `nodes` field should be serialized as part of our component. Additionally, the `Name` parameter (within the `DataMember` attribute) specifies that the field should be assigned the `"Nodes"` tag. As a rule of thumb, you should avoid using spaces for the `Name` parameter of the `DataMember`.

This being said, your component will still serialize correctly, however, the XML file (or stream) that results from serialization will be much cleaner and more readable if you avoid using spaces.

Continuing on, recall that we also need to serialize all of the parameters for our component (located in the SomeCustomComponentParameters class). For example, remember that the SomeCustomComponentParameters class has a “learning rate” parameter (declared using the lr field). To specify that this parameter should be serialized, we can do the following:

```
[DataContract(Namespace = "ClarionLibrary")]
public class SomeCustomComponentParameters : ImplicitComponentParameters
{
    [DataMember(Name = "LearningRate")]
    private double lr = .1;
}
```

Note that the local parameters instance is stored at the base level of our component (i.e., in the ClarionComponent class) and has already been setup with the necessary DataMember attribute (using the "Parameters" tag). Therefore, once we have specified the DataMember attribute for all of our component’s parameters, the system will have everything it needs to serialize the local instance of the SomeCustomComponentParameters class.

If you recall, from the “*Using Plugins*” tutorial in the section on how to use the SerializationPlugin, when a component is reloaded (i.e., deserialized), it is automatically recommitted. Of course, you likely will **NOT** want your component to reinitialize itself when it is recommitted. With this in mind, as part of the serialization process, the SerializationPlugin sets the value of the InitializeOnCommit parameter to false. This is done so that when the component is deserialized, it can be recommitted without losing any of its settings. Remember that earlier in this tutorial we explained how to setup the Commit method of your custom component using the InitializeOnCommit property flag so that the process will skip the initialization steps. If your custom component is going to be serializable, you will need to make sure you use this flag in the Commit method of your component in order to avoid the loss of any settings following the deserialization process when using the SerializationPlugin.

At this point, let’s take a moment to discuss some pre and post serialization and deserialization customizations that are available.

Pre/Post Serialization and Deserialization Attributes

As part of the serialization process, C# provides several attributes that you can specify in conjunction with methods that the system will use to perform any operations that may be necessary prior to or following the loading or unloading of a component. These attributes include: OnSerializing, OnSerialized, OnDeserializing, OnDeserialized.

To implement a method for performing pre or post serialization or deserialization, we will first need to specify the appropriate attribute above the declaration for the method that we wish to perform these operations. The code below demonstrates how we might setup a method to handle the post deserialization processes:

```
[OnDeserialized]
void CompleteDeserialization(StreamingContext sc)
{
    ... //Elided post deserialization code
}
```

The most common place where we need to implement one of these methods is in the parameters class for our custom component. Specifically, within the parameters class, we need to set up a method that will reregister the global parameters instance to our "global parameter change" event handler. To accomplish this, for our `SomeCustomComponent` example, we could do the following (within the `SomeCustomComponentParameters` class):

```
[OnDeserialized]
void CompleteDeserialization(StreamingContext sc)
{
    if (IsGlobal)
    {
        g_SomeCustomComponent_pEvent += Global_ParameterChanged;
    }
}
```

Note that the specifics as to the sorts of things that should be performed within the pre or post serialization or deserialization methods depends on the particulars of the class. So, at this point, we will not be able to delve into this topic any further. However, if you run into problems setting up a pre or post serialization or deserailzation method within your custom component, drive, or parameters class, then we suggest that you consult Microsoft's MSDN API resources or search the Internet for additional help.[15]

Serializing the Global (`static`) Parameters

First, we should mention is that `static` fields are **NOT** serialized as part of the process for serializing a class. Given this, there is no reason for us to specify the `DataMember` attribute above the global (`static`) parameters instance declaration as it will have no effect.[16] This being said, however, there is another way for us to setup our component so that it can still serialize and deserialize the global parameters instance. Specifically, we can create a `private` property that gets and sets the global (`static`) parameters instance and then specify the `DataMember` attribute for that property. By doing this, the global parameters instances (including the global parameters for all base classes) for our component will also be serialized.

[15] You can also contact us at clarion.support@gmail.com for assistance once you have exhausted all other avenues. However, if you do decide to contact us, then please provide **clear details** regarding the **exact** nature of the issues you are having.
[16] Although it certainly will not break anything either

The following code demonstrates how we might setup this "global serialization property" for the SomeCustomComponent example:

```
[DataMember(Name = "GlobalParametersInstance")]
private SomeCustomComponentParameters GoalParametersSerialization
{
    get
    {
        return g_p;
    }
    set
    {
        g_p = value;
    }
}
```

We should note here that, by using this method for serializing the global parameters, all of the individual instances of your component will also offload a copy of the global parameters instance when they are serialized. As a result, every time we reload an instance of our component, the global parameters instances (including the global parameters instances for the component's base classes) will be replaced (which effects **ALL** instances of the component). Therefore, as we mentioned in the tutorial for the `SerializationPlugin`[17], you need to make sure that you perform **ALL** deserialization **BEFORE** making changes to **ANY** global (`static`) parameters.

You should now have everything you need in order to implement your own custom components within the Clarion Library. However, as always, if you have any questions, want to submit a bug, or make a feature request, please feel free to post on our message boards (http://www.clarioncognitivearchitecture.com) or email us at clarion.support@gmail.com and we will do our best to respond back to you as quickly as possible.

[17] See the "*Using Plugins*" tutorial in the "*Features & Plugins*" section of the "*Tutorials*" folder