

A Brief Note Before You Begin...

First, thank you for downloading the Clarion Library, version 6.1.1!

This is the 1st release since April of 2012 and as such represents a significant enhancement over 6.1.0.7. If you are not already familiar, the Clarion Library is the premier implementation of the Clarion cognitive architecture (a theory by Ron Sun). I hope you enjoy all of the new features and capabilities that have been added in this release, including:

- **IMPORTANT:** The name of the DLL and Serialization namespace has changed
 - **CLARIONLibrary** to **ClarionLibrary**
 - **Note:** definite implications for simulations using 6.1.0.7
 - Make sure you update the DLL reference in your projects!
- Fully implemented NACS reasoning and retrieval
 - Now interacts with the ACS and MCS modules
 - Added **InputFilterer** and **KnowledgeFilterer** delegates
 - Can be specified in the **ActionCenteredSubsystem** to do custom filtering to & from the NACS
 - Episodic memory and offline ACS/NACS learning currently in development (slated for the next release)
- Core improvements
 - Refactored event timing and truly asynchronous event invocation
 - Threads now only start when needed (improved performance & memory)
 - Improved how **SensoryInformation** is propagated and updated
 - Added/moved several (inner) “tuple” classes to *Clarion.Framework.Core*
 - Added various core “tracking” classes (e.g., a reasoning tracker for the NACS)
 - Added several new interfaces and delegates
- Added new concept **MetaInfoReservations**
 - Replaces things like **TypicalIO** in **Drive**
 - New formatting for accessing this meta information in the **SensoryInformation** object
 - **Note:** possible implications for simulations using 6.1.0.7
- Redesigned **Chunk** class(es)
 - Chunk weight and strength calculation methods relocated
 - Custom delegate options for both methods
 - Added dimension weight specifications
 - Implements bottom-up “weight matrix” concept (from CLARION-H addendum)
 - Allows for “dimension NOT activated” specification
 - Replaces `dvRepDimension` specification in the `New...Chunk` methods in **World**
 - **Note:** possible implications for simulations using 6.1.0.7
 - New **ConditionalChunk** class
 - Replaces inner **Rule.Condition** class
 - New **CustomMetaCognitiveActionChunk** class
 - Initialized using **World**
- Input and Output layers in implicit components (e.g., **BPNetwork**) are now specified by **InputOutputLayer** instead of **ActivationCollection**
- Feature Enhancements
 - Added “switch” (i.e., `PERFORM_LEARNING` parameter) to turn on/off ALL learning in **ActionCenteredSubsystem** and **MetaCognitiveModule**

- New LOCAL_EPISODIC_MEMORY_RETENTION_THRESHOLD in [ActionCenteredSubsystem](#)
 - Now only retains previous 10 episodes in local memory (by default)
 - Allows for better memory management
- Added time stamps to trace logging
- Added variability options to perception and actuation response times (credit Emily O’Leary)
 - New parameters/delegates in [Agent](#) to facilitate this feature
- Added optional [out](#) parameter to [GetChosenExternalAction](#) (and asynchronous method) to return the activations for all actions that were considered during action decision-making (credit Shane Bretz)
- Various improvements/simplifications in [MetaCognitiveModule](#)
 - Added [OperationOptions](#) and [OutcomeActivationOptions](#)
 - Added [OnTimedEvent](#) method
- Changed [Deserialize](#) methods in [SerializationPlugin](#) to use the [out](#) parameter concept (instead of returning the object)
 - Allows for a cleaner serialization code
 - **Note:** possible implications for simulations using 6.1.0.7
- Miscellaneous fixes and additions in various extension classes (i.e., in [Clarion.Framework.Extensions](#))
 - Moved [Set/RemoveRelevance](#) methods from [GoalSelectionEquation](#) to [GoalSelectionModule](#)
 - **Note:** possible implications for simulations using 6.1.0.7
 - Added [GoalSelectionModuleParameters](#)
- Tutorials and documentation have been added to and updated
 - Added Tutorial Table of Contents
 - Gives general ordering to tutorials, plus makes it easier to determine where specific topics are located
 - Still incomplete (slated for the next release)
- New Samples
 - Inductive Reasoning (credit Dan Cannon)
 - Full Reasoner (credit Shane Bretz)
- Many bug fixes!!!

As always, I encourage you to take a moment to peruse all of the documents that we have provided as part of this package; especially the “Getting Started” guide as well as the other tutorials in the “Tutorials” folder. Also, make sure that you read through and agree to all of the licensing terms and conditions before you start using the library.

Note that this is still technically a beta release (although it is getting very close to being an official release). With this in mind, you should be aware that some aspects of the library are incomplete at this point or are still in development.

If you have any questions, want to submit a bug, or make a feature request, please feel free to post on our message boards (at <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.

Thank you again for downloading and trying out the Clarion Library!

Sincerely,



Nicholas Wilson Ph.D.
Lead Developer, The Clarion Team

Getting Started

© 2013. Nicholas Wilson

Table of Contents

Introduction	1
Why C#?.....	2
Installation.....	2
About the Clarion Library package	3
Creating a Simulation Project	4
Referencing the Clarion Library.....	4
The Clarion Library Namespaces.....	5
Basic Guideline for Setting Up Simulations	6
Descriptive vs. Functional Objects.....	6

Introduction

Welcome to Clarion! This guide is intended to help you get started working with the Clarion Library so that you can create and run your own task simulations using the Clarion cognitive architecture as your foundation. Clarion is a modern, true hybrid cognitive architecture that has been under development for many years by Dr. Ron Sun. Version 6.1.1 of the Clarion Library is the latest version, and is significantly more powerful and easier to use than its predecessors.

To get the most out of this guide, you should already have a basic knowledge of the Clarion theory and of cognitive modeling in general. Ideally, you should also be somewhat familiar with an object-oriented programming language (like Java, C++, or C#), and with [object oriented programming](#) (OO) in general. The Clarion Library, version 6.1, is written in C#, however if you are used to Java, do not be alarmed, as the languages are actually quite similar. If you already know Java, learning the minimal amount of C# that you need will not be very difficult.

In order to help you get familiar with C# and object-oriented programming we have provided links (like the one above), throughout this guide, to various resources on the web that we think may be helpful for gaining some extra background on certain topics and concepts that we will be discussing. So keep your eyes out for those links and feel free to click on them if you feel you need clarification on something.

Also, while we are certainly not in the business of advertising, if you are looking for a good reference manual on C#, we would highly recommend "[C# \[4.0\] in a Nutshell](#)" from O'Reilly Books.

Why C#?

Beginning with version 6.1, the Clarion Library is now written entirely in C#, while all previous versions were written in Java. “Why the switch to C#?”, you may ask. Well, we had several reasons for abandoning Java and switching to C#. First, while both languages are equally “expressive,” we found it much easier to build all of the new features and capabilities for version 6.1 of the Clarion Library in C#. This is owing to several language constructs in which Java is either lacking or (in our opinion) less proficient. Furthermore, C# provides many enhancements over Java, such as [delegates](#), anonymous ([lambda](#)) functions, [dynamic](#) binding, enhanced [event](#) mechanisms, simple and straight-forward [serialization](#), and [LINQ](#). These feature (among others) have had the effect of greatly simplifying several aspects of the Clarion Library and we are confident that once you begin working with it, you will agree.

We should also note that, while it was originally developed by Microsoft, C# was and is intended to be an open language with both ECMA and ISO standards behind it. So those of you who tend to be afraid of that “evil monopoly” should rest easy. C# can be developed perfectly well on other operating systems by making use of the open-source (and free) Mono project development environment ([Mono Develop](#), also known as **Xamarin Studio**). We will also point out (again, we are not in the business of advertising here) that if you prefer developing in Windows, Microsoft also provides a free version of their software ([Visual Studio Express](#)). The Clarion Library has been tested using both of these development environments, so please feel free to use whichever you prefer.

At this time, we won't go any further into discussing why we chose C# over Java (or a different language for that matter). The debate between the “superiority” of various programming languages (and styles) is ongoing and eternal. It would not serve the purpose of this document for us to spend any more time on discussing the benefits, differences, and/or shortcomings of C# versus Java. However, for those of you who are interested in further reading, a very thorough breakdown between C# and Java can be found online on [Wikipedia](#).

We hope, however, that you are at least intrigued enough to want to try out this new system, so let's jump into it.

Installation

To begin, you need to know that the new Clarion Library leverages many of the features specific to version 4.0 of the .NET framework. Therefore, regardless of which development environment you use, it is import to make sure that you have the most recent .NET (or Mono) framework installed on your machine. If you are using Windows XP (service pack 2) or later, the newest .NET framework should be

downloadable through Windows update (if it is has not been automatically installed already). Otherwise, the mono framework can be found [here](#).¹

Installing either of the development environments that we have been discussing so far is fairly straightforward and instructions can easily be found on their websites, so we won't get into the details for installing these development environments here. Instead, we are just going to assume at this point that you have successfully installed the appropriate framework and development environment and are ready to begin.

About the Clarion Library package

To start, since you are reading this document, we can assume that you have already downloaded the zip file for the Clarion Library and have unzipped it to a folder somewhere on your hard drive. Within this folder you will find several things along with this guide. Specifically, you should note the following contents:

- The Getting Started guide (this document)
- The Clarion Library dynamic link library (dll)
 - The “assembly” (i.e., resource library)
- Clarion Library IntelliSense XML file
- A “Tutorials” folder
 - Contains guides (like this one) for configuring and customizing various aspects of the Clarion Library
- A “Samples” folder
 - Contains basic, intermediate, and advanced simulation samples
- A “Documentation” folder
 - Contains “MSDN-like” API reference documentation for the library

While we have striven, in this implementation, to simplify the process as much as possible, there can still be a bit of a learning curve when you are first starting with the Clarion library (and theory for that matter). Therefore, we encourage you to peruse the various documents, tutorials, and samples that have been included with the Clarion library before you begin using it. These items are provided in order to help clarify any confusion that may arise while you are developing your simulations. Additionally, taking the time to peruse the tutorials first should help minimize the amount of time it takes you to become a competent builder of Clarion-based agents.

Let's turn our attention now to setting up a simulation. We will also go over a few key concepts that you should keep in mind when using the Clarion Library.

¹ The individual implementations for the mono framework and development environment vary somewhat based on the operating system. Therefore, you may need to do some separate searching online to find the latest builds of mono for your OS.

Creating a Simulation Project

The first thing you need to do when setting up a simulation is to specify a “solution” for your task simulations. If you have not already created one, you should do so now. The specifics on how to do this varies slightly based on the development environment you are using, so please consult the guides for your particular environment if you need help creating a new solution. Feel free to call your solution whatever you would like (we suggest calling it “Clarion Simulations”).

After we have created the solution, the next step is to create a new project (again, please consult the guides for your particular development environment if you need help with how to do this). A solution can house many different projects, so you do not need to create a new solution for every simulation you are going to write. However, each simulation should be in its own project. Creating a new project can usually be done within the “solution explorer” by simply right-clicking on your solution and choosing the “new project” option under the “add” menu item. Again, feel free to name the project whatever you would prefer. We recommend that you name your project so that it succinctly describes your simulation or task (e.g., “*SimpleHelloWorld*”, as will be demonstrated later in the “*Setting Up & Using the ACS*” tutorial).

Referencing the Clarion Library

If you are somewhat unfamiliar with programming, we will need to spend a few moments discussing the concept of a “resource” (if you are familiar with it then feel free to skip this part). A resource is a collection (i.e., assembly or library) of objects (classes, interfaces, delegates, etc.) that you can use within your own code. The Clarion Library is simply a collection of this nature. In other words, the Clarion Library is not, in and of itself, executable. Instead, it provides you with the necessary tools in order to build Clarion-based “agents” within your own executable simulating environment. In fact, the files located in the “Samples” folder all serve as examples of this sort.

To use the Clarion Library in our project, 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 location where you unzipped the Clarion Library, and select the “ClarionLibrary.dll” assembly file.

Once you have completed these steps, the Clarion Library 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 Clarion Library for your project, and will be able to use it.

Now that you have the Clarion Library resource referenced, let's look at how the library is structured.

The Clarion Library Namespaces

Clarion				
Clarion.Framework			Clarion.Plugins	Clarion.Samples
Clarion. Framework. Core	Clarion. Framework. Extensions ³	Clarion. Framework. Templates		

The table above provides you with a hierarchal breakdown of the namespaces for the Clarion Library. You can consult the API reference document to get a complete list of all the classes within each namespace. However, in general, the namespaces are organized as follows:

- **Clarion** – This is the base namespace for the Clarion Library and it contains three classes: [World](#), [AgentInitializer](#), & [ImplicitComponentInitializer](#).
- **Clarion.Framework** – This namespace contains the majority of the classes needed for initializing and running a simulation. Most of the classes contained within this namespace can be correlated directly to terms or concepts from the Clarion theory.
- **Clarion.Framework.Core** – This namespace contains the necessary constructs for the core operation of the system.²
- **Clarion.Framework.Extensions³** – This namespace contains extensions (e.g., meta-cognitive modules, various components, etc.) for the Clarion Library that, while not (necessarily) being specified within the Clarion theory itself, can still be used within an agent in the same fashion as the classes found in the `Clarion.Framework` namespace.
- **Clarion.Framework.Templates** – This namespace contains abstract classes, interfaces, and delegates, etc. that act as "templates" for building some custom user-defined objects (e.g., implicit components, drives, etc.).
- **Clarion.Plugins** – This namespace provides various plugins and tools that may be used in by a simulating environment to enhance the capabilities and applications of Clarion-based agents setup using the Clarion Library.
- **Clarion.Samples** – This namespace contains several simulating environment examples that serve as guides to help you learn how to use the Clarion Library.

² You should rarely need to access this namespace in order to initialize or run an agent.

³ Note that an additional namespace, *Clarion.Framework.Extensions.Templates*, has been added, which provides some useful templates for creating extensions

We should note that the `Clarion.Samples` namespace is not actually in the assembly (i.e., dll file). Instead, the files that constitute this namespace are located in the “Samples” folder. These samples have been placed in the “Samples” namespace because they are provided as part of the overall Clarion Library package (i.e., zip file), and thus can be considered as being a tangential part of the library.

Basic Guideline for Setting Up Simulations

First, we should stress that there are **NO** “requirements” for setting up a simulation (although there are certainly some requirements for setting up an agent). If you have any prior experience with developing simulations using cognitive architectures, you are probably aware of just how quickly it can become very difficult to maintain a strict format for developing simulations, especially as they increase in complexity. With this in mind, in the Clarion Library, we have resisted imposing rigid guidelines when creating simulating environments. Instead, we provide a general outline for setting up and running a simulation. The following describes the approximate steps you will usually want to take:

- 1st. Describe the features and objects in the world
- 2nd. Define the actions and motivations (goals) that will dictate how the agents interact with the world
- 3rd. Initialize the agents’ internal functions so that they can make action decisions based on how they perceive the world
- 4th. Provide mechanisms to enable the agents to interact with the world

Keep this guideline in mind when you are developing your simulating environment. Maintaining this approximate structure will help reduce the amount of time you might need to spend “debugging” your simulation once it is written.

At this point, we will conclude our introduction by talking about an important concept. That is, the distinction between two key aspects of the Clarion Library: “descriptive” and “functional” objects.

Descriptive vs. Functional Objects

Let’s begin by talking about descriptive objects. Descriptive objects are objects that are used to describe the features of things (e.g., the simulating environment, agents, actions, goals, declarative knowledge, etc.). In other words, they are interested in how the simulating environment looks. They include things like:

- Dimension-value pairs
- Agents
- Chunks (actions, goals, etc.)

In essence, a descriptive object does exactly what it suggests: it describes things. Within the Clarion Library, descriptive objects are generated, stored, and retrieved exclusively through the `World` singleton object. The `World` is a so-called “[singleton](#)” object created for you by the Clarion library. It already exists by the time your code starts running, so it is always available for you to access. Also, it is important to know that all of your interactions with the `World` object are done statically. In other words, making calls to the methods of the “world” is done directly through the class name. For example, suppose you want to specify the dimension-value pair {`Dim1`, `Val1`} as a feature of the world. The following line of code demonstrates how this would be accomplished:

```
DimensionValuePair dv1 = World.NewDimensionValuePair("Dim1", "Val1");
```

Thinking about it conceptually, the `World` essentially contains “everything”:

- The entire environment of your task
- The agents that exist within the environment
- Any special “internal” information that pertains to those agents⁴

There are plenty of more details regarding descriptive objects, but we will get into those other considerations in later tutorials. However, you should at least grasp the concept by this point, so we’ll move over to the functional considerations of the library.

A functional object is an object within the Clarion Library that actually does something. In other words, functional objects are used to describe the processes and mechanism that make up the inter-workings of an agent. In general, we refer to functional objects as “components” or “modules.” These components and modules include the following:

- Implicit Decision Networks
- Action Rules
- Drives
- Meta-cognitive Modules
- Etc.

Notice that the above names refer less to the actual classes that make up the various functional objects provided by the Clarion Library and instead represent the location (or container) in which the functional components are stored within the agent. These “internal locations” are referred to as the agent’s [InternalContainers](#)⁵ within the library. We will get into the specifics of this a bit

⁴ The concept of “agent-specific internal information” is beyond the scope of this guide. Feel free to consult the “*Intermediate Tutorials*” for more details concerning this topic.

⁵ Defined by an enumerator of the same name located within the `Agent` class

more in the later tutorials.⁶ For now, just be aware that the functional objects are generated using the static “initialize” methods that located in the [AgentInitializer](#) class.

By this point, you now have the necessary foundation to get started with building simulations using the Clarion Library. In the next tutorial, we will walk you through a simple example of how to setup a simulating environment, initialize an agent, and have that agent perform a task within the environment. This walk through can be found in the “*Setting Up & Using the ACS*” tutorial, which is located in the “*Basic Tutorials*” section of the “*Tutorials*” folder.

Also, 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.

⁶ Specifically, see the “*Useful Features*” guide located in the “*Features & Plugins*” section of the “*Tutorials*” folders

Terms of Use

Licensing Terms & Conditions

1. **Preamble:** By making use of any copyrighted material within the Clarion Library package (including, but not limited to: the Clarion Library assembly, sample source code, documentation, tutorials, etc.), you (the user of the Clarion Library, including any private person, registered company, partnership, etc.) hereby agree to abide by the terms and conditions contained herein. The terms of use specified by this document govern the relationship between you (hereinafter: the Licensee) and Nicholas Wilson and/or any of his subsidiaries (hereinafter: the Licensor). This agreement (hereinafter: the License) sets the terms, rights, restrictions and obligations regarding the access to and use of the Clarion Library (hereinafter: the Software), which has been created and is owned by the Licensor, by the Licensee.
2. **License Grant:** The Licensor hereby grants the Licensee a Personal, Non-assignable, Non-transferable, Non-commercial (without prior authorization), Non-exclusive license in accordance with the terms set forth (in addition to any other legal restrictions set forth by any 3rd party software[s] that are used by this Software).
 1. **Limited:** The Licensee may use the Software for the purpose of:
 1. Running the Software on the Licensee's Personal (and/or Workstation) Computer[s], Website[s], and/or Server[s];
 2. Allowing 3rd Parties to run the Software on the Licensee's Website[s] and/or Server[s];
 3. Publishing the Software's output to the Licensee and 3rd Parties;
 4. Distributing Verbatim Copies of the Software's output;
 5. Customizing certain, approved (as specified by the documentation and/or guides/tutorials, or otherwise permitted by the Licensor), aspects of the Software to suit the Licensee's needs and/or specifications;
 6. Distributing derived works, customizations, etc. that make use of or integrate with the Software, subject to additional terms.
 2. **Personal:** The licensee may not sublicense, lease, rent or otherwise allow 3rd parties to use the Software, or any portions thereof, apart from executing it in any form and/or apart from running it on the Licensee's Personal (and/or Workstation) Computer[s], Website[s], and/or Server[s].
 3. **Non-Assignable & Non-Transferable:** The Licensee may not assign or otherwise transfer his/her/its rights and duties under this License.
 4. **Non-Commercial:** The Licensee may not use the Software for any commercial purposes without first receiving express authorization from the Licensor. For the purpose of this License, commercial purposes

constitute any access and/or use of the Software in conjunction with a derivative work, customizations, etc. whose intention is (in any way) related to a monetary or financial gain associated with the access to and/or use of said works, customizations, etc. (in any form). This includes the sale, leasing, rental or any other types of monetary transactions associated with the access and/or use of any website[s], application[s], script[s], code, etc. that in any way accesses, runs or otherwise makes use of any part of the Software. The Licensee or any other 3rd party must obtain express written permission by the Licensor to use the Software for any of the aforementioned commercial purposes. Furthermore, the Licensee understands that he/she/it may be required to adhere to additional terms of use, royalties, payments, or other forms of compensation, as stipulated by the Licensor, before being granted a commercial license to access and/or use the Software.

5. **Attribution Requirements:** Any works, customization, etc. that leverage and/or use any aspects of the Software must make reference to its use and provide a citation and/or other form of credit to the Licensor as part of any documentation, publications, etc. associated with those works, customizations, etc. Furthermore, the aforementioned documents must expressly specify where, in what fashion, and to what extent the Software was used, customized, etc.
6. **[Multi-]Site:** The Licensee may use the Software on unlimited Personal (and/or Workstation) Computers, Servers, and/or Websites. However, the Licensee is afforded this right for his/her/its Computers, Servers, and/or Websites only.
7. **With Support & Maintenance:** To the extent permitted under the law, the Software is provided under an AS-IS basis. The Licensor agrees to provide some limited support upon request via email (clarion.support@gmail.com) or through any other mediums (as stipulated by the Software's website, tutorials/guides, documentation, etc.). However, the Licensee acknowledges that this offer of support is not guaranteed under the License and the Licensor can not be held accountable for any such support, or lack thereof. Additionally, the Licensor reserves the right to choose the nature and timeline for responses to any support-related inquiries made by the Licensee.
8. **Trademarks:** The Licensor shall retain full title in Trademarks related to the Software, and any trademarks or trade names contained therein, including the Software's names, logos, and any/all other intellectual property related to the Software. Unless specifically stated in this License, no license shall be made to use, associate or otherwise affiliate the Software with the Licensee in any manner. The Licensee may not use the Software's name, trade name, trademarks, or logo when distributing derivative works, customizations, etc. to 3rd parties without obtaining the express written consent by the Licensor.
9. **Intellectual Property Rights, & Non-Disclosure:** The Licensee hereby acknowledges that the Software is being provided as a closed source

product and contains the Licensor's trade secrets and other proprietary information that has not been disclosed to the public domain. The intellectual property within the Software is the sole possession of the Licensor and the Licensee shall not disclose, reveal, make available or convey to any 3rd party any portions of said intellectual property, including the Software's know-how, means of operations, algorithms, and/or any other proprietary information (hereinafter: Confidential Information). The Licensee hereby agrees to refrain from any attempts to "reverse engineer" or otherwise attain information about any undisclosed aspect of the Software or any related Confidential Information. The Licensee's obligations in this regard shall remain in effect as long as the Confidential Information does not (i) enter into the public domain by any voluntary act made by the Licensor; or (ii) The Licensee is ordered by a definite court order, or any other legal authority, to disclose the Confidential Information; or (iii) The Confidential Information was independently developed by a 3rd Party who was not exposed to the Software. Additionally, the Licensee shall require any 3rd party who sublicenses any works, customizations, etc. that access or make use of any part of the Software to sign a non-disclosure agreement no less restrictive than this License.

3. **Termination:** The terms of this License shall be until terminated. The Licensor may terminate this agreement, including the Licensee's License in the case where the Licensee:
 1. became insolvent or otherwise entered into any liquidation process; or
 2. exported the Software to any jurisdiction where the Licensor may not enforce his rights under the License; or
 3. was in breach of any of this License's terms and conditions and such breach was not resolved immediately upon notification; or
 4. was in breach of any of the terms of clause 2 to this License; or
 5. in any way attempted to "reverse engineer", disclose, or otherwise obtain and/or distribute any Confidential Information regarding the Software; or
 6. otherwise entered into any arrangement which caused the Licensor to be unable to enforce his/its rights under this License.
4. **Compensation:** In consideration of the License granted under clause 2 and conditioned under clause 3, the Licensor reserves the right to seek compensation or payment from the Licensee, as deemed adequate and/or necessary by the Licensor. Failure to adhere to the terms set forth by this License and/or to provide the appropriate compensation or payment (should one be required) shall be construed as a material breach of this License and may be liable to legal actions by the Licensor.
5. **Fair Use:** Any access to and/or use of the Software which falls under the United States Copyright Law with regard to Fair Use (see title 17, section 107 of U.S. code) is explicitly exempted from any compensatory requirements stipulated by this License and are under no threat of legal action by the Licensor. This right is also extended to any uses of the Software that fall under an equivalent law (or

internationally recognized charter), as stipulated by another U.N. certified nation besides the United States, regarding the fair use of copyrighted materials.

6. **Upgrades, Updates and Fixes:** The Licensor may provide the Licensee, from time to time, with Upgrades, Updates or Fixes, as detailed herein and according to his/its sole discretion. The Licensee hereby warrants to keep the Software up-to-date and install all relevant updates and fixes. The Licensee acknowledges that he/she/it may be required to purchase certain upgrades, according to any rates set by the Licensor. However, the Licensor agrees to provide updates and/or fixes free of charge; although, nothing in this License shall require the Licensor to provide Upgrades, Updates and/or Fixes.
 1. **Upgrades:** For the purpose of this License, an Upgrade shall be a material amendment in the Software that contains new features and or major performance improvements and shall be marked as a new version number. For example, should the Licensee obtain the Software under version 6.X.X, the subsequent upgrade shall commence at version 7.X.X.
 2. **Updates:** For the purpose of this License, an Update shall be a minor amendment in the Software, which may contain new features or minor improvements and shall be marked as a new sub-version number. For example, should the Licensee obtain the Software under version 6.1.X, the subsequent update shall commence at version 6.2.X.
 3. **Fix:** For the purpose of this License, a Fix shall be a minor amendment in the Software, intended to address bugs or alter minor features which impair the Software's functionality. A fix shall be marked as a new sub-sub-version number. For example, should the Licensee obtain the Software under version 6.1.1, the subsequent fix shall commence at version 6.1.2.
 4. **Licensing Updates:** All versions of the Software (past and present) are subject to the most current version of this License and only to that version. The Licensor agrees to clearly and explicitly enumerate any and all changes to the License as part of any Upgrades, Updates, and/or Fixes to the Software. In addition, the Licensee reserves the right to refuse any of these changes by simply discontinuing the use of the Software.
7. **Support:** The Software is provided with limited support, as detailed in clause 2.7 of this License. The Licensor has agreed to offer support via email (clarion.support@gmail.com) or through some other form of issue tracking (as stipulated by the Software's website, tutorials/guides, documentation, etc.).
 1. **Bug Notification:** The Licensee may provide the Licensor with details regarding any bug, defect or failure in the Software and should do so promptly and with no delay from such event. The Licensee must comply with any request made by the Licensor for information regarding said bugs, defects or failures and shall furnish him/it with any details that may be needed in order to reproduce such bugs, defects or failures.
 2. **Feature Request:** The Licensee may request additional features be added to the Software, provided, however, that (i) the Licensee waives any claim or right over such feature should it be developed by the Licensor; (ii) the Licensee shall be prohibited from disclosing such

feature request, or feature, to any 3rd party directly competing with the Licensor or any 3rd party which may be, following the development of such feature, in direct competition with the Licensor; (iii) the Licensee warrants that the feature does not infringe any 3rd party patent, trademark, trade-secret or any other intellectual property right; and (iv) the Licensee developed, envisioned and/or created the feature solely by himself, herself, or itself.

8. **Liability:** To the extent permitted under the law, the Software is provided under an AS-IS basis. The Licensor shall never, and without any limit, be liable for any damage, cost, expense or any other payment incurred by the Licensee as a result of the Software's actions, failures, bugs and/or any other interaction between the Software and the Licensee's computer[s], server[s], website[s], derived work[s], customization[s], other software, or any 3rd party computer[s] or service[s]. Moreover, the Licensor shall never be liable for any defect in source code written by the Licensee when relying on the Software or making use of any portion of the Software's features, plugins, samples, etc.

9. **Warranty:**

1. **Intellectual Property:** The Licensor hereby warrants that, to the best of his/its knowledge, the Software does not violate or infringe on any 3rd party claims with regards to intellectual property, patents, and/or trademarks and that no legal action has been taken against him/it for any infringement or violation of any 3rd party intellectual property rights.
2. **No-Warranty:** The Software is provided without any warranty and the Licensor hereby disclaims any warranty that the Software shall be error free, without defects or code which may cause damage to the Licensee (as enumerated in clause 8), or that the Software shall be guaranteed to be functional. The Licensee shall be solely liable to any damage, defect or loss incurred as a result of operating the Software and shall undertake the risks contained in running the Software on the Licensee's Personal (and/or Workstation) Computer[s], Server[s] and/or Website[s].
3. **Prior Inspection:** The Licensor hereby states that he/it has inspected the Software thoroughly and has found it to be satisfactory and adequate (to the best of his/its knowledge). Furthermore, the Licensor states that it does not interfere with regular operation and that it does meet the standards and scope of those systems and architectures under which it was tested. The Licensor contends that he/it found that the Software interacts appropriately with his/its development and test environment[s] and that it does not infringe on any End User Licensing Agreements for any software being used to enable its features, capabilities, and/or services. Given these attempts by the Licensor to verify the safety and stability of the Software, the Licensee hereby waives any claims regarding the Software's incompatibility, performance, results, and/or features, and warrants that he/she/it has inspected the Software to the best of his/her/its ability before using it.

10. **No Refunds:** The Licensee warrants that he/she/it has inspected the Software according to clause 9.3 and that it is adequate to his/her/its needs. Accordingly,

as the Software is an intangible good, the Licensee shall not, ever, be entitled to any refund, rebate, compensation or restitution for any reason whatsoever, even if it is determined that the Software contains material flaws.

11. **Indemnification:** The Licensee hereby warrants to hold the Licensor harmless and indemnify him/it for any lawsuit brought against it with regards to the Licensee's use of the Software, especially in means that violate, breach or otherwise circumvent this License, the Licensor's intellectual property rights or the Licensor's title with regards to the Software. Conversely, the Licensor agrees to promptly notify the Licensee in case of any legal action against the Licensor that may relate to the Licensee and request the Licensee's consent prior to any settlement in relation to such lawsuit or claim.
12. **Governing Law, Jurisdiction:** The Licensee hereby agrees not to initiate any class-action lawsuits against the Licensor in relation to this License and to compensate the Licensor for any court, attorney, or other legal fees should any claim, brought by the Licensee against the Licensor, be denied either in part or in full.

Tutorial Table of Contents

© 2013. Nicholas Wilson

A Brief Note:

This document provides a general outline for the tutorials that are provided with the Clarion Library. While it is by no means exact (or necessarily complete for that matter), the order in which these tutorials are presented herein represent an approximate ordering in which they should be viewed. As a general rule of thumb, by following our recommended order, the material presented within the tutorials should gradually move from easy to advanced.

Setting Up & Using the ACS (in Basics Tutorials)

Walkthrough of the Simple Hello World Task

- The using Clause**
- Declaring the SimpleHelloWorld Class**
- The Main Method**
- Initializing the World**
- Agent Initialization**
- Tweaking Parameters**

Running a Simulation (& Continued Walkthrough)

- Initializing the Sensory Information**
- Perceiving and Acting**
- Processing Outcomes and Delivering Feedback**
- Killing an Agent**

Intermediate ACS Setup (in Intermediate Tutorials)

Optimizing Task Performance via “Tuning” Parameters

- Making Global Parameter Changes**
- Making Local Parameter Changes**

Setting Up the Working Memory

- Manually Setting a Chunk in Working Memory**
- Using Action Chunks**

Setting Up & Using the Goal Structure (in Basics Tutorials)

Setting Up the Goal Structure
Manually Setting a Goal
Using Action Chunks

Intermediate MS and MCS Setup (in Intermediate Tutorials)

Setting Up and Using Drives and Meta-Cognitive Modules
Initializing a Drive
The Drive Equation
Stimulating a Drive
Accessing Agent Meta Info
Correlating Drives and Meta-Cognitive Modules
Meta-Cognitive Module Integration

Basic Customization (in Customizations)

Customized Methods (Using Delegates)
Specifying Delegates as Parameters during Initialization
Creating Custom Rules
Using the SupportCalculator Delegate to Set Up an IRL Rule
Initializing the IRL Rule
Using the SupportCalculator Delegate to Set Up a Fixed Rule
A Note on the Generically Typed
DimensionValuePair<DType, VType> Class
Initializing the Fixed Rule
Generic Equations

Useful Features (in Features & Plugins)

Viewing an Agent's "Internals"

Logging (using Trace)

The Implicit Component Initializer

Pre-Training

Auto-Encoding

Distributed Dimension-Value Pairs

Populating the Input and Output Layers of an Implicit Component

Timing

Response Time

"Real-time" Mode

Asynchronous Operation

Setting Up & Using the NACS (in Advanced Tutorials)

Setting Up & Performing Reasoning

A Walk-through of the "Simple Reasoner" Task

Distributed Dimension-Value Pairs

Adding Knowledge to the GKS

Initializing Associative Memory Networks

Initializing Associative Rules

Performing Reasoning

Setting Up & Using Episodic Memory

Creating Episodes

Initializing Associative Episodic Memory Networks

Generating New Knowledge and Associative Rules

Performing "Offline" Learning

Advanced ACS Setup (in Advanced Tutorials)

Interacting with the NACS

- Making Reasoning Requests
- Specifying Alternative Dimensions
- Filtering Input/Conclusions
- Retrieving Chunks from the GKS
- Interacting with Episodic Memory
 - Retrieving Episodes*
 - “Offline” Learning*

Generative Actions

- An Example: Using Generative Actions to Change Local Parameters

Using Plugins (in Features & Plugins)

The Serialization Plugin

- Serializing (or Saving) Various Aspects of a Simulating Environment
- De-serializing (or Loading) Various Aspects of a Simulating Environment

Interacting with Front-Ends

- Remote Simulating Environments
 - Communicating via XML*
 - Communicating via JSON*
- The Keyboard and Mouse Plugins
 - Using the “Built-In” Plugin Actions*

Advanced Customization (in Customizations)

Getting Started

- ACS Structure
- NACS Structure
- MS Structure
- MCS Structure
- Interfaces and Templates

How to Implement a Custom Component

- Requirements for Implementing a Custom Component
- Implementing a “Factory”
- Implementing a “Parameters” class

 - Local (per instance) Parameters*

 - Global (**static**) Parameters*

 - Factor # 1

 - Factor # 2

 - Factor # 3

 - Factor # 4

 - Factor # 5

 - Factor # 6

 - Factor # 7

Committing and Retracting

 - Using the InitializeOnCommit Property*

How to Implement a Custom (Secondary) Drive

- Implementing the Nested “Factory” Class
- Implementing the Nested “Parameters” Class

Serializing a Custom Component (or Drive)

- Specifying the System.Runtime.Serialization Resource*

- The DataContract Attribute*

- The DataMember Attribute*

- Pre/Post Serialization and Deserialization Attributes*

- Serializing the Global (**static**) Parameters*

Setting Up & Using the ACS

© 2013. Nicholas Wilson

Table of Contents

Walkthrough of the Simple Hello World Task	1
The using Clause	2
Declaring the SimpleHelloWorld Class.....	2
The Main Method.....	2
Initializing the World	4
Agent Initialization	5
Tweaking Parameters.....	7
Running a Simulation (& Continued Walkthrough)	8
Initializing the Sensory Information	9
Perceiving and Acting	10
Processing Outcomes and Delivering Feedback.....	10
Killing an Agent.....	12

Walkthrough of the Simple Hello World Task

In this section, we will go line-by-line through one of the simplest tasks to simulate: *Hello World*. This simple task will provide you with the basics for getting up and running building simulations in Clarion. It should not, however, be considered to be the definitive method for developing simulating environments (or constructing agents for that matter). We encourage you to look at all of the samples and consult the various guides and documentation when building your own simulations. Once you get comfortable developing simulations, we encourage you to explore the capabilities of the Clarion Library and use your own creativity when constructing your models.

To fully convey the point that simulations need not follow any rigid format, the **SimpleHelloWorld** simulation is written entirely within a single method (i.e., the “Main” method). The basic layout for this task is as follows:

- It uses **ONLY** the ACS with a reinforcement trainable backpropagation network in the bottom level and RER (Rule Extraction and Refinement) turned on in order to extract rules based on the reinforcement.
- The goal is for the agent to learn the following:

If someone says “hello” or “goodbye” to me, then I should respond with either “hello” or “goodbye” respectively.

- At the end of the task, the above statement should be represented as rules in the top level of the ACS (having been learned via RER).
- Other than knowing the inputs and outputs, the agent should have no *a-priori* knowledge of the task dynamics.

So let's look now at how we would go about setting up this simple task. The first thing we need to do is add the "Simple Hello World" sample to our project. This is generally accomplished by simply right-clicking on your project (i.e., the one we setup in the "Getting Started Guide") and choosing the "Add Files" option (under the "Add" menu item). The file name for this task is "HelloWorld-Simple.cs" and it can be found in the "Simple" folder in the "Samples" section of the Clarion Library package. Once you have added the file, it should open in the main window of your development environment. If it does not, you should be able to open it by double-clicking on it from the Solution Explorer.

The using Clause

In the previous ("Getting Started") tutorial, we discussed how to add the Clarion Library as a resource (or reference) for a simulation project. However, if you are familiar with Java (or maybe Python), then you will already be aware that a resource library needs to also be signaled within a class file by specifying an inclusion keyword at the top of a code file (for example, with the `import` statement in Java). C# uses the keyword `using`. The code below lists the basic resources we use for our example:

```
using Clarion;  
using Clarion.Framework;
```

The `Clarion` and `Clarion.Framework` namespaces are the primary locations for the majority of the classes you will need when setting up and using a Clarion-based agent in your simulation. More advanced examples might use additional libraries.

Declaring the SimpleHelloWorld Class

Now that we have signaled the necessary namespaces, we need to specify the initial declaration for the class that will perform the task:

```
public class SimpleHelloWorld
```

This class contains all of the code for my simulating environment, including the initialization of the world, my agent, and the mechanisms that allow the agent to interact with the world. Since this is a simple task, we will actually perform all of these steps within a single (`Main`) method.

The Main Method

If you are familiar with Java, C, or C++, you know that a method called `Main` is where any program begins its execution. In C#, this routine is always spelled with a capital letter, and must be declared as `static void` (or `static int`, if you want to

return an exit value to the operating system). As is usual for Java, C, or C++, command-line arguments are handled with the string array, `args`, passed to the `Main` method. While this argument is optional, including it is usually the default behavior in most development environments. In addition, you may find it convenient to use in some of your simulations.

```
static void Main(string[] args)
{
    ...
}
```

We should also note that it is usually a good idea to limit the amount of work that is performed in the `Main` method. In practice, this method should initialize its enclosing class, writing out a couple of progress messages, and call a few methods to initiate the task. The major work of the simulation should be performed in other methods such that your `Main` method is easy to follow and very high-level. This being said, the Simple Hello World example does not follow this convention. Remember, for this particular simulation, we simply wish to demonstrate the simplicity in which simulations can be created. The Clarion Library will still perform a task correctly even if you use poor programming practices.

The first lines you will see in the `Main` method are these:

```
//Initialize the task
Console.WriteLine("Initializing the Simple Hello World Task");

int CorrectCounter = 0;
int NumberTrials = 10000;
int progress = 0;

World.LoggingLevel = TraceLevel.Warning;

TextWriter orig = Console.Out;
StreamWriter sw = File.CreateText("HelloWorldSimple.txt");
```

This code first sets up some task-specific variables to use for tracking the results of the task. It also configures C# to output the results to a text file (instead of to the console window). You do not need to implement any of this within your simulations. We have merely implemented it this way here so that the console will look nice and clean when you actually run the simulation.

Notice also that we specify a value for `LoggingLevel`. The Clarion Library lets you trace the inner workings of your agents at varying levels of detail. By default, if logging is turned on, the logging information gets output to "*Clarion Library.log*". However, if you wish to change this location (or if you want additional details on how logging works within the Clarion Library) see the "*Useful Features*" guide¹.

¹ Located in the "*Features & Plugins*" section of the "*Tutorials*" folder

Initializing the World

Some amount of initialization of the world itself is needed before we can run our task. The next lines of code in our demonstration are these:

```
DimensionValuePair hi = World.NewDimensionValuePair("Salutation", "Hello");  
DimensionValuePair bye = World.NewDimensionValuePair("Salutation", "Goodbye");
```

When building a simulation, the first thing we need to do is describe what the simulating environment looks like. For example, you will usually want to tell the `World` about the existence of any dimension-value pairs we need for our task. For our Simple Hello World task, there is one dimension named `"Salutation"`, and it is given a value of either `"Hello"` or `"Goodbye."` These DV pairs effectively represent the simulating environment for this task.

Note that we can refer to these objects as often as we need to as we continue setting up the simulation by either maintaining links (i.e., pointers) to them within our simulation, or by calling the `"Get"` methods that are provided in the `World` singleton object. Later, you will see that we can use our descriptive objects to create input nodes to the agent's implicit decision network. However, for now, simply note that these dimension-value pairs can now be considered to exist within the world.

We now need to define the external actions that the agent will be able to perform (in this case `"Hello"` and `"Goodbye"`) when presented with salutation inputs. The `NewExternalActionChunk` method accomplishes this:

```
ExternalActionChunk sayHi = World.NewExternalActionChunk("Hello");  
ExternalActionChunk sayBye = World.NewExternalActionChunk("Goodbye");
```

Here, our action chunks are about the simplest possible: indicating either responding `"Hello"` or `"Goodbye"`, which is all we need for this task. Note, however, that in practice, action chunks can be made as complex as required (since they are in fact chunks²).

With these essential initializations out of the way, the simulating environment has essentially been set up. There are, of course, more complex aspects that can potentially be set up here, but we will leave those alone for now and address them in the other, more advanced, guides. At this point, it is time to initialize our agent (whom we will call John):

```
//Initialize the Agent  
Agent John = World.NewAgent("John");
```

John is initialized within the world just like any other descriptive object, because technically John is just another object in the world. This is accomplished by calling the `NewAgent` method. Note that we also give the agent its name at this point, which is the string parameter `"John"` passed to `NewAgent`. Also note that this name (or

² Chunks are a well-established representational concept that contain arbitrarily large collections of dimension-value pairs

“label” when we are creating goal/action/declarative chunks) is optional and is only needed if you later want to “retrieve”, or `Get`, the agent (or chunk) back from the `World`.

To this end, keep in mind that since the names/labels are optional, they can be assigned to multiple agents/chunks. In other words, it is completely possible to have more than one agent in the world named John. The system will not break if you want to give your (non dimension-value pair) world objects the same name/label. However, if you choose to do this, it will limit your ability to retrieve the objects from the `World` using the `Get... (by_name_or_label)` methods. In particular, when multiple world objects (of the same object type) have the same name/label, the `Get` methods cannot guarantee that the object that is retrieved will be the object you intended.

With that in mind, if you plan to build a simulating environment with multiple agents, it is very important to make sure you segregate your agent initialization from your descriptive object declaration. Otherwise, you could inadvertently create multiple “copies” of the same object and potentially cause major problems with the operation of your task (if you also use the `World.Get...` methods).

Moving back to our example, we have now created an “empty” agent. We say it is empty, because it does not know anything about those DV pairs and action chunks that were set up earlier, nor does it contain any functional mechanisms with which to interact with the world. Therefore, we need to finish setting up our agent, John.

Agent Initialization

In this task, the only functional object we need to initialize ahead of time is an Implicit Decision Network (IDN) in the bottom level of the ACS. To set up this network, we will initialize a backpropagation network. More specifically we use a simplified Q-learning backpropagation network. This line demonstrates how to accomplish this initialization:

```
SimplifiedQBPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork  
    (John, SimplifiedQBPNetwork.Factory);
```

The network type that we use for the implicit decision network in this simulation is a `SimplifiedQBPNetwork`, one of several artificial neural network components located in the `Clarion.Framework` namespace of the Clarion Library. We generate this network (and associate it with agent John) through the `AgentInitializer` (located, as with the `World` class, within the root `Clarion` namespace).

Note again that the `AgentInitializer` is the primary means by which we attach the myriad of internal functional objects (e.g., implicit components, drives, meta-cognitive modules, etc.) to a Clarion agent. Along with the `World` class, the agent initializer represents one of the primary foundational objects for interacting with a Clarion agent within a simulated task environment.

As we mentioned earlier, for the simple Hello World task, all we need is an implicit decision network (or IDN) in the bottom-level of the ACS, so we call the

`InitializeImplicitDecisionNetwork` method to generate the neural network. To initialize the IDN, we pass two items to the initializer method in the `AgentInitializer`:

- The agent (John, in this case) to which the network is to be attached
- A factory that will be used to generate the network we want

Note that this method will place the network within John in an “initializing” state (more on this in a moment). Since a complex agent could literally have dozens or even hundreds of special-purpose networks, we have implemented a standard [factory pattern](#), which allows for you to specify any type of internal object you desire (including customized ones of your own creation³). Here, we want to use a `SimplifiedQBPNetwork` so we pass in the factory that the system will use to generate this network.

For all of the internal functional objects in the Clarion Library (as specified by the Clarion theory), a factory has already been provided for you and can be accessed by statically invoking the `Factory` [property](#). In our current example, specifying `SimplifiedQBPNetwork`.`Factory` will provide the factory for creating the `SimplifiedQBPNetwork` in the bottom level of the ACS. Note here that we can also pass along any number of parameters to the factory. In this particular example, we do not need to do so, however, you will see other examples as you go along with these tutorials where such parameters are required. For the “built-in” functional objects, lists for both the required and optional parameters can be found in the documentation.

The `InitializeImplicitDecisionNetwork` returns an implicit component (of the type that is specified by the factory). For our example, we are initializing a `SimplifiedQBPNetwork`, and have assigned it to the `net` variable. This `net` can be considered as being part of our agent, John. In other words, the network has not only been generated by calling the initialize method, but it has also been attached to the agent that was specified when the method was called. This means that the network belongs to the agent and cannot be used by any other agent.

In this simple example, we are creating only one neural network component (as an IDN in the bottom-level of the ACS). In more elaborate simulations, you will, no doubt, call several other methods in `AgentInitializer` many times in order to generate all of the components you will need for constructing your agents. For details on initializing functional objects (within the other “agent internals containers”) consult the documentation for the various “Initialize” methods in the `AgentInitializer`. Additionally, further information on initializing certain other components can also be found in the later, more advanced, guides.

At this point, however, we need to finish initializing our IDN, `net`, using the DV pairs and action chunks that we defined earlier. This will give our agent, John, the ability

³ See the “*Advanced Customization*” tutorial (located in the “*Customizations*” section of the “*Tutorials*” folder) for details on how to implement a custom component

to choose actions based upon the sensory information it receives from the world. The four lines below accomplish this:

```
net.Input.Add(hi);  
net.Input.Add(bye);  
  
net.Output.Add(sayHi);  
net.Output.Add(sayBye);
```

First, we begin by specifying two input nodes for the two DV pairs that represent the salutations. To accomplish this, we call the `Input.Add` method. Similarly, we will also add two output nodes to represent the two response actions by calling `Output.Add`. Note that the number of nodes in the hidden layer is computed automatically, which is fine for this example. In more advanced simulations, however, you can take control of this number, but that is outside the scope of this guide (see the documentation for more details).

As part of the initialization phase for any simulation, your job will be to create and set up all of the functional objects you wish for your agent(s) to use. During initialization these objects are open and available for you to configure using your initialization code. Once initialized, the objects will exist in a special, temporary, initialization area within the agent until you are finished setting them up. When you are finished, you will need to signal to the agent that it can begin using the functional object. When the agent is notified of this fact, it will remove the object from the temporary location and wire it into the appropriate container (e.g., the bottom level of the ACS in our current example). At that point, in most cases, you will **NOT** be able to make additional changes to that functional object (except to “tune” its parameters). In the Clarion Library, this concept is called “committing.” All functional objects in the Clarion Library **MUST** be “committed” to an agent after they have been initialized. Committing a functional object makes it “[immutable](#)” (i.e., “locked” or read only). You should not attempt to make changes after you make this call.

At this point we have finished initializing `net`; however, we now need to tell John that it can begin using the network. To signal to our agent that we have completed the initialization of `net` we make the `Commit` call:

```
John.Commit(net);
```

Tweaking Parameters

The final initialization that is usually performed is parameter tweaking. The Clarion technical specification (which can be found on Ron Sun’s [website](#)) contains a list of default settings for the various mechanisms described by the Clarion theory. However, you will probably find that, in many cases, these settings do not provide you with the optimal results for your task. In such cases, you will likely want to tune some of the parameters that are introduced in this tutorial.

We will discuss how to tweak parameters in the next guide (i.e. the “*Intermediate ACS Setup*” tutorial located in the “*Intermediate Tutorials*” section of the “*Tutorials*” folder). However, for the Simple Hello World task, the following parameter settings are used to optimize the agent’s performance:

```
net.Parameters.LEARNING_RATE = 1;  
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

Running a Simulation (& Continued Walkthrough)

In addition to initializing the world and agents, it is also the job of a simulating environment to act as the intermediary between the agent and the world during the actual running of a task.

In general, the simulating environment needs to handle the following communications:

1. Specifying to an agent the sensory information it perceives on a given perception-action cycle
2. Capturing and recording the action that is chosen by an agent
3. Updating the state of the world (if necessary) based on an agent’s actions
4. Providing feedback to an agent as to the “goodness” or “badness” of its actions
5. Tracking the performance of the agent (for the sake of reporting results at the end of the task)

Now that the world and agent have been set up, it is time to give John a means for interacting with the world. This is where the bulk of our simulation code actually comes into play.

```
//Run the task  
Console.WriteLine("Running the Simple Hello World Task");  
Console.SetOut(sw);  
  
Random rand = new Random();  
SensoryInformation si;  
  
ExternalActionChunk chosen;
```

The above lines mainly set up some variables that will be useful as part of running the simulation: a random number generator (`rand`) for choosing the configuration of the sensory information, a sensory information pointer to hold onto the sensory information for the current perception-action cycle, and a variable named `chosen` to capture the action chosen by the agent.

The next line begins the major body of the task:

```

for (int i = 0; i < NumberTrials; i++)
{
    ...
}

```

The `for` loop (above) is in charge of the “flow” for the task. Inside this loop, the sensory information is set up for each perception-action cycle (i.e., each trial), the chosen action is captured, feedback is given, and the agent’s performance is recorded.

Initializing the Sensory Information

The first thing we must do on any given trial is get a sensory information object. The code below handles this request:

```

si = World.NewSensoryInformation(John);

```

We obtain sensory information objects from the world by calling the `NewSensoryInformation` method and specifying the agent for whom the sensory information is intended. Note that, while it isn’t particularly pertinent for the present task, sensory information objects **cannot** be shared between agents. This is necessary because the sensory information object not only tracks the state of the world (as seen by that agent), but it also tracks the “internal meta information” of an agent (i.e., the state of its goals, working memory, drives, etc.). If you would like to “share” a certain configuration of the world between two agents, an overload for the `NewSensoryInformation` method exists that will copy the configuration of a sensory information object into a new sensory information object (for the specified agent).

These next lines in the simulation are used to decide on the configuration for a sensory information object:

```

//Randomly choose an input to perceive.
if (rand.NextDouble() < .5)
{
    //Say "Hello"
    si.Add(hi, hi.MAXIMUM_ACTIVATION);
    si.Add(bye, bye.MINIMUM_ACTIVATION);
}
else
{
    //Say "Goodbye"
    si.Add(hi, hi.MINIMUM_ACTIVATION);
    si.Add(bye, bye.MAXIMUM_ACTIVATION);
}

```

For our simple task, we are basically just “flipping a coin” to decide whether John perceives someone saying “hi” or “bye.” More complex simulating environments will likely have more complicated methods for determining the appropriate sensory information. However, for this task, a random choice is sufficient.

We set up the sensory information object by adding descriptive objects to it (along with an activation level for each object). This is done using the `Add` method.⁴ Also, note that the agent’s “internal meta information” is pre-loaded into the sensory information object, so if your simulation needs to set the activation for a part of the agent’s “internal state”, you would do so by “setting” the activation rather than “adding” it. The following is an example of how you could do this:

```
si[...] = 1;
```

Perceiving and Acting

The next lines initiate the agent’s perception of the sensory information and retrieve the agent’s action (when one is chosen):

```
//Perceive the sensory information
John.Perceive(si);

//Choose an action
chosen = John.GetChosenExternalAction(si);
```

The `Perceive` method initiates the process of decision-making and `GetChosenExternalAction` returns the action that is chosen by John (given the current sensory information). Note that both of these methods can only be called **ONCE** for any given sensory information object (or time stamp). This means that an agent **cannot** perceive a sensory information object more than once, so you **must** generate a **new** sensory information object each time you want your agent to perceive something. It also means that your agent will not repeat itself, so make sure that you either apply its action immediately or store its choice somewhere in the simulating environment.

The above example represents the simplest method for initiating a perception-action cycle. There are other, more complicated, ways to interact with an agent (e.g., using asynchronous operations⁵). These other methods are outside of the scope of this guide, so we will leave further discussion on that topic to later, more advanced, guides.

Processing Outcomes and Delivering Feedback

After the chosen action has been captured, the following `if` statement determines the “consequences” of that action, records the outcome, and reward or punishes John accordingly:

⁴ The `+/-` operators can also be used to add or remove items from the sensory information (as well as the inputs/outputs of implicit components). Note, however, that the activation when using these operations will always be set to the minimum activation.

⁵ Details on how to set up asynchronous operation can be found in the “*Useful Features*” tutorial (located in the “*Features & Plugins*” section of the “*Tutorials*” folder)

```

//Deliver appropriate feedback to the agent
if (chosen == sayHi)
{
    //The agent said "Hello".
    if (si[hi] == hi.MAXIMUM_ACTIVATION)
    {
        //The agent was right.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent was wrong.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
else
{
    //The agent said "Goodbye".
    if (si[bye] == bye.MAXIMUM_ACTIVATION)
    {
        //The agent was right.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent was wrong.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
}

```

You should notice two primary things about this segment of code. First, notice that we are able to compare our actions and sensory information objects using the standard “==” comparator. All of the descriptive objects in the Clarion Library have had their operations overloaded so that they act more like [value types](#).⁶ Second, to give John feedback, all we need to do is call the `ReceiveFeedback` method. Calling this method automatically initiates a round of learning inside John. Even if we were

⁶ This is also the case for sensory information objects, and the input/output layers of implicit components.

using a Q-learning network instead of a simplified Q-learning network, we would still only need to call `ReceiveFeedback` and the system will take care of the rest.⁷

Note also that the feedback on this simulation is between 0 and 1. Feel free to use whatever scales you want (although see the “*Intermediate ACS Setup*” for additional considerations regarding this). However, unless it is absolutely necessary, you should try to stick to keeping your activations and feedback between 0 and 1. Using this convention, a feedback of 1 would be tantamount to the highest level of positive feedback possible, and 0 would equate to the highest level of negative feedback (with .5 being more or less neutral feedback).

The remaining lines of code in the simulation basically just update the progress of the simulation and report the results after all of the trials have completed. Therefore, we will not go over these lines. However, once we have finished processing all of our results, there is still one more thing that has to be done before our simulation can be exited.

Killing an Agent

The last step we need to take in order to terminate our simulation is to terminate our agent. For our current example, we kill John by calling the following line of code:

```
John.Die();
```

This command initiates the termination of all of John’s internal processes (in other words, he “dies”). Note that even though the agent’s processes are terminated, its internal configuration is still maintained. This means you can still access/view or “save” (i.e., serialize) the agent’s internal configuration even after the agent dies.⁸ If we didn’t remember to kill John, then the application would not be able to close because John’s internal mechanisms would still be running. However, once John is dead, the simulation will be able to exit by returning from the `Main` method.

At this point, you should have everything you need to start writing your first (simple) simulation. If you get stuck at any point, consult this guide again or take a look at the API resource document (in the “*Documentation*” folder). When you are ready to move on to the more complicated aspects of the ACS, the next tutorial (“*Intermediate ACS Setup*”), can be found in the “*Intermediate Tutorials*” section of the “*Tutorials*” folder.

In addition, you may also want to check out the “*Useful Features*” tutorial.⁹ It contains information about some of the “built-in” enhancements of the Clarion

⁷ In the case that an implicit component expects new input (as is the case for Q-learning), the system will actually wait until the `perceive` method is called on the next sensory information object before performing learning.

⁸ Details on the latter can be found in the “*Using Plugins*” guide (located in the “*Features & Plugins*” section).

⁹ Located in the “*Features & Plugins*” section of the “*Tutorials*” folder.

Library. These enhancements have been designed to aid you in the development of your simulating environments.

Remember, 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.

Intermediate ACS Setup

© 2013. Nicholas Wilson

Table of Contents

Optimizing Task Performance via “Tuning” Parameters	1
Making Global Parameter Changes	2
Making Local Parameter Changes	3
Setting Up the Working Memory	4
Manually Setting a Chunk in Working Memory	5
Using Action Chunks	5

Optimizing Task Performance via “Tuning” Parameters

Frequently, you will find that a task runs reasonably well by simply setting up an agent using all of the default settings. However, there will likely be times where the defaults simply do not provide “optimal” performance and you may want to “tune” the agent’s settings to get it to perform a task more effectively.

The Clarion theory specifies several parameters for various mechanisms (see technical specification document [here](#) for more details). These parameters have been implemented into the Clarion Library in two ways: as global (`static`) parameters, and as local (instance) parameters. The parameters have been stored within “Parameters” classes, which are located throughout the system based upon their most logical position (as specified by the Clarion theory). For example, the `RefineableActionRule` class implements a rule type that is refineable (and is usually extracted via RER). As part of being “refineable”, these rules contain methods for generalization and specialization, each of which have some threshold parameters that can be “tuned” in order to optimize the frequency in which either process (i.e., specialization or generalization) occurs. The following code (from the “*Full Hello World*”¹ simulation sample) demonstrates how two of these thresholds might be changed during the initialization of a task:

```
RefineableActionRule.GlobalParameters.SPECIALIZATION_THRESHOLD_1 = -.6;  
RefineableActionRule.GlobalParameters.GENERALIZATION_THRESHOLD_1 = -.2;
```

In addition to providing all of the default parameters specified by the Clarion theory, the Clarion Library also provides several extra parameters designed to aid you in running a task. For example, suppose we wanted to turn off the various forms of learning that take place in the ACS (i.e., rule refinement, bottom-up learning or rule

¹ Located in the “*Intermediate*” section of the “*Samples*” folder (filename: “*HelloWorld - Full.cs*”).

extraction, top-down learning, and/or learning in the bottom level). The following lines of code could be used to accomplish this:

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;  
ActionCenteredSubsystem.GlobalParameters.PERFORM_RULE_EXTRACTION = false;  
ActionCenteredSubsystem.GlobalParameters.PERFORM_TOP_DOWN_LEARNING = false;  
ActionCenteredSubsystem.GlobalParameters.PERFORM_BL_LEARNING = false;
```

All of these are examples of “global” parameter changes, so let’s begin our discussion on how to make parameter changes by first considering the global (`static`) method for making parameter changes.

Making Global Parameter Changes

The first thing you need to know with regards to global parameter changes is how these parameters are accessed. As mentioned earlier, global parameters are stored statically. This means that they are accessible from a static call (i.e., using the class name) to the class with whom the parameters are associated. For all of the “built-in” classes defined by the Clarion Library, the global parameters can be found within the parameters class that is returned by the `GlobalParameters` property. In our earlier examples, the global parameters associated with specialization and generalization of action rules were accessed via:

```
RefineableActionRule.GlobalParameters
```

And the parameters for turning on and off learning in the ACS were accessed by:

```
ActionCenteredSubsystem.GlobalParameters
```

However, before you begin tuning all of your parameters using the global (`static`) method, it is essential to understand a few important points about how global parameters are implemented and what the consequences are with regards to how and when global parameter changes can be made.

First, global parameters changes are only applicable to an instance of a class **BEFORE** it is initialized. The (`static`) global parameters for any given class are only used during the initialization process in order to set the values of the local parameters of an instance. Once the instance has been initialized, it will thereafter only use the local parameters. In other words, making a global parameter change **AFTER** an instance of a class has been initialized will have **NO** effect on the corresponding local parameter for that instance. For example, let’s look at the following lines of code:

```
Agent John = World.NewAgent("John");  
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;
```

Note that the ACS is initialized as part of the initialization of an agent, so John’s ACS will already be instantiated by the time the global parameter change is made (in the second line). As a result, if it was our intention to turn off refinement in John’s ACS,

our code (from above) would fail. Instead, the local `PERFORM_RER_REFINEMENT` parameter would still be set to `true` and John's ACS would still perform refinement.² The correct way of changing the parameter globally (so as to achieve our intended behavior) would be as follows:

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;  
  
Agent John = World.NewAgent("John");
```

Moving on, the second thing you need to know is that, for all of the “built-in” classes of the Clarion Library, the global parameters have been set up such that they can be changed at any point within the inheritance hierarchy. For example, suppose you wanted to change the `POSITIVE_MATCH_THRESHOLD` parameter for **ALL** rules (regardless of their type). The following line of code would accomplish this:

```
Rule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

This command will change the parameter for any class that derives from `Rule` (e.g., `RefineableActionRule`, `IRLRule`, `AssociativeRule`, etc.). However, suppose you just wanted to change the `POSITIVE_MATCH_THRESHOLD` parameter for IRL rules only. This would be accomplished in essential the same way as before, except you would call it on the `IRLRule` class instead:

```
IRLRule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

Making parameter changes at different points in the inheritance hierarchy provides a convenient way for making wholesale (and possibly even targeted) parameter tweaks. However, you may find that you want to change the parameters for specific instances of a class, or that you want different instances of a class to have slightly different settings (e.g., depending on where it is located within an agent, or based upon the “group” of the agent in which it was initialized). For this reason, the Clarion Library also provides a method for changing parameters locally.

Making Local Parameter Changes

The other method for performing parameter changes within the Clarion Library is to make such changes on a per-instance basis. This is what we refer to as “local” parameter changing. Local parameter changes are made on instances of a class through its `Parameters` property. For example, if we wanted to change the `PERFORM_RER_REFINEMENT` parameter for just John's ACS (from the previous example), then we could accomplish this by doing the following:

```
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

All of the “built-in” classes in the Clarion Library (which contain parameters) have a local `Parameters` property. In addition, unlike the global parameters, local

² Although the local parameter will be changed for any agents that are initialized **AFTER** the global parameter change is made

parameters can be tuned at any point (after the local instance has been initialized, of course). Note that the local parameters are not subject to the “immutable” (that is, read-only) restriction placed on an agent’s internal (functional) objects, so they can be altered even after these objects have been “committed” to the agent. For example, suppose you wanted to change the `LEARNING_RATE` parameter for an instance of a `SimplifiedQBPNetwork` (called `net`) that was initialized within the bottom level of John’s ACS. The following code could be called at any point during the task:

```
//Retrieves the network from the bottom level of John's ACS
SimplifiedQBPNetwork net = (SimplifiedQBPNetwork)John.
    GetInternals(Agent.Internals.IMPLICIT_DECISION_NETWORKS).First();

net.Parameters.LEARNING_RATE = .5;
```

This parameter change will take effect the next time the learning rate is applied (which is presumably the next time John receives feedback). Note also that the first line of code (above) will retrieve the network from the bottom level of John’s ACS, but only if it is either by itself in the bottom level of John’s ACS or it is the first network in the collection that is returned by the `GetInternals` method. We won’t get into the specifics of the `GetInternals` method at this point. Instead, you can consult the “*Useful Features*” tutorial³ for more information about how to use this method.

Finally, we should also mention here that there is an additional way to change parameters in a more “automatic” fashion (i.e., by having either the ACS or a module within the MCS initiate the parameter change). However, using this method is beyond the scope of this tutorial, as it makes use of a concept called “Generative Actions” (which we will discuss in a later tutorial⁴).

At this point, we have covered the two primary techniques for changing parameters. These techniques should suffice any time it is necessary to tune a simulation.

Setting Up the Working Memory

In this section we will discuss how to set up and use the working memory. Broadly speaking, the working memory can be thought of as being a “container” within an agent that holds knowledge about the world (i.e., declarative chunks, previous action chunks, etc.). Technically speaking, it is located within the ACS. However, all interaction with the working memory (from the simulating environment) is performed directly via the `Agent` class. For instance, we can view the contents of working memory by calling the `GetInternals` method. The code below demonstrates how we might accomplish this for our agent, John:

```
IEnumerable<Chunk> wmContents =
    (IEnumerable<Chunk>)John.GetInternals
```

³ Located in the “*Features & Plugins*” section of the “*Tutorials*” folder.

⁴ See the “*Advanced ACS Setup*” tutorial in the “*Advanced Tutorials*” section of the “*Tutorials*” folder.

```
(Agent.InternalWorldObjectContainers.WORKING_MEMORY);
```

Note that the working memory can hold any type of chunk. Additionally, whenever a chunk is “set” in the working memory, it becomes a part of the “internal sensory information” and will automatically be “activated” in the [SensoryInformation](#) the next time one is perceived.

World objects (i.e., chunks) are added to working memory either manually or by using a “working memory update action chunk.” First, let’s look at how chunks can be set manually.

Manually Setting a Chunk in Working Memory

The simplest way to “set” (or add) a chunk in working memory is to do it manually. We do this by calling the `SetWMChunk` method for the agent where the chunk is being set in working memory. The code below demonstrates how we can do this for our agent, John:

```
John.SetWMChunk(ch, 1);
```

To set a chunk in working memory we must specify two things when calling the above method: the chunk to be set and the “activation level” for that chunk. This will “set” (or add) the chunk in working memory. To “deactivate” (or remove) the chunk from working memory we call the `ResetWMChunk` method. The following code demonstrates how we can manually reset (i.e., deactivate or remove) the chunk in working memory:

```
John.ResetWMChunk(ch);
```

These two simple methods provide you with all of the power you need to be able to use chunks within working memory. However, manually setting working memory may not be enough for your simulating environment. Recall that the Clarion theory provides many more details regarding various additional methods for setting chunks in working memory. For example, we can use “working memory actions” in the ACS or in the MCS to perform operations on the working memory itself. In the following section, we will look at how chunks can be set using “working memory actions” in the ACS.

Using Action Chunks

To begin, while the Clarion theory refers to actions that affect the working memory as being “working memory actions”, the implementation uses a clearer term for describing these sorts of actions. In the Clarion Library, actions that perform updates on the working memory are defined using the [WorkingMemoryUpdateActionChunk](#) class. The contents of these action chunks contain information about the sorts of updates that are to be performed. For example, suppose we want an action that “sets” the chunk `ch` in working memory. The following code sets up such an action:

```
WorkingMemoryUpdateActionChunk wmAct = World.NewWorkingMemoryUpdateActionChunk();  
wmAct.Add(WorkingMemory.RecognizedActions.SET, ch);
```

Note that we specify, as the first parameter in our Add method, an enumerator called `RecognizedActions`. Several classes within the Clarion Library (namely those mechanism that can be manipulated using actions, e.g., the `NonActionCeneteredSubsystem`, the `GoalStructure`, `WorkingMemory`, etc.) define a `RecognizedActions` enumerator. This enumerator provides the list of commands that an action can perform on an instance of that class. The `WorkingMemory` recognizes four types of actions:

- `SET`. “Adds” the chunk to working memory
- `RESET`. “Removes” the chunk from working memory
- `RESET_ALL`. “Removes” **ALL** of the chunks from working memory
- `SET_RESET`. Combines the `RESET_ALL` and `SET` actions

If we want a component in the ACS to use this action, all we have to do is specify it in the output layer of the component. Below is an example of how we would set up this action in a network on the bottom level of the ACS.

```
... //Elided code performing additional initialization for the network  
net.Output.Add(wmAct);
```

Now, whenever the ACS selects this `WorkingMemoryUpdateActionChunk`, the system will perform the commands specified by that action.

At this point, you now know how to make use of the working memory and how to update it via two different methods. This concludes the tutorial for the intermediate aspects of the ACS. In the final guides on the ACS, we will cover:

- **“Basic Customization”** – How to do some basic customizations in the Clarion Library. As this relates to the ACS, this guide covers how to use delegates to setup IRL and Fixed rules in the top level.⁵
- **“Advanced ACS Setup”** – Covers how to interface the ACS with the NACS.⁶ Note that you should familiarize yourself with setting up the NACS first before looking at this guide.⁷

Remember, as always, 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.

⁵ This tutorial can be found in the “Customizations” section of the “Tutorials” folder.

⁶ This guide can be found in the “Advanced Tutorials” section of the “Tutorials” folder.

⁷ The details for setting up the NACS can be found in the “Setting up and Using the NACS” tutorial, which is located alongside the “Advanced ACS Setup” guide

Setting Up & Using the Goal Structure

© 2013. Nicholas Wilson

Table of Contents

Setting Up the Goal Structure	1
Manually Setting a Goal	2
Using Action Chunks	3

Setting Up the Goal Structure

In this section we will discuss how to set up and use the Goal Structure. Broadly speaking, the goal structure can be thought of as being a “container” within an agent that holds the agent’s goals (in the form of [GoalChunk](#) world objects). Technically speaking, it is located within the top level of the MS. However, all interaction with the goal structure (from the simulating environment) is performed directly via the [Agent](#) class. For instance, we can view the contents of the goal structure by calling the `GetInternals` method. The code below demonstrates how we might accomplish this for our agent, John:

```
//Gets all of the items in the goal structure
IEnumerable<GoalChunk> gsContents =
    (IEnumerable<GoalChunk>)John.GetInternals
    (Agent.InternalWorldObjectContainers.GOAL_STRUCTURE);

//Gets the current goal
GoalChunk currentGoal = John.CurrentGoal;
```

Like actions, goals are represented as chunks (i.e., using the [GoalChunk](#) class) and are initialized through the [World](#) singleton object:

```
GoalChunk g = World.NewGoalChunk();
```

There are also two parameters that you can set in order to “tune” the behavior of the goal structure. They are located in the parameters class of the [MotivationalSubsystem](#) and can be used to specify:

1. The behavior of the goal structure (i.e., does it behave like a list or a stack)
2. How to set the activation for the current goal (i.e., use the actual activation specified when the goal was set, or use the full activation for the goal)

Below is an example of how to set these parameters (locally) for our agent, John:

```
John.MS.Parameters.CURRENT_GOAL_ACTIVATION_OPTION =
```

```
MotivationalSubsystem.CurrentGoalActivationOptions.FULL;
```

```
John.MS.Parameters.GOAL_STRUCTURE_BEHAVIOR_OPTION =  
    MotivationalSubsystem.GoalStructureBehaviorOptions.STACK;
```

The following lines of code demonstrate how a goal is initialized and used as part of the input for a component (in this example, a [SimplifiedQBPNetwork](#) used in the bottom level of the ACS):

```
... //Elided code initializing other world objects  
  
GoalChunk g = World.NewGoalChunk();  
Agent John = World.NewAgent("John");  
  
SimplifiedQBPNetwork net =  
    AgentInitializer.InitializeImplicitDecisionNetwork(John,  
        SimplifiedQBPNetwork.Factory);  
  
net.Input.Add(g);  
  
... //Elided code performing additional initialization for the network
```

Note that all of the goals in the world are always specified as part of the “internal sensory information” and will automatically be “activated” in the [SensoryInformation](#) the next time one is perceived.

Now that we have shown you how to setup an agent to use goals, you need to know how to “activate” them. There are two methods for accomplishing this. The first is to set goals in the goal structure manually. The second is to set goals by using the “goal structure update action chunk.” We begin by looking at how chunks are set manually.

Manually Setting a Goal

The simplest way to “activate” (or add) a goal in the goal structure is to manually “set” it. We do this by calling the `SetGoal` method for the agent where the goal is to be set. The code below demonstrates how we can do this for our agent, John:

```
John.SetGoal(g, 1);
```

We specify two items when calling this method: the goal that is to be set and its “activation level”. This will “set” (or add) the goal in the goal structure. To “deactivate” (or remove) the goal from the goal structure we will call the `ResetGoal` method. In the Clarion theory, the term “reset” is equivalent to “remove” as it relates to the goal structure (as well as Working Memory). The following code demonstrates how we can manually reset (i.e., deactivate or remove) the goal in the goal structure:

```
John.ResetGoal(g);
```

These two simple methods provide you with all of the power you need to be able to use goals within the Clarion Library. However, manually setting the goals is only one of two ways to work with goals, and will often not be enough for more advanced simulations. The Clarion theory provides many more details regarding various additional methods for setting goals. For example, we can use “goal actions” in the ACS or in the MCS to perform operations on the goal structure. In the following section, we will look at how goals can be set using “goal actions” in the ACS.

Using Action Chunks

To begin, while the Clarion theory refers to actions that affect the goal structure as being “goal actions”, the implementation uses a clearer term for describing these sorts of actions. In other words, in the Clarion Library, actions that perform updates on the goal structure are defined using the `GoalStructureUpdateActionChunk` class. The contents of these action chunks contain information about the sorts of updates that are to be performed. For example, suppose we want an action that “sets” the goal `g` in the goal structure. The following code sets up such an action:

```
GoalStructureUpdateActionChunk gAct = World.NewGoalStructureUpdateActionChunk();
gAct.Add(GoalStructure.RecognizedActions.SET, g);
```

Note that we specify, as the first parameter in our `Add` method, an enumerator called `RecognizedActions`. Several classes within the Clarion Library (namely those mechanism that can be manipulated using actions, e.g., the `NonActionCenteredSubsystem`, the `GoalStructure`, `WorkingMemory`, etc.) define a `RecognizedActions` enumerator. This enumerator provides the list of commands that an action can perform on an instance of that class. The `GoalStructure` recognizes four types of actions:

- `SET`. “Adds” the goal to the goal structure
- `RESET`. “Removes” the goal from the goal structure
- `RESET_ALL`. “Removes” **ALL** of the goals from the goal structure
- `SET_RESET`. Combines the `RESET_ALL` and `SET` actions

If we want a component in the ACS to use this action, all we have to do is specify it in the output layer of the component. Below is an example of how we would set up this action in a network on the bottom level of the ACS.

```
... //Elided code performing additional initialization for the network
net.Output.Add(gAct);
```

Now, whenever the ACS selects this `GoalStructureUpdateActionChunk`, the system will perform the commands specified by that action. We will discuss another variation of this method (i.e., using a meta-cognitive module) in a later tutorial.

At this point you should have a basic foundation for building simulations in the Clarion Library using the goal structure. When you are ready to move on to the more complicated aspects of the MS (including integrating the MS with the MCS), the next tutorial, “*Intermediate MS & MCS Setup*”, can be found in the “*Intermediate Tutorials*” section of the “*Tutorials*” folder.

Remember, 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.

Intermediate MS & MCS Setup

© 2013. Nicholas Wilson

Table of Contents

Setting Up and Using Drives and Meta-Cognitive Modules	1
Initializing a Drive.....	1
The Drive Equation.....	3
Stimulating a Drive	4
Accessing Agent Meta Info.....	5
Initializing a Meta-Cognitive Module.....	6
The Goal Selection Equation.....	7
Correlating Drives and Meta-Cognitive Modules	8
Meta-Cognitive Module Integration	9

Setting Up and Using Drives and Meta-Cognitive Modules

Drives use factors from both the internal and external state information (located within the [SensoryInformation](#) object) to transform them into a “drive strength” (i.e., the amount of activation for a drive). However, without mechanisms to process these drive strengths and make decisions based upon them, the drives alone will have little effect on the overall operation of an agent. Therefore, we rely on meta-cognitive modules to make decisions based upon these drives strengths (as well as other factors) and to initiate a variety of internally-directed meta-cognitive actions.

It is because of the tight coupling between drives and meta-cognitive modules that we have chosen to present these concepts together. We begin by demonstrating how to set up and initialize a drive in the bottom level of the motivational subsystem. Afterwards, we present an example of a meta-cognitive module that will combine the drive strengths from various drives and then use that information to update the goal structure using a [GoalStructureUpdateActionChunk](#). But first let’s begin by describing how to initialize a drive.

Initializing a Drive

To start, we should note that a drive object is considered to be a “special form” of a functional object within the Clarion Library. We say this because the drive object doesn’t directly extend from the base [ClarionComponent](#) class but is essentially just a wrapper around an [ImplicitComponent](#) that provide a few additional drive-specific features.

The Clarion Library comes equipped with all of the primary drives specified in the technical specification document. Each of these drives is defined by their own class and use the following naming convention:

PrimaryDriveNameDrive

For example, the class names for the food drive and dominance and power drive are “[FoodDrive](#)” and “[DominancePowerDrive](#)”¹ respectively. These drives are initialized using the `InitializeDrive` method in the `AgentInitializer` class. For example, the following code will initialize a [FoodDrive](#) in our agent, John:

```
FoodDrive food = AgentInitializer.InitializeDrive(John, FoodDrive.Factory, .5);
```

Drives are designated as being one of the following groups within the bottom level of the MS:

- Approach drives (i.e., BAS drives)
- Avoidance drives (i.e., BIS drives)
- “Both” drives (i.e., both approach and avoidance oriented)
- “Unspecified” drive types (i.e., they either don’t belong to a behavioral system, or their behavioral system has not been specified)²

In general, you shouldn’t need to access this group specification in order to use a drive. All of the “built-in” drives in the Clarion Library (based on the Clarion theory) specify their appropriate group during initialization. However, if you want to change the group affiliation of a built-in drive, you can specify it as an optional parameter during initialization. Below is an example of what this might look like if we were to change the group specification for the [FoodDrive](#) in our agent, John:

```
FoodDrive foodDrive = AgentInitializer.InitializeDrive  
(John, FoodDrive.Factory, .5,  
MotivationalSubsystem.DriveGroupSpecifications.BOTH);
```

Additional, you may also want to create your own “deficit change processor” to process how the deficits change over time.³ This is accomplished using “custom delegates”, however implementing something like this is a more advanced concept outside of the scope of this guide.⁴

¹ For those drives that contain the “&” symbol in their name, the “&” conjunction has been left out of the class name for those drives.

² The specification of the drive’s behavior system can be found in the documentation for the drive’s class as well in an instance of a class (via the `BehaviorSystem` property).

³ By default, drive deficits change by a multiplicative factor of the `DEFICIT_CHANGE_RATE` (located in the local parameters class instance of the drive).

⁴ Details on how to implement a “custom delegate” can be found in the “*Basic Customization*” tutorial in the “*Customizations*” section.

The Drive Equation

Recall that we mentioned earlier that a drive is just a “wrapper” for an `ImplicitComponent`, so the next thing we need to do is initialize an `ImplicitComponent` inside of the `FoodDrive`. Ideally, you will want to use something like a pre-trained `BPNetwork` in your drive. However, since pre-training an implicit component can get a bit complicated, for our current example we will demonstrate a quicker and easier component, the `DriveEquation`, instead.⁵

The `DriveEquation` is an extension of the Clarion theory.⁶ In general, you should usually only use it when you are testing the configuration of your agents. Once an agent is properly configured, you should replace it with a more “distributed” type of implicit component (such as a `BPNetwork`). The following code demonstrates how to set up a `DriveEquation` within the `FoodDrive` of our agent, John:

```
DriveEquation foodEq =  
    AgentInitializer.InitializeDriveComponent(foodDrive, DriveEquation.Factory);
```

We initialize the equation by calling the `InitializeDriveComponent` method located in the `AgentInitializer`. Note that the `DriveEquation` class uses the following equation for calculating the drive strength of a drive:

$$DS_{d,t} = Gain_{universal} \times Gain_{system} \times Gain_{drive} \times Stimulus_{d,t} \times Deficit_d + Baseline_d$$

The details regarding this equation can be found in addendum #1 of the Clarion Technical Specification (located on Ron Sun’s [website](#)). What is of particular importance here is the series of variables defined by the equation.

We refer to these variables as “typical drive inputs.” They can be found within variables located in either the “parameters class” for the drives (e.g., `UNIVERSAL_GAIN`⁷, `DRIVE_GAIN`, or `BASELINE`) or in the “parameters class” for the motivational subsystem (e.g., `SYSTEM_GAIN`).

Generating dimension-value pairs to represent these “typical inputs” can be readily generated by calling the `GenerateTypicalInputs` method, which is statically available in the `Drive` class. Furthermore, if the `ImplicitComponent` within a drive contains any of these “typical inputs”, the system will automatically fill them in with the values appropriate for those inputs.

When an instance of `DriveEquation` is initialized using the `AgentInitializer`, the following will already be configured for you in the `DriveEquation` instance that gets returned:

⁵ The Clarion Library comes equipped with a feature (the `ImplicitComponentInitializer`) that can aid with the initialization and pre-training of implicit components. The details on how to use it can be found in the “Useful Features” tutorial located in the “Features & Plugins” section.

⁶ Located in the `Clarion.Framework.Extensions` namespace.

⁷ `UNIVERSAL_GAIN` is a `static` parameter (in the `DriveParameters` class) and therefore always applies to all drives regardless of the subclass from which it is changed.

1. The inputs to the `DriveEquation` (populated with the dimension-value pairs representing the “typical inputs” for the equation)
 - Note that the dimension ID will be set to the `Type` of the drive in which the equation is being initialized and the value IDs will be of the enumerated type `Drive.MetaInfoReservations`.
 - Note also that these inputs are generated by statically calling the `Drive.GenerateTypicalInputs` method.
2. The output from the `DriveEquation` (specified as a dimension-value pair representing the “drive strength” for that drive).
 - Note that the dimension ID will be set to the `Type` of the drive in which the equation is being initialized and the value ID will be the `DRIVE_STRENGTH` specification from the enumerated type `Drive.MetaInfoReservations`.
 - Note also that this output is generated by statically calling the `Drive.GenerateTypicalOutput` method

This pre-loading behavior of the input layer (as was described in #1, above) is unique to `DriveEquation`. However, the pre-loading of the output layer is not. The Clarion Library requires that, for those implicit components that are being used within a drive, the output layer of those components **MUST** contain the “DRIVE_STRENGTH” dimension-value pair (as was described in #2, above) and can **ONLY** contain that dimension-value pair. If you attempt to put a different dimension-value pair in the output layer, the `ImplicitComponent` will fail when you try to commit it to the drive. As a result of this requirement, the `InitializeDriveComponent` method will automatically populate the output layer of the `ImplicitComponent` it generates with the appropriate “DRIVE_STRENGTH” dimension-value pair for the drive in which the `ImplicitComponent` is initialized.

Finally, as is always the case, once you have finished initializing the drive’s `ImplicitComponent`, you must commit that component to the drive. In addition, you will also then need to commit the drive itself to the agent. The following code demonstrates how this is done for the `FoodDrive` of our agent, John:

```
foodDrive.Commit(foodEq);
John.Commit(foodDrive);
```

Stimulating a Drive

If you use only the “typical inputs,” then you will only need to specify the activation of the `STIMULUS` input. The following code demonstrates how you might specify the `STIMULUS` for the `FoodDrive` in a sensory information object (generated during the running of a task) for our agent, John:

```
si = World.NewSensoryInformation(John);
```



```
si[FoodDrive.MetaInfoReservations.STIMULUS, typeof(FoodDrive).Name] = 1;
... //Elided initialization of other aspects of the sensory information
John.Perceive(si);
... //Elided code for running the rest of the task
```

Note that you are not required to use the “typical inputs” for your drives. However, if you don’t use them, you will then have to specify the activations for the inputs of your drives during the running of the task every time you create a new sensory information object.

Accessing Agent Meta Info

As you may have noticed from the previous code segment, we do not need to “add” the STIMULUS variable to the sensory information object. This is because the `NewSensoryInformation` method automatically populates the sensory information object with all of the agent’s “meta information” (which includes the drive inputs and outputs, among other things) before it is returned to the simulating environment. Instead of “adding” the internal meta information, we can simply access it from the sensory information object (as was demonstrated in the above code).

The value for any agent “meta information” (such as drive inputs and outputs) will usually be the string name of the class from which the meta information was specified. For example, the stimulus for the food drive could have also been set using the following:

```
si[FoodDrive.MetaInfoReservations.STIMULUS, "FoodDrive"] = 1;
```

The dimensions for the “meta information” are usually specified by an enumerator called `MetaInfoReservations`. This convention for defining meta-information dimension-value pairs is intuitive (and also correct with regard to proper representation within Clarion). It is also a very useful organization. For example, it is very easy to figure out what objects are declaring what information. For example, the STIMULUS dimension in the above sensory information object will contain specifications for all of the drives that are set up to use this typical input.

It can sometimes be a little tricky trying to figure out what meta information is available for a given agent. The best thing to do here while building the “run” portion of your simulation is to start by simply getting a sensory information object from the `World` and writing it out to the console (or set a breakpoint on your debugger for immediately after the new sensory information call). This should give you an idea of what meta info is being specified by an agent. Below is an example

(from the “*Full Hello World*” simulation sample⁸) of what a sensory information object, which contains meta information, might look like:

Activations:

```
(Dimension = BASELINE, Value = AffiliationBelongingnessDrive), Activation = 0,  
(Dimension = BASELINE, Value = AutonomyDrive), Activation = 0,  
(Dimension = BOTH, Value = MotivationalSubsystem), Activation = 0,  
(Dimension = DEFICIT, Value = AffiliationBelongingnessDrive), Activation = 0,  
(Dimension = DEFICIT, Value = AutonomyDrive), Activation = 0,  
(Dimension = DRIVE_GAIN, Value = AffiliationBelongingnessDrive), Activation = 0,  
(Dimension = DRIVE_GAIN, Value = AutonomyDrive), Activation = 0,  
(Dimension = DRIVE_STRENGTH, Value = AffiliationBelongingnessDrive), Activation = 0,  
(Dimension = DRIVE_STRENGTH, Value = AutonomyDrive), Activation = 0,  
(Dimension = STIMULUS, Value = AffiliationBelongingnessDrive), Activation = 0,  
(Dimension = STIMULUS, Value = AutonomyDrive), Activation = 0,  
(Dimension = UNIVERSAL_GAIN, Value = Drive), Activation = 0
```

Note that the activations all meta information will be 0 when the sensory information is first returned by the `World`. This is because meta information is usually set internally. We set the drive `STIMULUS` value within the simulating environment simply as a matter of convenience. However, ideally, we would prefer that such operations be performed internally by a meta-cognitive module whose job it is to determine drive stimulus signals.⁹

At this point, you should now have everything you need in order to set up drives in the bottom level of the MS. However, in order to use these drives, you also need to implement one or more meta-cognitive module(s) that will act based on these drives. So let’s turn to discussing how to initialize meta-cognitive modules, after which we will demonstrate how to integrate the drives with them.

Initializing a Meta-Cognitive Module

Operationally, a meta-cognitive module acts essentially like a “mini-ACS,” except that the actions of a meta-cognitive module is directed towards manipulating the internal aspects of an agent (such as the goals in the goal structure, certain parameters within other subsystems, etc.). A meta-cognitive module can be comprised of any combination of `ImplicitComponent` instances in the bottom-level and `RefineableActionRule` instances in the top level. However, unlike the ACS, meta-cognitive modules are more limited in their capabilities. For example, a meta-cognitive module does not use `FixedRule` instances in the top level and the action recommendations from the top and bottom levels are always combined.

In general, you should mainly set up a meta-cognitive module using the bottom level. This makes sense conceptually, since meta-cognitive processes tend to be sub-conscious. This being said, rule extraction and refinement is enabled by default within the modules. Note, however, that no mechanism is provided for delivering the specialized feedback that would be needed in order to take advantage of RER

⁸ Located in the “*Intermediate*” section of the “*Samples*” folder

within these modules. In fact, in order to leverage the RER capabilities in the MCS, we would need to develop a meta-cognitive module that could interpret both internal and external factors and then deliver the reinforcement signal to the other modules in the MCS.⁹

The process for setting-up a meta-cognitive module is similar to initializing a drive. For this tutorial, we will look at a commonly used module: the [GoalSelectionModule](#). Below is an example of how you would initialize this module within the agent, John:

```
GoalSelectionModule gsm =
    AgentInitializer.InitializeMetaCognitiveModule
        (John, GoalSelectionModule.Factory);
```

After we have initialized the module, we can start populating it with implicit components and rules. You can initialize these components by calling either the [InitializeMetaCognitiveDecisionNetwork](#) or the [InitializeMetaCognitiveActionRule](#) methods located within the [AgentInitializer](#). Below is an example of how we could initialize a [GoalSelectionEquation](#)¹⁰ within the bottom level of the [GoalSelectionModule](#):

```
GoalSelectionEquation gse =
    AgentInitializer.InitializeMetaCognitiveDecisionNetwork
        (gsm, GoalSelectionEquation.Factory);
```

The input layer for the implicit components (and conditions of any rules for that matter) of a meta-cognitive module can consist of any type of (descriptive) [IWorldObject](#) (just like in the ACS). However, they can also make use of several other types of inputs that you wouldn't normally use in the ACS. Specifically, meta-cognitive modules will often specify "DRIVE_STRENGTH" dimension-value pairs (from the previous section) as part of the input layer of their implicit components.

The Goal Selection Equation

Remember that the bottom level of the MS is in charge of determining drive strengths based on the combination of stimulus from the sensory information as well as certain "individual differences" considerations (i.e., gains, deficit, etc.). For example, the [GoalSelectionEquation](#) combines the drive strengths (and any other descriptive world object) to make goal recommendations for the goal structure based on the following equation:

⁹ This capability, while within the scope of the Clarion theory, is currently only conceptual. However, future research into this concept may eventually lead to the implementation of such a module.

¹⁰ Like the [DriveEquation](#), the [GoalSettingEquation](#) is an extension component and can be found in the *Clarion.Framework.Extensions* namespace.

$$GS_{g,t} = \sum_d ds_{d,t} \times Relevance_{d,g} + \sum_i dv_{i,t} \times Relevance_{dv,g}$$

Let's break down this equation to better understand how to set up the [GoalSelectionEquation](#) within your code. The first half of the equation relates specifically to the drive strengths. This part of the equation sums together the drive strengths for all of the drives. In addition, a weighting factor is applied to each drive strength. This weighting factor specifies the “relevance” that each drive has to the goal whose “goal strength” is being calculated. The second half of the equation is essentially the same as the first half, except that it applies the process to the other descriptive world objects (e.g., dimension-value pairs, chunks, etc.) that are “relevant” to the goal. The goal strength of each goal, which is the output of this equation, indicates the “value” for setting a goal within the goal structure.

Correlating Drives and Meta-Cognitive Modules

The input layer of the [GoalSelectionEquation](#) can contain any number of “drive strength dimension-value pairs” or other relevant descriptive world objects. The output layer can **ONLY** contain goal structure update action chunks. Recall that in the previous tutorial we demonstrated how a [GoalStructureUpdateActionChunk](#) could be used to set (or remove) goals within the [GoalStructure](#).¹¹ The [GoalStructureUpdateActionChunk](#) is what enables the [GoalSelectionModule](#) to update the [GoalStructure](#).

Once the [GoalSelectionEquation](#) is set up within the [GoalSelectionModule](#), it gets used to calculate the goal strengths, which the [GoalSelectionModule](#) then uses to select a [GoalStructureUpdateActionChunk](#). The goal associated with that action chunk is set (or removed) in the [GoalStructure](#) by initiating a goal structure update event within the system. That event will prompt the MS, which will perform the update based on what is specified by the action.

At this point, let's step through an example to demonstrate the process of setting up the [GoalSelectionModule](#). This example correlates the [FoodDrive](#) to a [GoalStructureUpdateActionChunk](#) that “resets” the goal structure and then “sets” a goal, *g*, in the goal structure of our agent, John. The first line of our example is as follows:

```
gse.Input.Add(foodDrive.GetDriveStrength());
```

This line adds the “drive strength dimension-value pair” of the [FoodDrive](#) to the input layer of the [GoalSelectionEquation](#) that we initialized earlier. The next three lines initialize the [GoalStructureUpdateActionChunk](#) that “sets” goal *g* in the goal structure and add it to the output layer of the [GoalSelectionEquation](#):

```
GoalStructureUpdateActionChunk gAct = World.NewGoalStructureUpdateActionChunk();
```

¹¹ See the “*Setting Up & Using the Goal Structure*” tutorial located in the “*Basics Tutorials*” section of the “*Tutorials*” folder.

```
gAct.Add(GoalStructure.RecognizedActions.SET_RESET, g);  
gse.Output.Add(gAct);
```

After the input and output layers have been set up we need to specify the relevance that each input has to each output. This is done using the following convention for each of the input → output relevance pairings:

```
SomeGoalSelectionModule.SetRelevance(SomeGoalStructureUpdateActionChunk,  
    SomeDrive or SomeWorldObject, SomeRelevanceValue);
```

To correlate the `FoodDrive` to the `GoalStructureUpdateActionChunk` for our example the code will look something like this:

```
gsm.SetRelevance(gAct, foodDrive, 1);
```

Finally, as was the case with initializing a drive, once we are finished setting up a component for our meta-cognitive module, we need to commit it to the module. Additionally, after the module has been completely initialized, we have to commit it to the agent. The code below shows how this would be done in our current example:

```
gsm.Commit(gse);  
John.Commit(gsm);
```

Meta-Cognitive Module Integration

Once all of the components and modules have been committed, the system will automatically integrate them into the internal processes of the agent. No other interventions are required to make the modules interact correctly with the other parts of the system. In general, when a meta-cognitive module has been set up and committed to an agent, it will operate “behind-the-scenes.” However, if you would like to view the inner workings or outcomes from either the motivational subsystem or the meta-cognitive modules, several features are available in the Clarion Library to accomplish this.

For instance, the results from either updating the drive strengths in the MS or choosing meta-cognitive actions in the MCS will be viewable as part of the `SensoryInformation` that was perceived by the agent **AFTER** the agent is finished choosing an external action based upon it. Below is an example (from the “*Full Hello World*” simulation sample¹²) of what this might look like:

```
Activations:  
    (Dimension = BASELINE, Value = AffiliationBelongingnessDrive), Activation = 0,  
    (Dimension = BASELINE, Value = AutonomyDrive), Activation = 0,  
    (Dimension = BOTH, Value = MotivationalSubsystem), Activation = 0,  
    (Dimension = DEFICIT, Value = AffiliationBelongingnessDrive), Activation = 0.41,
```

¹² Located in the “*Intermediate*” section of the “*Samples*” folder

(Dimension = DEFICIT, Value = AutonomyDrive), Activation = 0.57,
(Dimension = DRIVE_GAIN, Value = AffiliationBelongingnessDrive), Activation = 1,
(Dimension = DRIVE_GAIN, Value = AutonomyDrive), Activation = 1,
(Dimension = DRIVE_STRENGTH, Value = AffiliationBelongingnessDrive), Activation = 0.41,
(Dimension = DRIVE_STRENGTH, Value = AutonomyDrive), Activation = 0.57,
(Dimension = GoalChunk:Salute, Value = Salute), Activation = 1,
(Dimension = Salutation, Value = Goodbye), Activation = 0,
(Dimension = Salutation, Value = Hello), Activation = 1,
(Dimension = STIMULUS, Value = AffiliationBelongingnessDrive), Activation = 1,
(Dimension = STIMULUS, Value = AutonomyDrive), Activation = 1,
(Dimension = UNIVERSAL_GAIN, Value = Drive), Activation = 0

You can also turn on logging, which, depending on the level, will provide you with additional details concerning the internal operations of the system (including the motivational subsystem and meta-cognitive modules). The details concerning how to use the logging feature can be found in the “*Using Features*” tutorial (located in the “*Features & Plugins*” section of the “*Tutorials*” folder).

This concludes the tutorial for setting up drive and meta-cognitive modules. At this point, you should have the necessary foundation for using these aspects of the Clarion theory. Note, however, that the example we demonstrated herein was fairly easy, however, setting up the MS and MCS can actually become quite complex. If you find that you need additional information on setting up a particular meta-cognitive module, please consult the API resource document (located in the “*Documentation*” folder). It will provide you with additional details for how to use any of the “pre-packaged” modules that come with the Clarion Library.

Furthermore, we suggest consulting the “*Advanced Customization*” tutorial (located in the “*Customizations*” section of the “*Tutorials*” folder). This tutorial provides details on how to create a custom drive (as well as other types of custom components). Additionally, while it is not included as part of the Clarion Library package, a guide on how to create a custom meta-cognitive module is available upon request. However, be forewarned that implementing a custom meta-cognitive module is a **VERY** advanced (i.e., developer level) undertaking. Therefore, before endeavoring to undertake any developer level (or even advanced level) customizations, you will need to have a thorough and complete understanding of the Clarion theory as well as extensive experience working with the Clarion Library.

Remember, as always, if you run into any problems, have additional questions, want to report a bug, or wish to request the tutorial on implementing a custom meta-cognitive module, you can contact us at clarion.support@gmail.com or post on our message boards at <http://www.clarioncognitivearchitecture.com>.

Basic Customization

© 2013. Nicholas Wilson

Table of Contents

Customized Methods (Using Delegates)	1
Specifying Delegates as Parameters during Initialization	2
Creating Custom Rules	3
Using the SupportCalculator Delegate to Set Up an IRL Rule.....	3
Initializing the IRL Rule.....	5
Using the SupportCalculator Delegate to Set Up a Fixed Rule	5
A Note on the Generically Typed DimensionValuePair<DType, VType> Class.....	7
Initializing the Fixed Rule.....	8
Generic Equations	9

Customized Methods (Using Delegates)

For several of the algorithms specified by the Clarion theory (e.g., eligibility checking, rule refinement, match calculating, etc.), the library only implements the default method, even though there may be other ways to perform those operations. This being said, you may run into instances where you will need to designate a different operation to replace the system’s default behavior for a certain algorithm. To address this need, the Clarion Library leverages C#’s [delegate](#) feature and defines a series of “delegate signatures” that can be used to define your own custom algorithms. Whenever a custom method is specified during initialization, the system will use this method in lieu of its default behavior.

So let’s begin our tutorial on setting up and using delegates by looking at one of the algorithms that you are most likely to want to customize: checking the eligibility of a component. For customizing this method, the Clarion Library defines the [EligibilityChecker](#) delegate. The signature for this delegate is:

```
public delegate bool EligibilityChecker
    (ActivationCollection currentInput = null, ClarionComponent target = null);
```

The default eligibility checking algorithm for an [ImplicitComponent](#) is simply to return the value of the component’s “ELIGIBILITY” parameter. While this provides a simple way for you to manually prevent or allow a component to be used by the system, it does not, otherwise, provide any additional logic for determining the eligibility. For example, suppose you wanted to define some conditions for when a component should be used and you want the system to be able to integrate this “condition eligibility check”. To accomplish this, you need to implement a method

that will perform the eligibility checking operation and then inform the system of the result of this check.

Implementing a custom eligibility checking delegate is accomplished by creating a method within your code that uses the same inputs and returns a value of the same type as is specified by the “delegate signature” (from above). The following pseudo-code demonstrates how such a method might look:

```
public bool Custom_EligibilityCheck (ActivationCollection currentInput = null,
    ClarionComponent target = null)
{
    ... // Do operations to determine if the target component is eligible
    return true or false;
}
```

After the method is set up, if we wanted a particular component to make use of it when checking its eligibility, we would need to specify the method as a parameter (in the form of an `EligibilityChecker` delegate) during the initialization of that component. The system will use our custom delegate method to check the eligibility of any components with which it was initialized.

Specifying Delegates as Parameters during Initialization

For the most part, we specify delegate methods during the initialization (using the `AgentInitializer`) of an internal (functional) object. For example, suppose we created a method called `Custom_EligibilityCheck` (in our own code) that matches the signature for the `EligibilityChecker` delegate. The following code demonstrates how our custom method could be specified as part of the initialization of a `BPNetwork` in the bottom level of the ACS of the agent, John:

```
BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, BPNetwork.Factory, (EligibilityChecker)Custom_EligibilityCheck);
```

Note that we have explicitly casted our custom method to the correct delegate specification (i.e., `EligibilityChecker`) during the initialization call. We do this because it is required in order to pass a `delegate` using the `dynamic` type designation. This being said, there are other ways to specify the type of our delegate method. Specifically, we can “wrap” our delegate method in a property and use that property instead of explicitly casting our custom method. From the previous example, initializing the `BPNetwork` can alternatively be done as follows:

```
// During the initialization method:
BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, BPNetwork.Factory, CustomEligibilityCheckerDelegate);
... // At some other point in your code:
public EligibilityChecker CustomEligibilityCheckerDelegate
{
    get
    { return Custom_EligibilityCheck; }
}
```


You can use whichever method you'd prefer, however, we recommend using the property method as it is generally cleaner and easier to follow.

Specifying custom delegate methods during the initialization of an internal object is usually optional. To find out which delegates an internal (functional) object can use, consult the API resource document (in the “*Documentation*” folder) for the **factory** class that is used to generate that internal object.

At this point, we should point out that while custom delegates are mainly optional, there are some internal (functional) objects that do actually **require** you to implement some custom delegates in order to be initialized.¹ In the following section, we will look at two such internal objects: IRL rules, and fixed rules.

Creating Custom Rules

Depending on the specifics of the task you are simulating, you may discover that you need to implement a rule whose dynamics are more complex than what can be captured using a simple [RefineableActionRule](#). To handle this case, the Clarion theory defines two types of rules: IRL rules, and fixed rules.²

In the Clarion Library, we have implemented these rule types using two classes: the [IRLRule](#) class, and the [FixedRule](#) class. These classes provide the large majority of the mechanisms that are required for the rules. All you need to do is specify a single custom delegate method in order to initialize either an IRL rule or a fixed rule. More specifically, you need to define the method that is used for calculating the support for the rule (via the [SupportCalculator](#) delegate). The system uses this support measure to determine if a rule is eligible for action recommendation at a given time step (based on a “partial match threshold”³). The signature for the [SupportCalculator](#) delegate is as follows:

```
public delegate double SupportCalculator
    (ActivationCollection currentInput, Rule target = null);
```

To help clarify this concept further, let's look at a few examples where we may want to use each of these rule types.

Using the SupportCalculator Delegate to Set Up an IRL Rule

In this section we will cover an example of where an [IRLRule](#) would be necessary. One of the most common instances when this rule type is necessary is when a certain factor of a rule's condition is, itself, conditioned upon another factor of that condition. For example, let's assume that we have the dimension-value pairs: {dim1, a}, {dim1, b}, {dim2, c}, and {dim2, d} and the action: {do_something}. Now, suppose we want to create the following rule:

¹ Again, consult the API resource document of the factory class for this information

² See Sun (2003) for more details

³ Captured by the PARTIAL_MATCH_THRESHOLD parameter

If {dim1, a} AND {dim2, c}, but NOT {dim2, d} then recommend the {do_something} action, otherwise don't recommend it

To capture the condition of this rule, we will need to write a custom method that can be initialized using the [SupportCalculator](#) delegate signature. Using pseudo-code, we could express this method as follows:

```
public double CalculateSupport_IRL(ActivationCollection currentInput, Rule r)
{
    return the maximum activation between {dim1, a} and {dim2, c} if both
           {dim1, a} and {dim2, c} are specified as being part of the condition
           while {dim2, d} is both specified as NOT being part of the condition
           and is NOT activated in currentInput. Otherwise, 0
}
```

Note that the [IRLRule](#) class derives from the [RefineableActionRule<>](#) class, so it has all of the same refinement capabilities as that rule type. Therefore, we can use the Clarion Library's built-in generalization and specialization processes to automatically refine our [IRLRule](#) without needing any additional customizations be set up to facilitate it. For instance, suppose that the system decided that the following refined rule is better able to capture the dynamics of our current task example:

If {dim1, a} OR {dim1, b} AND {dim2, c}, but NOT {dim2, d} then recommend the {do_something} action, otherwise don't recommend it

We want to make sure, when constructing custom delegate methods, that we capture the most essential factors for your rule while still maintaining enough flexibility to accommodate any refinements that may be made to the rule. For our current example, the main factors are the co-activation of **any** dimension-value pair in **dim1** **and** the {dim2, c} dimension-value pair, but **NOT** the activation of the {dim2, d} dimension-value pair. Therefore, our support calculator method should capture these factors. Below is an example of how we might express this in C#:

```
public double CalculateSupport_IRL(ActivationCollection currentInput, Rule r)
{
    var d1 = from d in currentInput
              where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "dim1" &&
                    r.GeneralizedCondition[d.WORLD_OBJECT] == true
              select d;

    return (r.GeneralizedCondition["dim2","c"] == true &&
            currentInput["dim2", "c"] > 0 &&
            r.GeneralizedCondition["dim2", "d"] == false &&
            currentInput ["dim2", "d"] == 0)? d1.Max(e => e.ACTIVATION) : 0;
}
```

To fully understand the above code, you need to be aware of the C# language features that it is leveraging. The first line uses [LINQ](#) to get all of the dimension-value pairs in **dim1** that are specified as being part of the condition. The second line

(i.e., the `return` line) uses a combination of [lambda expressions](#) and the [conditional operator](#) to return the maximum activation (which captures the **OR** operation) of the dimension-value pairs that were found in the first line if the dimension-value pair {dim2, c} is activated and {dim2, d} is not. Alternatively, if the condition is not true (i.e., either {dim2, c} is not activated or {dim2, d} is), then the second line will return zero.

Now that we have setup our custom method, let's look at how we would go about initializing the `IRLRule`.

Initializing the IRL Rule

To setup the `IRLRule` we need to do three things. First, we need to initialize it:

```
// During the initialization method:
IRLRule rule1 = AgentInitializer.InitializeActionRule
    (John, IRLRule.Factory, some_action, SupportDelegate);

DimensionValuePair dv1 = World.NewDimensionValuePair("dim1", "a");
DimensionValuePair dv2 = World.NewDimensionValuePair("dim1", "b");
DimensionValuePair dv3 = World.NewDimensionValuePair("dim2", "c");
DimensionValuePair dv4 = World.NewDimensionValuePair("dim2", "d");

... // At some other point in your code:
public SupportCalculator SupportDelegate
{
    get
    { return CalculateSupport_IRL; }
}
```

Second, we need to setup the initial condition for the rule:

```
// Elided rule initialization (see above)

rule1.GeneralizedCondition.Add(dv1, true);
rule1.GeneralizedCondition.Add(dv2, false);
rule1.GeneralizedCondition.Add(dv3, true);
rule1.GeneralizedCondition.Add(dv4, false);
```

Finally, we need to commit the rule:

```
John.Commit(rule);
```

That is everything you need to do to setup an `IRLRule`. So let's turn our attention now to an example of how to setup a `FixedRule`.

Using the SupportCalculator Delegate to Set Up a Fixed Rule

The process for setting up a `FixedRule` is very similar to setting up an `IRLRule`. A good example of where we might want to use a `FixedRule` is when part of the algorithm for determining the support of a rule requires that we perform some sort of mathematical translation on parts of the `SensoryInformation`. For instance,

let's suppose we want to setup the following rule, which determines whether a "carry-over" operation is needed when performing addition:

If {operator, +} and {digit1, x} + {digit2, y} > 9, then recommend the {carry-over} action, otherwise don't recommend it.

We capture the condition of this rule within our custom delegate method. The following code demonstrates how we would do this in pseudo-code:

```
public double CalculateSupport_FR(ActivationCollection currentInput, Rule r)
{
    return 1, if {operator, +} is activated and {digit1, x} + {digit2, y} > 9
        Otherwise, 0
}
```

While it is a little longer, expressing the pseudo-code in C# would look something like this:

```
public double CalculateSupport_FR(SensoryInformation currentInput, Rule r)
{
    if (currentInput["operator", "+"] > 0)
    {
        var d1 = (from d in currentInput
                 where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "digit1"
                 select d).OrderByDescending(e => e.ACTIVATION).First();

        var d2 = (from d in currentInput
                 where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "digit2"
                 select d).OrderByDescending(e => e.ACTIVATION).First();

        if (((int)d1.WORLD_OBJECT.AsDimensionValuePair.Value.AsComparable) +
            ((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsComparable) > 9)
            return 1;
        else return 0;
    }
    else
        return 0;
}
```

The LINQ queries (i.e., the first 2 lines inside of the first `if` statement) are used to find the maximally activated digits (represented as dimension-value pairs). Note that we assume, in this example, that only one digit will be activated for each dimension, so getting the `First` value from the dimension (after it has been sorted in descending order by activation) should give us digits `x` & `y` (from the pseudo-code). The first `if` statement checks to see if the `+` operator is activated in the `SensoryInformation`. The second `if` statement checks to see if the sum of the two digits will require that the `carry-over` action be performed and will return 1 (i.e., the action should be recommended) if it does, or 0 (i.e., the action shouldn't be recommended) if it does not.

A Note on the Generically Typed `DimensionValuePair<DType, VType>` Class

You may have noticed the following from the example we are using to demonstrate how to set up a `FixedRule`:

```
Value.AsComparable
```

Beginning with version 6.1.0.7 of the Clarion Library, dimension-value pairs actually come in two flavors:

- The standard `DimensionValuePair` class
- A generically typed `DimensionValuePair<DType, VType>` subclass

Although this is the case, you likely have not realized it until this point, since the vast majority of your interaction with the `World` class automatically gives you the generically type `DimensionValuePair<DType, VType>`. This generically typed `DimensionValuePair<DType, VType>` is very useful as it significantly simplifies your interaction with dimension-value pairs by reducing the amount of explicit casting that is necessary. For example, suppose we did the following:

```
World.NewDimensionValuePair("Digit", 1)
```

If we were to subsequently call the `Dimension` or `Value` properties of the `DimensionValuePair<DType, VType>` object that is returned by that method, the dimension or value that we will get back will already be cast as the appropriate type (i.e., as a `string` or an `int` respectively). This is the case whenever working with generically typed dimension-value pairs.

While this is certainly a very useful addition, the primary reason that we added the generically type `DimensionValuePair<DType, VType>` class, was to enable values of different types to inhabit the same dimension. This has been accomplished by implementing a special “wrapper” class (called `V`) within the standard `DimensionValuePair` class. This wrapper class has specially overloaded `IComparable` methods that allow differently typed values to be compared using their `ToString()` representations. While there are **MANY** benefits to implementing this, the downside is that calling the `Value` property of the standard `DimensionValuePair` (e.g., when using the `AsDimensionValuePair` property of the `IWorldObject` interface) will actually return the `V` class “wrapper” instance as opposed to the underlying `IComparable` value itself (which is what is returned by the `Value` property of `DimensionValuePair<DType, VType>`).

We have gotten around this issue by defining a property, called `AsComparable`, within the `V` class that exposes the underlying `IComparable` value. Note, however, that you will likely also have to explicitly cast this value back to its appropriate type to make use of it.⁴ The following example (taken from the previous `FixedRule`

⁴ We acknowledge that this solution is “suboptimal”, however, unless (or until) Microsoft decides to allow custom implicit casting to interfaces or (better yet) allows custom down casting in C#, this is the best solution we could come up with.

example) demonstrates how we might go about exposing (and explicitly casting) the underlying `IComparable` value when working with the standard `DimensionValuePair` class:

```
if (((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) +
    ((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) > 9)
    return 1;
else return 0;
```

Now that we have addressed this point, let's look at how we might go about initializing a `FixedRule`.

Initializing the Fixed Rule

Setting-up the `FixedRule` is essentially the same process as it was for the `IRLRule`:

```
// During the initialization method:
FixedRule rule = AgentInitializer.InitializeActionRule
    (John, FixedRule.Factory, carry_action, SupportDelegate);
John.Commit(rule);

... // At some other point in your code:
public SupportCalculator SupportDelegate
{
    get
    { return CalculateSupport_FR; }
}
```

Note that, since fixed rules are not refineable, we technically do not need to specify a condition for them. We call these types of fixed rules “condition-less.” In addition, by default, fixed rules are also not deletable (e.g., via density considerations⁵). In general, you will want to use a fixed rule to capture the “one-shot-learning” paradigm. That is, fixed rules are usually obtained in some sort of explicit fashion (e.g., via instruction).

This concludes the basic customization tutorial. You should now have everything you need to know in order to do basic customizations using delegates within the Clarion Library. Feel free to explore the documentation to discover all of the places throughout the system where custom delegate methods can be used. Using delegate methods is a great way to customize your simulations without needing to “reinvent the wheel”, so to speak.

However, if you find that you have reached the limit of what custom delegates can provide or you feel like taking on a challenge, then you should know that you can implement your own customized internal (functional) objects (e.g., implicit components, drives, rules, etc.). Details on how to do this can be found in the “*Advanced Customization*” tutorial. However, be forewarned that implementing a custom internal (function) object is **NOT** a simple process. Therefore, you should have a thorough understanding of the Clarion theory as well as significant

⁵ Although this can be enabled by toggling the `DELETABLE_BY_DENSITY` parameter

experience working with the Clarion Library before endeavoring to take on this sort of customization.

Generic Equations

It is not uncommon that, during the development and tuning of your task, you may find it both quicker and more preferable to temporarily “shortcut” some of the more laborious aspects of initialization (e.g., pre-training implicit components such as neural networks) for the bottom level of an agent’s subsystems. Frequently, these implicit components are simply expected to report the results of an already known equation. For this sort of event, the Clarion Library provides [GenericEquation](#)⁶, which can easily be setup and “plugged into” the bottom level anywhere within your agent. Specifically, this component makes use of the [Equation delegate](#) in order to allow you to easily create your own, custom equation. The signature for this delegate is:

```
public delegate void Equation
    (ActivationCollection input, ActivationCollection output);
```

As was the case for the other basic customizations (discussed previously), to implement a custom equation, all you need to do is define a method within your own code that conforms to the above signature and the Clarion Library will then be able to use that method to set the activations for the “nodes” on the output layer (given the specified input).

For example, let’s suppose that we want to create a [GenericEquation](#) that can be used to solve a simple linear equation (i.e., $Y = X$). We can accomplish this using the following method:

```
public void LinearEquation
    (ActivationCollection input, ActivationCollection output)
{
    output["Variable", "Y"] = input["Variable", "X"];
}
```

You should note that the [GenericEquation](#) class conforms to the library’s standard convention of transforming/bounding activations between 0 and 1. However, you can specify your own range for your equation by simply changing `Parameters.MIN_ACTIVATION` and `Parameters.MAX_ACTIVATION` (found by using the `Parameters` property in the [GenericEquation](#) class). By setting these parameters, both the input and output that is passed to your [delegate](#) method will be automatically transformed/bounded to between your specified range.

After we have created our [delegate](#) method, all we need to do in order to use it is provide it as a parameter during the initialization of a [GenericEquation](#). For example, suppose we wanted to use the simple linear equation in the bottom level of the ACS for our agent, John. This could be accomplished as follows:

⁶ Located in the *Clarion.Framework.Extensions* namespace

```
GenericEquation eq = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, GenericEquation.Factory, (Equation)LinearEquation);
```

As a reminder, in order to complete the initialization of our `GenericEquation`, we also need to define the inputs and outputs, as well as “commit” the component to the agent. Taken together, the following code demonstrates how we might setup and initialize the simple linear equation (with a range between ± 10) in our agent, John:

```
public void InitializeAgent()
{
    DimensionValuePair x = World.NewDimensionValuePair("Variables", "X");
    DimensionValuePair y = World.NewDimensionValuePair("Variables", "Y");
    Agent John = World.NewAgent("John");
    ... //Elided additional agent initialization
    GenericEquation eq = AgentInitializer.InitializeImplicitDecisionNetwork
        (John, GenericEquation.Factory, (Equation)LinearEquation);

    eq.Input.Add(x);
    eq.Output.Add(y);

    eq.Parameters.MIN_ACTIVATION = -10;
    eq.Parameters.MAX_ACTIVATION = 10;

    John.Commit(eq);
}
... //Elided code for running the task
public void LinearEquation
    (ActivationCollection input, ActivationCollection output)
{
    output["Variable", "Y"] = input["Variable", "X"];
}
```

Using the `GenericEquation` can save valuable time while in the debugging and tweaking phases of developing your task. However, keep in mind that ultimately you will want to replace these “shortcuts” with the actual pre-trained bottom level constructs that are defined within the Clarion theory (e.g., a backpropagation neural network, or `BPNetwork`). Note that, in order to help simplify these sorts of pre-training processes, the Clarion Library provides a very useful tool, called the `ImplicitComponentInitializer`⁷. Details on how to use the `ImplicitComponentInitializer` can be found in the *Useful Features* tutorial (located in the *Features & Plugins* section of the *Tutorials* folder).

Remember, 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.

⁷ Located in the main *Clarion* namespace

Useful Features

© 2013. Nicholas Wilson

Table of Contents

Viewing an Agent’s “Internals”	1
Logging (using Trace)	2
The Implicit Component Initializer	3
Pre-Training	4
Auto-Encoding	7
Populating the Input and Output Layers of an Implicit Component	9
Timing	10
Response Time	10
“Real-time” Mode	12
Asynchronous Operation	12

Viewing an Agent’s “Internals”

Reporting the outcome of a task usually involves at least some amount of investigation into what, exactly, the agent learned. We can retrieve any of the internal (functional) objects that are contained within an agent using the `GetInternals` method located in the `Agent` class. For example, suppose your task used bottom-up learning. In this case, we will likely want to know what rules were learned. The code below demonstrates how we could access the rules in the action rule store for our agent, John:

```
foreach (var i in John.GetInternals(Agent.Internals.ACTION_RULES))  
    Console.WriteLine(i);
```

The `GetInternals` method takes, as its input, the `InternalContainers` enumerator. This enumerator lists all of the functional internal objects that are available for retrieval from within an `Agent`. This includes:

- Drives
- Action rules
- Implicit decision networks
- Associative rules
- Associative memory networks
- Associative episodic memory networks

- Meta cognitive modules¹

The previous example (above) iterated through the rules that were returned by the `GetInternals` method and wrote them out to the console. Below is an example of a possible output from that code²:

Condition:

```
(Dimension = GoalChunk, value = Salute), Setting = True
(Dimension = GoalChunk, value = Bid Farewell), Setting = True
(Dimension = Salutation, value = Hello), Setting = True
(Dimension = Salutation, value = Goodbye), Setting = False
```

Action:

```
ExternalActionChunk Hello:
  DimensionValuePairs:
    (Dimension = SemanticLabel, value = Hello)
```

The `GetInternals` method is not only useful for outputting the agent's internal (functional) objects. It also provides an easy way to retrieve these objects from an agent for the sake of performing other tasks (such as pre-training or "offline" training, parameter tuning, etc.).

While you will likely find this feature to be very useful, you will probably also find that you want to know more about the inter-workings of the internal processes being performed by your agent (for the sake of tracing or debugging your simulation). The next section (below) covers the logging features of the Clarion Library.

Logging (using Trace)

Often times, when in the process of building and tuning a simulation, you may find it useful to view the internal processes of your agent(s). For example, by adjusting the generalization and specialization thresholds, you can increase/decrease the rate at which your agent performs either type of refinement. However, you obviously need some way to determine these metrics in order to appropriately tune the parameters.

To address those situations where you may want to trace the internal processes of the system, the Clarion Library provides several different levels of logging by leveraging C#'s [tracing mechanisms](#). By default, the logging level is set to "Off" (i.e., logging is not performed), however, you can specify a logging level by setting the

¹ Meta-cognitive modules have the `MetaCognitiveDecisionNetworks` and `MetaCognitiveActionRules` properties that you can use to view the internal components for those modules

² The rule output was taken from a run of the "HelloWorld-Full.cs" simulation, which is located in the "Beginner" folder under the "Samples" folder in the Clarion Library package

`LogLevel` property located in the `World` singleton. The following code demonstrates how this might be done:

```
World.LogLevel = TraceLevel.Warning;  
  
//Elided additional initialization of the simulating environment and agent(s)
```

The Clarion Library uses the default tracing levels that are defined by the `TraceLevel` enumerator³. Below is a breakdown of the kinds of things that are logged by the system at the various trace levels:

- **Off** – No logging is performed
- **Error** – Only a few, “abnormal”, system behaviors are logged
- **Warning** – The system will “warn” you when certain, basic, events occur (e.g., the agent chooses an action, the goal structure or working memory are updated, rules are added or deleted from the top level of the ACS, etc.)
- **Info** – Similar to the **Warning** level, except it provides more detailed information. For example, it will inform you of when:
 - Certain mechanisms (such as the drive strengths, goal structure, or working memory updating threads) begin/end their processes
 - The ACS (or MCS) determines that a rule should be extracted/generalized/specialized/deleted
 - Components are eligible and/or used during decision-making (in the ACS or MCS)
 - A certain rule/action type is targeted during decision-making
 - Etc.
- **Verbose** – The most detailed logging level. The system will provide very detailed information about all of the internal mechanisms within the system (e.g., the state of all processes, i.e., threads; all events; anything specified by the lower logging levels, etc.)

The Implicit Component Initializer

One of the most difficult parts of initializing a Clarion-based agent is the process of setting up and pre-training implicit components such as neural networks. To address this issue, the `ImplicitComponentInitializer` has been provided to assist you with this process. The initializer can be used with any functional object that extends from the `ImplicitComponent` class.

In the sections that follow, we will go over the various features that are at your disposal when using the `ImplicitComponentInitializer`.

³ Located in the `System.Diagnostics` namespace

Pre-Training

When in the course of developing a task, you may occasionally find that you need to “pre-ordain” your agent (or part of your agent) with a capability that is either not appropriate for, or simply can not be learned using standard “online” learning techniques (such as reinforcement learning). While this sort of “online” learning may not be a necessary aspect of your task, you may still want the bottom level of one or more subsystems to be “imbued” with some sort of proceduralized or “automated” functionality (e.g., capabilities or knowledge that are the consequence of an evolutionary process or the result of past experiences). For these types of instances, the Clarion Library provides a very useful feature known as the [ImplicitComponentInitializer](#).

Before we begin, in order to pre-train an [ImplicitComponent](#), we will first need two things:

1. A target that is to be trained⁴
2. A trainer to provide the correct (or “desired”) output to the target

The trainer can really be any type of [ImplicitComponent](#) (e.g., an equation, a table lookup, another previously trained [ImplicitComponent](#), etc.). However, the one thing you **MUST** be sure of is that it can provide the **correct** output(s) for all of the training data sets that are being using to train the target.

Note that, by default, the Clarion Library provides several built-in “extension” components⁵ that can very be easily be designated as trainers (i.e., without needing any training themselves). The simplest of these extensions is the [GenericEquation](#)⁶. In the demonstration that follows, this component will be used in the role of trainer for a [BPNetwork](#), that will act as the target. By using the `Train` method of the [ImplicitComponentInitializer](#), the target component will learn how to report the value of a simple linear equation (i.e., $Y = X$), as specified by the [GenericEquation](#).

To begin, we need to setup and initialize both our target and trainer. Let’s assume that we want to use this target in the bottom level of the ACS. The steps needed in order to accomplish this have already been discussed elsewhere⁷, so we will not go into those details here. Instead, let’s move on to our trainer.

Initializing a trainer works slightly differently than the standard method for initializing an agent’s internal functional objects. Specifically, trainer components are not required to exists within an agent. You can, of course, use a component that is within an agent if you so choose. However, as is more typically the case, you will likely just want to initialize your trainer externally for the sole purpose of training your target. To accomplish this, we use the `InitializeTrainer` method of the

⁴ The target component **MUST** implement the [ITrainable](#) interface

⁵ See the *Clarion.Framework.Extensions* namespace

⁶ See the *Basic Customization* tutorial (located in the *Customizations* section of the *Tutorials* folder) for details about this component

⁷ See the *Setting Up and Using the ACS* tutorial (located in the *Basics Tutorials* section)

`ImplicitComponentInitializer`. This method is essentially the same as the `Initialize` methods in the `AgentInitializer`, except that it does not “tie” the initialized component to an agent. The following code demonstrates how the `InitializeTrainer` method might be used to initialize a `GenericEquation`:

```
GenericEquation eq = ImplicitComponentInitializer.InitializeTrainer
    (GenericEquation.Factory, (Equation)LinearEquation);
```

A few things should be noted at this point. First, to initialize a `GenericEquation`, we must specify a `delegate` method (e.g., `LinearEquation`), which conforms to the `Equation` signature. This method is used by the component in order to calculate the activations for the “nodes” on the output layer.⁸ Second, an `ImplicitComponent`, initialized using this method, can **ONLY** be used as a trainer. In other words, it is not possible to later use this component as an internal functional object within an agent. Third, like any component that is initialized using the `AgentInitializer`, components initialized using the `ImplicitComponentInitializer` **MUST** be “committed” before they can be used. However, unlike how an agent’s internals are committed, to commit a trainer, we will need to call that component’s own `Commit()` method.

The following code demonstrates how we might initialize both a `BPNetwork` (target), in the bottom level of the ACS of our agent, John, as well as initialize a `GenericEquation` (trainer):

```
DimensionValuePair x = World.NewDimensionValuePair("Variables", "X");
DimensionValuePair y = World.NewDimensionValuePair("Variables", "Y");
Agent John = World.NewAgent("John");

//Elided Agent Initialization

BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, BPNetwork.Factory);

GenericEquation eq = ImplicitComponentInitializer.InitializeTrainer
    (GenericEquation.Factory, (Equation)LinearEquation);

net.Input.Add(x);
eq.Input.Add(x);

net.Output.Add(y);
eq.Output.Add(y);

John.Commit(net);
eq.Commit();
```

After we have initialized our trainer and target, the next thing that we need to do is setup training data sets. Each training data set should specify a different configuration for the activations of input layer of our trainer and target components. There are two options, however, for specifying training data sets. The most

⁸ See the *Basic Customization* tutorial for more details

straightforward method is to simply create a bunch of data sets using fixed input activation patterns. This is the preferred method when you are working with a very specific, known, set of training data. The other method that is available to you is to define a **range** for each input node (or a subset thereof), between which training should occur (and at a specified increment).

To define a **range**, for a given node on the input layer, we will use the `AddRange` method in the `ImplicitComponentInitializer`. As part of calling this method, we will need to specify the following:

- The `IWorldObject` associated with the input node
- The upper and lower bounds for the range
- The increment at which the range should be traversed (optional)⁹

Below is an example of how we would define a range (between 0 and 1, with an increment of 0.1) for our variable “X”:

```
ImplicitComponentInitializer.AddRange(x, 0, 1, .1);
```

Note that if a range has been specified for a particular input node, it will be used for that node, irrespective of if a data set specifies a fixed value for that node. Keep this in mind in case you are using the `ImplicitComponentInitializer` to train multiple networks as you may want to remove one or more ranges (by calling the `RemoveRange` method) between training operations.

You **MUST** create at least one training data set, even if you have defined **ranges** for all of your input nodes. The `NewDataSet` method in the `ImplicitComponentInitializer` can be used to generate new data sets. Each data set is represented as an `ActivationCollection`. You will notice very quickly, while creating and using data sets, that they work essentially the same as `SensoryInformation`. This is because `ActivationCollection` is actually the base class for `SensoryInformation`. The following example demonstrates how to setup a single data set for our variable “X”:

```
List<ActivationCollection> dataSets = new List<ActivationCollection>();  
  
dataSets.Add(ImplicitComponentInitializer.NewDataSet());  
dataSets[0].Add(x);
```

Recall that we specified a range for our variable “X”, so we do not need to worry about specifying an activation. However, if we hadn’t specified a range for “X”, then specifying an activation for “X” would be **exactly** the same process as is normally done for `SensoryInformation` objects.

At this point, we are now ready to begin training. The `Train` method takes the following as inputs:

- The target

⁹ The default increment is 0.01

- The trainer
- The data sets (as a collection of [ActivationCollection](#) objects)
- A termination condition (optional, FIXED or SUM_SQ_ERROR)
- The number of times over which the data sets should be iterated (optional, if the FIXED termination condition is used)
- The threshold under which the sum of squared error must fall (optional, if the SUM_SQ_ERROR termination condition is used)
- The selection temperature (optional, if the target is [IReinforcementTrainable](#))
- Whether the data sets should be traversed in random order (optional)
- Whether the call to the method is intended **only** to **test** the performance of the target given the data set (optional)

In order to initiate training on our target [BPNetwork](#), using the [GenericEquation](#) trainer (with SUM_SQ_ERROR termination condition and the default threshold), we need to do the following:

```
ImplicitComponentInitializer.Train(net, eq, dataSets,
    ImplicitComponentInitializer.TrainingTerminationConditions.SUM_SQ_ERROR);
```

When the above code returns, the [BPNetwork](#) will be fully trained to report the outputs (as specified by the [GenericEquation](#)) for the training data set (i.e., the range of values that we defined for “X”).

As a final note, you can follow the status of the training operation simply by enabling the Clarion Library’s built-in logging feature (see the section above). In addition, the [ImplicitComponentInitializer](#) can also be serialized, thus allowing you to save and reload your range specifications.

Auto-Encoding

One key feature of Clarion is the use of implicit “auto-encoder” components (e.g., Hopfield networks¹⁰) in the bottom level of the NACS to help facilitate associative reasoning. These networks can be used to enable some of Clarion’s more unique reasoning capabilities (especially with regard to the synergy of rule-based and associative reasoning processes).

With the above being said, encoding knowledge into these sorts of components can sometimes be a bit tricky. Therefore, in order to assist you with this process, the [ImplicitComponentInitializer](#) also provides an Encode method, which is specifically designed to “train” implicit components that implement the [IAutoEncoder](#) interface.

Currently, the [HopfieldNetwork](#) class is the primary “auto-encoder” that comes pre-packaged in the Clarion Library. Therefore, it will be used for our demonstrations on how to Encode using the [ImplicitComponentInitializer](#).

¹⁰ See the Clarion-H addendum to the technical specification document (located [here](#))

The steps for initializing a target `ImplicitComponent` for encoding is actually very similar to what was described in the previous section for training. In fact, in many ways, the two processes are essentially the same. However, the main place in which they differ is that encoding does not require a “trainer.” Instead, we simply need to specify the data sets that are being encoded into the auto-encoder, and the `ImplicitComponentInitializer` will handle the rest.

Similar to the `Train` method, the `Encode` method takes the following inputs:

- The target auto-encoder
- The data sets
- A termination condition (optional, `FIXED` or `UNTIL_ENCODED`)
- The number of times over which the data sets should be iterated (optional, if the `FIXED` termination condition is used)
- Whether the data sets should be traversed in random order (optional)
- Whether the call to the method is intended **only** to **test** the retrieval accuracy of the target given the data set (optional)

When the `Encode` method returns, the target will be fully encoded and will be able to rebuild any of the patterns in the data sets. In the code example below, we demonstrate how you could setup a `HopfieldNetwork` (in the bottom level of the NACS) with 10 nodes and use the `ImplicitComponentInitializer` to encode 3 activation patterns for those nodes (using the default encoding options):

```
//Initialize the 10 nodes
DimensionValuePair n1 = World.NewDimensionValuePair("Node", 1);
DimensionValuePair n2 = World.NewDimensionValuePair("Node", 2);
DimensionValuePair n3 = World.NewDimensionValuePair("Node", 3);
DimensionValuePair n4 = World.NewDimensionValuePair("Node", 4);
DimensionValuePair n5 = World.NewDimensionValuePair("Node", 5);
DimensionValuePair n6 = World.NewDimensionValuePair("Node", 6);
DimensionValuePair n7 = World.NewDimensionValuePair("Node", 7);
DimensionValuePair n8 = World.NewDimensionValuePair("Node", 8);
DimensionValuePair n9 = World.NewDimensionValuePair("Node", 9);
DimensionValuePair n10 = World.NewDimensionValuePair("Node", 10);

Agent John = World.NewAgent("John");

//Elided other agent initializations

HopfieldNetwork net = AgentInitializer.InitializeAssociativeMemoryNetwork
    (John, HopfieldNetwork.Factory);

//Add the 10 nodes to the Hopfield network
net.Nodes.Add(n1);
net.Nodes.Add(n2);
net.Nodes.Add(n3);
net.Nodes.Add(n4);
net.Nodes.Add(n5);
net.Nodes.Add(n6);
net.Nodes.Add(n7);
net.Nodes.Add(n8);
net.Nodes.Add(n9);
```



```

net.Nodes.Add(n10);

//Don't forget to commit the network!
John.Commit(net);

ActivationCollection[] patterns = new ActivationCollection[3];

//Pattern 1
patterns[0] = ImplicitComponentInitializer.NewDataSet();
patterns[0].Add(n1, 1);
patterns[0].Add(n2, 0);
patterns[0].Add(n3, 1);
patterns[0].Add(n4, 0);
patterns[0].Add(n5, 1);
patterns[0].Add(n6, 0);
patterns[0].Add(n7, 1);
patterns[0].Add(n8, 0);
patterns[0].Add(n9, 1);
patterns[0].Add(n10, 0);

//Pattern 2
patterns[1] = ImplicitComponentInitializer.NewDataSet();
patterns[1].Add(n1, 1);
patterns[1].Add(n2, 1);
patterns[1].Add(n3, 0);
patterns[1].Add(n4, 0);
patterns[1].Add(n5, 1);
patterns[1].Add(n6, 1);
patterns[1].Add(n7, 0);
patterns[1].Add(n8, 0);
patterns[1].Add(n9, 1);
patterns[1].Add(n10, 1);

//Pattern 3
patterns[2] = ImplicitComponentInitializer.NewDataSet();
patterns[2].Add(n1, 0);
patterns[2].Add(n2, 0);
patterns[2].Add(n3, 0);
patterns[2].Add(n4, 1);
patterns[2].Add(n5, 1);
patterns[2].Add(n6, 1);
patterns[2].Add(n7, 0);
patterns[2].Add(n8, 0);
patterns[2].Add(n9, 0);
patterns[2].Add(n10, 1);

ImplicitComponentInitializer.Encode(net, patterns);

```

Populating the Input and Output Layers of an Implicit Component

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Timing

You may have noticed by this point that timing is an important feature in the Clarion library. The system uses time stamps to track everything from the interactions between the various subsystems and modules to response times. In general these time stamps can be accessed using a (somewhat ubiquitous) property called `TimeStamp`. In the sections that follow, we will look at two particular features that use this timing and that may be useful for your Clarion-based project.

Response Time

Many tasks, especially those which aim to explore human cognitive phenomenon, place importance on the role of response times as a measurement for performance. Therefore, as part of the process of action decision-making, the Clarion library also keeps track of various timings. These timings include:

- Perception time
 - Includes drive strength updating operations in the MS and any “pre-action selection” operations (e.g., goal setting) in the MCS
- Decision time
 - Varies based on the level selection method that is used in the ACS
- Actuation time
- Reasoning time

Each of the above timings are tunable via parameters. These parameters can be located as follows:

```
//Perception Time
Agent.GlobalParameters.PERCEPTION_TIME
John.Parameters.PERCEPTION_TIME

//Actuation Time
Agent.GlobalParameters.ACTUATION_TIME
John.Parameters.ACTUATION_TIME

//Decision Time
ActionCenteredSubsystem.GlobalParameters.TOP_LEVEL_DECISION_TIME;
John.ACS.Parameters.TOP_LEVEL_DECISION_TIME

ActionCenteredSubsystem.GlobalParameters.BOTTOM_LEVEL_DECISION_TIME;
John.ACS.Parameters.BOTTOM_LEVEL_DECISION_TIME

//Reasoning Time
NonActionCenteredSubsystem.GlobalParameters.REASONING_ITERATION_TIME;
John.NACS.Parameters.REASONING_ITERATION_TIME
```

Every time an agent perceives, the action that results from that perception will also be accompanied by a response time. In general, this response time is determined by:

$$ResponseTime = PerceptionTime + DecisionTime + ActuationTime$$

If your simulation is being run in “asynchronous mode”, this response time will be provided to you by default as part of the parameters passed into the `ProcessChosenExternalAction` method. Otherwise, to get the response time related to a particular perception you need to do the following:

```
long rt = John.GetResponseTime(si);
```

Note that the system will only hold onto the response time data for so long (as determined by the `PREVIOUS_RT_CAPACITY` and `LOCAL_EPISODIC_MEMORY_RETENTION_THRESHOLD` parameters, located in the `Agent` and `ActionCenteredSubsystem` classes respectively), so this method should typically be called immediately after the `GetChosenExternalAction` method is called.

The response time represents the time that it took the agent to perform a particular action (which was initiated by perceiving a sensory information object). Conceptually, the response time .

In fact, the system already uses these response times in a variety of ways:

- The `MAX_RESPONSE_TIME` is used by default to increment the time stamp when a new `SensoryInformation` object is created using `World.NewSensoryInformation...`
 - Note that the actual response time can also (optionally) be used
- To determine when an action is delivered to the simulating environment
 - By adding the response time to the time stamp of the affiliated perception

By default, response times can tend to look fairly one-dimensional since the calculation of these times are based on parameters that don't vary (except, of course, for decision times). In reality, however, human response times fluctuate a bit, even when tasks are highly learned and proceduralized. Therefore, in the Clarion library, a set of parameters have been added to allow for variability in both the perception and actuation times. This variability can be added by simply changing the following parameters:

```
Agent.GlobalParameters.PERCEPTION_TIME_VARIABILITY_THRESHOLD  
John.Parameters.PERCEPTION_TIME_VARIABILITY_THRESHOLD
```

```
Agent.GlobalParameters.ACTUATION_TIME_VARIABILITY_THRESHOLD  
John.Parameters.ACTUATION_TIME_VARIABILITY_THRESHOLD
```

Once set, the perception and actuation times will vary $\pm threshold$. The distribution of this variation is normalized by default. However, if you prefer to use a different

distribution, you can also specify your own custom variability calculator using the following delegate:

```
delegate long ResponseTimeVariabilityCalculator(long defaultTime, double threshold);
```

Methods written using this delegate signature can be used by an agent to calculate either the perception or the actuation time variability. Feel free to use the same method for either timing, as the `defaultTime` and `threshold` parameters will be specific to each timing. However, note that you can also specify different distributions for these timings if you so desire.

If you do choose to write your own variability calculator, be aware that you must set it for the agent in whom it is to be used. This is done by calling either of the following properties:

```
John.PerceptionTimeVariabilityCalculator =  
    (ResponseTimeVariabilityCalculator)SomeCalculatorMethod;  
John.ActuationTimeVariabilityCalculator =  
    (ResponseTimeVariabilityCalculator)SomeCalculatorMethod;
```

“Real-time” Mode

As was mentioned earlier, timing in the Clarion library is everything. However, the system keeps track its own time stamps, which are not actually correlated to the speed at which the system runs. In other words, it has no basis on the time within the real world. By default, the system will simply run as quickly as it possibly can. This is likely preferred for most tasks, as it would be very laborious if you had to wait for more than 1 second each time your agent perceived something.

That being said, you may find cases where you will want your agent to operate in “real-time” (e.g., when actually interacting with the real world). In these cases, agents can be forced to take the amount of time they are supposed to take when performing various operations. For example, if perception time takes 200 milliseconds, then an agent can be made to will wait that amount of time before it starts performing decision-making; if actuation time takes 500 milliseconds, then an agent will wait that long before delivering an action to the outside world; and so on and so forth.

Turning on “real-time” mode is as simple as flipping a switch. Below is an example of how we can specify that our agent, John, should run in real-time mode:

```
John.Parameters.IN_REAL_TIME = true;
```

Asynchronous Operation

As we have mentioned a couple of times throughout these tutorials, the Clarion Library is an asynchronous (i.e., multi-threaded) system that leverages an advanced publisher/subscriber event model in order to facilitate the interactions between all

of the internal mechanisms within an agent. Because of this fact, the Clarion Library can also be setup to interact asynchronously with a simulating environment.

Setting up the simulating environment to interact asynchronously with agents is actually a fairly simple process. All we need to do is extend the abstract `AsynchronousSimulatingEnvironment`¹¹ class:

```
public class SomeSimulatingEnvironment : AsynchronousSimulatingEnvironment
{
    ... //Elided simulation code
}
```

After we have extended this class, there are only two other things that need to be done:

1. Override the abstract `ProcessChosenExternalAction` method (which is defined by the `AsynchronousSimulatingEnvironment` class)
2. Register the asynchronous simulating environment with the agent (by calling the agent's `RegisterAsynchronousSimulatingEnvironment` method)

The follow code sample demonstrates how we might accomplish the above for our agent, John:

```
// Register the simulating environment in the initialization section
John.RegisterAsynchronousSimulatingEnvironment(this);

...

protected override void ProcessChosenExternalAction(Agent actor,
    ExternalActionChunk chosenAction, SensoryInformation relatedSI,
    Dictionary<ActionChunk, double> finalActionActivations, long performedAt,
    long responseTime)
{
    ... //Elided code to process the agent's chosen action and deliver feedback
}
```

Remember, 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.

¹¹ Located in the *Clarion.Plugins* namespace

Setting Up & Using the NACS

© 2013. Nicholas Wilson

Table of Contents

A Brief Note	1
Setting Up & Performing Reasoning	1
A Walk-through of the “Simple Reasoner” Task	2
Distributed Dimension-Value Pairs.....	3
Adding Knowledge to the GKS.....	4
Initializing Associative Memory Networks.....	4
Initializing Associative Rules.....	6
Performing Reasoning.....	6
Setting Up & Using Episodic Memory	9
Creating Episodes	9
Initializing Associative Episodic Memory Networks	9
Generating New Knowledge and Associative Rules	9
Performing “Offline” Learning	10

A Brief Note

Before you get started using the NACS you should note that the Clarion Library provides two methods for interacting with the NACS. The simpler method (i.e., as a stand-alone mechanism) is outlined herein. The other method (i.e., integrated with and initiated by the ACS and/or MCS¹) can be found in the “*Advanced ACS Setup*” tutorial.² However, keep in mind that you should still read this tutorial first before attempting to use the NACS via the integrated method. At the very least, this tutorial will teach you how to initialize the top and bottom levels of the NACS. Furthermore, you may find that the stand-alone method is very useful for testing whether the NACS is operating correctly before moving onto the somewhat more complicated matter of integrating the NACS with the other subsystems.

Setting Up & Performing Reasoning

In this section we will go over an example of how to set up and run a task using the NACS’s reasoning mechanism. If you are interested in following along, the specific example through which we will be walking is called “*Reasoner – Simple.cs*” and it can be found in the *Advanced* section of the *Samples* folder.

The “simple reasoner” simulation sample was designed with the same objective in mind as the “simple hello world” task. That is, its primary purpose is to provide a

¹ As is specified by the Clarion theory

² In the *Advanced* section of the *Tutorials* folder

simple introduction to the NACS. The specifics of the task, themselves, are not particularly interesting, nor were they intended to be. Instead, this task is simply meant to clearly demonstrate how to correctly setup, train, and use the various aspects of the NACS's reasoning mechanism. So let's begin our walk-through:

A Walk-through of the "Simple Reasoner" Task

The first thing you need to know are the necessary namespaces. As is normally the case, the primary classes you will use are located in either the `Clarion` or `Clarion.Framework` namespaces:

```
using Clarion;  
using Clarion.Framework;
```

With this point out of the way, let's move on to the `Main` method:

```
public static void Main()  
{  
    Agent reasoner = World.NewAgent();  
  
    InitializeWorld(reasoner);  
  
    foreach (DeclarativeChunk dc in chunks)  
        reasoner.AddKnowledge(dc);  
  
    HopfieldNetwork net = AgentInitializer.InitializeAssociativeMemoryNetwork  
        (reasoner, HopfieldNetwork.Factory);  
  
    net.Nodes.AddRange(dvs);  
  
    reasoner.Commit(net);  
  
    EncodeHopfieldNetwork(net);  
  
    SetupRules(reasoner);  
  
    reasoner.NACS.Parameters.REASONING_ITERATION_COUNT = 2;  
    reasoner.NACS.Parameters.CONCLUSION_THRESHOLD = 1;  
  
    DoReasoning(reasoner);  
  
    reasoner.Die();  
  
    Console.WriteLine("Press any key to exit");  
    Console.ReadKey();  
}
```

Most of the interesting details of this task are actually in other methods that are called by the `Main` method. However, you may notice a few unfamiliar things in the above code. First, note the following line:

```
InitializeWorld();
```

For our “simple reasoner” task, we begin by initializing the `World` with 30 dimension-value pairs and 5 unique declarative “pattern” chunks. These chunks are manually specified by the following:

```
static int [][] patterns =
{
    new int [] {1, 3, 5, 11, 13, 16, 19, 23, 27},
    new int [] {3, 6, 7, 8, 12, 15, 20, 21, 26},
    new int [] {2, 4, 8, 9, 11, 17, 18, 24, 30},
    new int [] {1, 4, 10, 12, 15, 17, 19, 22, 29},
    new int [] {3, 5, 8, 10, 14, 18, 20, 25, 28}
};
```

Each of the sub arrays (located in the 2nd dimension of the above 2-dimensional array) specifies a different activation pattern for the 30 dimension-value pairs. The “value” of each dimension-value pair is actually numbered, and the integers in the above patterns are associated with these values. As mentioned previously, we will also need to create a `DeclarativeChunk` for each of these patterns. The following `World` initialization code demonstrates how we can accomplish this:

```
static void InitializeWorld(Agent a)
{
    for (int i = 1; i <= nodeCount; i++)
    {
        dvs.Add(World.NewDistributedDimensionValuePair(a, i));
    }

    for (int i = 0; i < patterns.Length; i++)
    {
        DeclarativeChunk dc =
            World.NewDeclarativeChunk(i, addSemanticLabel:false);

        foreach (var dv in dvs)
        {
            if (patterns[i].Contains(dv.Value))
            {
                dc.Add(dv);
            }
        }

        chunks.Add(dc);
    }
}
```

Note that the “dvs” and “chunks” collections in the above code are simply used to track our dimension-value pairs and declarative chunks between the different “phases” (or methods) of the task. As has been discussed in previous tutorials, this is generally a useful thing to do within any simulating environment, as it saves the additional overhead of using the “`World.Get...`” methods.

Distributed Dimension-Value Pairs

In the previous code sample, you may have noticed that we called the `NewDistributedDimensionValuePair` method instead

`NewDimensionValuePair`. This has been done in order to introduce you to a new (as of 6.1.1) feature of the Clarion library: `DistributedDimensionValuePair`. The concept behind distributed dimension-value pairs was first introduced by Helie & Sun (2010). The idea here is that more semantic representations (such as chunks) can be translated into more “distributed” (i.e., neural-like) representations in the bottom level of Clarion. In other words, instead of having the nodes of a network be tied to dimension-value pair that has inherent semantic meaning, collections of somewhat arbitrarily defined nodes could instead be utilized to represent semantic concepts more in a more sub-symbolic way (see Helie & Sun, 2010 for more details). As a matter of implementation, this idea has been actualized in the form of `DistributedDimensionValuePair`.

Distributed dimension-value pairs are agent-specific and do not require a dimension be specified (just a value). Otherwise, once initialized, they can be utilized essentially like a normal dimension-value pair. When added to a chunk (and as nodes in an `ImplicitComponent`), these distributed dimension-value pairs provide a “sub-symbolic” featurized representation for the chunk itself.

Adding Knowledge to the GKS

Moving back to our discussion of the `Main` method, the next thing you may have noticed is the call to the `AddKnowledge` method (located in the `Agent` class):

```
foreach (DeclarativeChunk dc in chunks)
    reasoner.AddKnowledge(dc);
```

This method is used to add the declarative chunks³ into the GKS of our agent. Be aware that **ALL** chunks **MUST** be added to the GKS if they are to be used as part of reasoning. Besides the obvious theoretical consideration, we also need to do this for implementation-specific purposes. In particular, various aspects of the GKS’s backend are actually used to help facilitate the reasoning process.

You should also note here that chunks should **NEVER** be altered (say, by adding or removing a dimension-value pair) after they have been added to the GKS. Doing so will break the storage method that is used to store these chunks within the GKS. To relate this to a well-known concept from object-oriented programming, altering a chunk once it is in the GKS is essentially the same as changing the hash code of an `object` after it has been stored within a `HashMap`. In other words, **DON’T DO IT!**

Initializing Associative Memory Networks

Moving along with our walk through of the `Main` method, the next thing to notice is the following:

```
HopfieldNetwork net = AgentInitializer.InitializeAssociativeMemoryNetwork
    (reasoner, HopfieldNetwork.Factory);

net.Nodes.AddRange(dvs);
```

³ Technically, any type of `Chunk` can be added as “knowledge” into the GKS.

```
reasoner.Commit(net);
```

These lines are used to initialize a [HopfieldNetwork](#) in the bottom level of the NACS of our agent. Note that the [HopfieldNetwork](#) is a so called “auto-encoder”, and as such, is mainly used as an auto-associative memory network.⁴ Initializing a [HopfieldNetwork](#) is slightly different than initializing your standard “feed-forward” network. In particular, since the [HopfieldNetwork](#) is conceptualized as asynchronous (meaning it technically doesn’t have an input and output layer⁵), [IWorldObject](#) objects are actually just added to a general collection of “nodes” for this network instead of being specified as part of either the input or output layer.

Once our [HopfieldNetwork](#) is set up, we need to encode some knowledge into it. Auto-associative memory networks work by “reconstructing” encoded knowledge (or patterns) given a partial (or noisy) “input.” For our current task, we will want to encode the 5 patterns (i.e., the declarative chunks) that were discussed previously.

The `Encode` method, in the [ImplicitComponentInitializer](#), actually handles the majority of the encoding work.⁶ The only thing we need to do to use this method is specify the “data sets” that are being encoded. Also, you may wish to perform a separate “test” run to ensure that the data sets are correctly recalled.⁷ We can do this by simply calling the `Encode` method and specifying `true` for the “testOnly” parameter. Note that this would most often be done for cases where you wished to use a different `TRANSMISSION_OPTION` for the “encoding” and “testing” phases.

The following code, from the “simple reasoner” task, demonstrates how we might encode patterns into, and then “test” the recall accuracy of our [HopfieldNetwork](#):

```
static void EncodeHopfieldNetwork(HopfieldNetwork net)
{
    double accuracy = 0;

    do
    {
        net.Parameters.TRANSMISSION_OPTION =
            HopfieldNetwork.TransmissionOptions.N_SPINS;

        List<ActivationCollection> sis = new List<ActivationCollection>();
        foreach (DeclarativeChunk dc in chunks)
        {
```

⁴ The difference between auto-associative and hetero-associative networks is mainly conceptual. The bottom level of the NACS can actually store any combination of these two types of networks and both will function as expected according to their own purpose and capabilities.

⁵ As a matter of implementation, the [HopfieldNetwork](#) actually uses the non-asynchronous methodology (i.e., with equivalently configured input and output layers). However, all interactions have been purposely designed so that the network can be initialized using either conceptualization.

⁶ For more details on how to use this initializer, see the “*Useful Features*” tutorial (located in the “*Features & Plugins*” section of the “*Tutorials*” folder).

⁷ Although the `Encode` method actually performs this step automatically, if the default `UNTIL_ENCODED` option is used.

```

        ActivationCollection si = ImplicitComponentInitializer.NewDataSet();
        si.AddRange(dc, 1);
        sis.Add(si);
    }

    ImplicitComponentInitializer.Encode(net, sis);

    net.Parameters.TRANSMISSION_OPTION =
        HopfieldNetwork.TransmissionOptions.LET_SETTLE;

    accuracy = ImplicitComponentInitializer.Encode(net, sis, testOnly: true);
} while (accuracy < 1);
}

```

After we have encoded knowledge into the bottom level of the NACS, the next thing we need to do is generate and add associative rules to the top level.

Initializing Associative Rules

The process for initializing associative rules in the top level of the NACS is very similar to the process used to add action rules to the top level of the ACS. For our “simple reasoner” task, we want to set up 5 rules, with the following convention:

If pattern X, then conclude pattern X + 1

For example, if the input to the top level is the `DeclarativeChunk` representing pattern 1, then the top level should conclude the `DeclarativeChunk` representing pattern 2. The following code demonstrates how we would set up these sorts of associative rules in the top level of the NACS:

```

static void SetupRules(Agent reasoner)
{
    for (int i = 0; i < chunks.Count - 1; i++)
    {
        RefineableAssociativeRule ar =
            AgentInitializer.InitializeAssociativeRule(reasoner,
                RefineableAssociativeRule.Factory, chunks[i + 1]);

        ar.GeneralizedCondition.Add(chunks[i], true);

        reasoner.Commit(ar);
    }
}

```

Performing Reasoning

The last thing we may want to do before we initiate the reasoning process is to set any (optional) reasoning parameters. For our current task, we will need to set the following parameters:

```
reasoner.NACS.Parameters.REASONING_ITERATION_COUNT = 2;

reasoner.NACS.Parameters.CONCLUSION_THRESHOLD = 1;
```

The first parameter specifies that the NACS should perform 2 reasoning iterations before return its conclusions. The second parameter indicates that we only want those “fully activated” conclusions to be returned. There are many other reasoning parameters that can be set, and which will alter the behavior of the reasoning mechanism. For more information on them, see the “auto generated” documentation⁸ for the [NonActionCenteredSubsystemParameters](#) class.

At this point, though, we should now be ready to start reasoning. Note that reasoning is currently only operational as a stand-alone mechanism. Future versions of the Clarion Library will provide a more natural integration into the overall system. However, as this integration is currently under development, to use the NACS’s reasoning mechanism, you will need to call the `PerformReasoning` method (found in the NACS of an agent) and specify the “input” that is being used to initiate this reasoning:

```
var o = reasoner.NACS.PerformReasoning(si);
```

The `PerformReasoning` method will return the conclusion(s) from reasoning in the form of a collection [ChunkTuple](#) objects. The [ChunkTuple](#) is essentially just a “wrapper” for a conclusion [Chunk](#) and its associated activation (which specifies the “support” for that conclusion). For our “simple reasoner” task, we use a partial (noisy) reconstruction of each pattern as inputs (into 5 different rounds of reasoning). These “noisy” patterns are created by “zeroing-out” a percentage of each pattern. For example, with a noise value of .4, the final 40% of the input will have nothing but 0 activations.

The following code demonstrates how, for our current example, we might set up input patterns, initiate reasoning, and process the conclusions:

```
static void DoReasoning(Agent reasoner)
{
    int correct = 0;

    foreach (DeclarativeChunk dc in chunks)
    {
        ActivationCollection si = ImplicitComponentInitializer.NewDataSet();

        int count = 0;

        foreach (DimensionValuePair dv in dvs)
        {
            if (((double)count / (double)dc.Count < (1 - noise)))
            {
                if (dc.Contains(dv))
                {
```

⁸ Located in the “Documentation” folder.

```

        si.Add(dv, 1);
        ++count;
    }
    else
        si.Add(dv, 0);
}
else
    si.Add(dv, 0);
}

Console.WriteLine("Input to reasoner:\r\n" + si);

Console.WriteLine("Output from reasoner:");

var o = reasoner.NACS.PerformReasoning(si);

foreach (var i in o)
{
    Console.WriteLine(i.CHUNK);
    if (i.CHUNK == dc)
        correct++;
}
}
Console.WriteLine("Retrieval Accuracy: " +
    (int)((((double)correct / (double)chunks.Count) * 100) + "%");
}

```

If everything is working correctly, we should see the following behavior:

- 1st iteration = the bottom level will complete the partial input pattern
- 2nd iteration = the top level will receive the conclusion associated with the “reconstructed pattern” from the bottom level and will conclude the following pattern
- Conclusions = the “conclusion chunks” from each reasoning iteration

For example, if the input is based on a “partial reconstruction” of pattern 1, the conclusions from reasoning should be the declarative chunks associated with patterns 1 and 2.

Finally, to complete our task, we will need to kill our agent (as always):

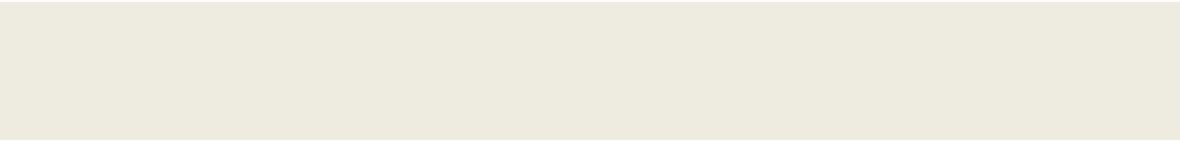
```
reasoner.Die();
```

This concludes our walk through to the “simple reasoner” task. At this point, you should have everything you need to get started on developing your own reasoning-specific tasks using the Clarion Library’s NACS. If you are interested, you can learn more about how to integrate the NACS with the ACS in the “*Advanced ACS Setup*” tutorial located in the *Advanced* section of the *Tutorials* folder.

Setting Up & Using Episodic Memory

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

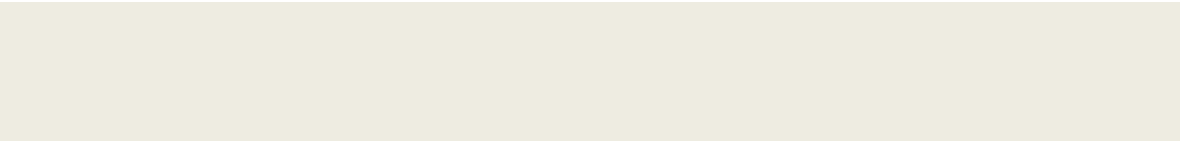
In future releases, this section will contain information about how to use this feature (when it becomes available).



Creating Episodes

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

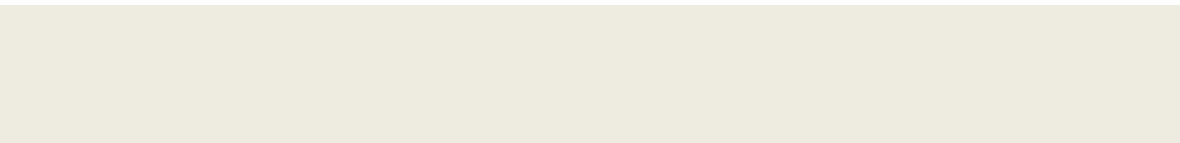
In future releases, this section will contain information about how to use this feature (when it becomes available).



Initializing Associative Episodic Memory Networks

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).



Generating New Knowledge and Associative Rules

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

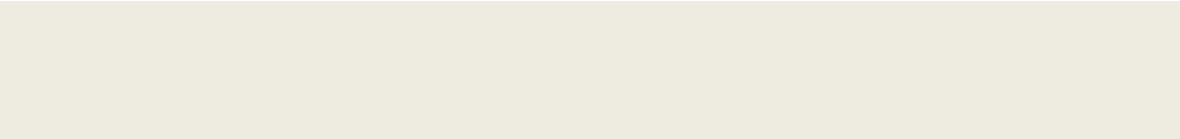
In future releases, this section will contain information about how to use this feature (when it becomes available).



Performing “Offline” Learning

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).



Remember, 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.

Advanced ACS Setup

© 2013. Nicholas Wilson

Table of Contents

A Brief Note	1
Interacting with the NACS	1
Making Reasoning Requests	2
Specifying Alternative Dimensions	5
Filtering Input/Conclusions	5
Retrieving Chunks from the GKS	6
Interacting with Episodic Memory	6
Retrieving Episodes.....	6
“Offline” Learning.....	7
Generative Actions	7
An Example: Using Generative Actions to Change Local Parameters	7

A Brief Note

The primary focus of this tutorial is to demonstrate how to interact with the NACS using the ACS (and by creative extension the MCS). However, there is another, simpler, method for using the NACS (i.e., as a stand-alone mechanism). That method is described in the “*Setting Up and Using the NACS*” tutorial (found in the *Advanced* folder of the *Tutorials* section).

Keep in mind that you should read that tutorial first before attempting to integrate the ACS and NACS. At the very least, it will show you how to initialize the top and bottom levels of the NACS. Furthermore, you may find that the stand-alone method is very useful for testing whether the NACS is operating correctly before moving onto the somewhat more complicated matter of integrating the NACS with the other subsystems.

Interacting with the NACS

As part of performing more advanced tasks, you may wish you agent to reason over its declarative knowledge about the world before (and in order to aid in) action decision making. To this end, the ACS (or MCS) and NACS have been designed to interact with one another. In the following sections we will look at an example of how the NACS and ACS can be setup to work in tandem on a simple reasoning-type task. In particular, we will be looking at the sample “*Reasoner – Full.cs*”, which can be found in the *Advanced* section of the *samples* folder. This example extends “*Reasoner – Simple.cs*” to integrate ACS and NACS functioning. If you have not

already done so, you will want to refer to the *Advanced* tutorial “*Setting Up & Using the NACS*” for an overview of this simple reasoning task as well as instructions on how to initiate the NACS.

The full reasoner task is similar to the simple reasoner, except that the specifics of the task itself are actually a bit simpler. In particular, the ACS is set up using only the top level whereas the NACS is set up with only the bottom level. The task itself entails the perceiving of one of 5 “noisy” patterns. This pattern triggers a call by the ACS to the NACS. The NACS then uses an auto-associative network on the bottom level to complete the pattern and returns a fully activated chunk back to the ACS (which represents the full pattern). That chunk is then stored in working memory. Finally, the agent is asked whether the initial input was the partial pattern for a particular target (i.e., the 2nd pattern). The expected output (from the agent) for this simulation is: *No, Yes, No, No, No*.

Making Reasoning Requests

Initiating the NACS reasoning using the ACS is as simple as setting a goal in the goal structure or a chunk in working memory. In other words, to have the ACS initiate reasoning, all you need to do is use a [ReasoningRequestActionChunk](#). The following code demonstrates how to initialize this action chunk:

```
ReasoningRequestActionChunk think =  
    World.NewReasoningRequestActionChunk ("DoReasoning");  
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.NEW, 1);
```

Note that the second line is similar to how we specify set/reset commands when using [GoalStructureUpdateActionChunk](#) or [WorkingMemoryUpdateActionChunk](#).¹ Specifically, we use the [RecognizedReasoningActions](#) to specify the type of reasoning request that is being made by the ACS. In our current example, we are specifying that the “think” action will prompt a new round of reasoning and perform 1 iteration. In total, the [ReasoningRequestActionChunk](#) recognizes the following actions:

- NEW – Specifies that a new round of reasoning is being requested
- CONTINUE – Similar to NEW, but specifies that reasoning should be continued from a previous round
- PEEK – Specifies that the chunks that have thus far been concluded should be returned, but that reasoning should continue
- INTERRUPT – Similar to PEEK, but specifies that reasoning should be halted immediately

Note that if you do not specify the number of iterations, the NACS will reason continuously. The action commands can also be chained. For example, suppose you

¹ See the *Setting Up & Using the Goal Structure* or *Intermediate ACS Setup* tutorials for details on how to implement these action chunk types

want to reason for 20 iterations, but look at the conclusions halfway through. The following shows how this could be accomplished:

```
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.NEW, 10);
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.CONTINUE, 10);
```

On top of this, by leveraging the “parameter change” capabilities inherent to all action chunks, you can also specify various settings that you wish to use for the round of reasoning that is being requested. For example, suppose you only wanted the top level of the NACS to be used, the following demonstrates how this could be set up:

```
think.Add (John.NACS, "USE_BOTTOM_LEVEL", false);
think.Add (John.NACS, "USE_TOP_LEVEL", true);
```

Of course keep in mind that this will only work if the “think” action is initialized specifically for and is used solely by John. Also note that, as is the case with other internally-oriented actions, if a [ReasoningRequestActionChunk](#) is chosen by the ACS, the agent will deliver the “DO_NOTHING” action to the simulating environment as its chosen action for the particular sensory information that caused the [ReasoningRequestActionChunk](#) to be selected.

Although setting up a [ReasoningRequestActionChunk](#) is fairly straightforward, handling the coordination of the subsystems requires a little more thought. For example, the determination of which level(s) should recommend the reasoning request action and what the ACS should do while waiting for the conclusions from the NACS (or if it should wait) must be carefully devised and implement. Of course the particulars of how you implement this is based on the requirements of your task. However, the primary reason why this consideration is so important is because the NACS is designed to run in parallel with the ACS. This is done so that an agent can continue to interact with the world while reasoning is being performed. However, in many tasks (such as this), you may want the ACS to wait until the NACS has completed reasoning. For the full reasoner sample, a fairly simple process is used, with rules on the top level of the ACS and “states” utilized to coordinate the two subsystems. In other words, we will use the state concept to “modulate” when certain actions are recommended by the ACS. The following code demonstrates how this can be set up:

```
World.NewDimensionValuePair ("state", 1);
World.NewDimensionValuePair ("state", 2);
World.NewDimensionValuePair ("state", 3);

RefineableActionRule doReasoning =
    AgentInitializer.InitializeActionRule (reasoner,
        RefineableActionRule.Factory, think);

doReasoning.GeneralizedCondition.Add (World.GetDimensionValuePair ("state", 1));
reasoner.Commit (doReasoning);

RefineableActionRule doNothing =
```

```
AgentInitializer.InitializeActionRule (reasoner,  
RefineableActionRule.Factory, ExternalActionChunk.DO_NOTHING);
```

```
doNothing.GeneralizedCondition.Add (World.GetDimensionValuePair ("state", 2));  
reasoner.Commit (doNothing);
```

The idea here is that only certain rules will be eligible in certain states. For example, the first rule from above will only be eligible to fire when the ("state", 1) dimension-value pair is activated and second rule is only eligible when the ("state", 1) dimension-value pair is activated. Within the simulating world, the states transition as follows:

1. This state occurs in conjunction with the first presentation of the noisy pattern
2. This state occurs after the first time the agent issues the "DO_NOTHING" action (which is also correlated with the ACS making the reasoning request)
3. This state occurs when the working memory is updated inside of the agent

The code below demonstrates how this "state transition" process might be accomplished:

```
int state_counter = 1;  
while (chosen == null || chosen == ExternalActionChunk.DO_NOTHING)  
{  
    SensoryInformation si = World.NewSensoryInformation (reasoner);  
    si.Add (World.GetDimensionValuePair ("state", state_counter), 1);  
  
    int count = 0;  
    foreach (DimensionValuePair dv in dvs)  
    {  
        if (((double)count / (double)dc.Count < (1 - noise)))  
        {  
            if (dc.Contains (dv))  
            {  
                si.Add (dv, 1);  
                ++count;  
            } else  
                si.Add (dv, 0);  
        } else  
            si.Add (dv, 0);  
    }  
  
    reasoner.Perceive (si);  
    chosen = reasoner.GetChosenExternalAction (si);  
  
    if(reasoner.GetInternals(  
        Agent.InternalWorldObjectContainers.WORKING_MEMORY).Count() > 0)  
        state_counter = 3;  
    else  
        state_counter = 2;  
}
```

The `while` loop will continue the perception → action process for the current input until the agent responds either “yes” or “no” to whether that noisy pattern was the 2nd one. The rules and actions that enable this response can be set up as follows:

```
ExternalActionChunk yes_act = World.NewExternalActionChunk ("Yes");
ExternalActionChunk no_act = World.NewExternalActionChunk ("No");

RefineableActionRule yes =
    AgentInitializer.InitializeActionRule (reasoner,
        RefineableActionRule.Factory, yes_act);

yes.GeneralizedCondition.Add (World.GetDeclarativeChunk (1), true);
reasoner.Commit (yes);

RefineableActionRule no =
    AgentInitializer.InitializeActionRule (reasoner,
        RefineableActionRule.Factory, World.GetActionChunk ("No"));

no.GeneralizedCondition.Add (World.GetDeclarativeChunk (0), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (2), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (3), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (4), true, "altdim");
reasoner.Commit (no);
```

Specifying Alternative Dimensions

You might have noticed in the above code that we have specified something called `"altdim"` when adding conditional inputs to our “no” rule. This is the case because chunks themselves, by default, are considered self-contained and therefore are technically represented at the dimension-value pair level within their own dimension. That is, the library will treat the relation between chunks, when added to the condition of a rule, using the AND operation. However, at times it may be preferable to have the chunks organized together (i.e., within the same dimension) so that they can be compared using an OR operation. To enable this possibility, the `Add` method of a rule’s condition has an option “alternative dimension” parameter. Any world object (be it a chunk, dimension-value pair, etc.) that is added to the condition with an alternative dimension, will be “placed” in that dimension (for that rule). This does not change the objects dimension at the “descriptive” (i.e., world) level. However, it does allow for objects that are naturally correlated (by default) to be compared as if they were.

Filtering Input/Conclusions

The interaction between the ACS and NACS occurs seamlessly. That is, when reasoning request actions are chosen in the ACS, the NACS will automatically begin the reasoning process. Subsequently, when the NACS is finished reasoning, it’s conclusions will automatically be sent back to the ACS and the ACS will populate the working memory all on its own. However, this is done based on certain default assumptions. For instance, the ACS will send the entire sensory information object to the NACS to use as its input into reasoning whenever a reasoning request action

is chosen. Additionally, the ACS will always load all of the chunks that are concluded by the NACS into working memory (provided that the number of chunks do not exceed the working memory capacity). However, you may find instances where you will want to filter the inputs into the NACS and the conclusions from the NACS. To enable this, the `InputFilterer` and `KnowledgeFilterer` delegates have been defined:

```
delegate ActivationCollection InputFilterer(ActivationCollection input);  
delegate IEnumerable<ChunkTuple> KnowledgeFilterer(IEnumerable<ChunkTuple> input);
```

These delegates allow you to specify your own custom filtering operations. If specified (using the `ReasoningInputFilterMethod` or `ReasoningOutputFilterMethod` properties, found in `Agent.ACS`), the ACS will call these delegates prior to initiating reasoning or loading the conclusions into working memory.

Retrieving Chunks from the GKS

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Interacting with Episodic Memory

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Retrieving Episodes

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

“Offline” Learning

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Generative Actions

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

An Example: Using Generative Actions to Change Local Parameters

Remember, 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.

Using Plugins

© 2013. Nicholas Wilson

Table of Contents

The Serialization Plugin	1
Serializing (or Saving) Various Aspects of a Simulating Environment	1
De-serializing (or Loading) Various Aspects of a Simulating Environment.....	2
Interacting with Front-Ends	2
Remote Simulating Environments.....	2
Communicating via XML	2
Communicating via JSON	3
The Keyboard and Mouse Plugins	3
Using the “Built-In” Plugin Actions.....	3

The Serialization Plugin

Often times it may be the case, while in the process of tuning or training (etc.) a simulation, that you might wish to “suspend” the running of your task and then “resume” it at a later date while still maintaining all of the configurations, settings, and/or learning that has taken place within your simulating environment. To address these sorts of needs, the “built-in” objects (including both descriptive and functional objects) within the Clarion Library have been designed to be [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).

Furthermore, the library also contains a useful tool, the [SerializationPlugin](#)¹, whose purpose is to aid you in the process of serializing and de-serializing your simulating environment. In this section we will demonstrate how you can use of the [SerializationPlugin](#) to preserve the configuration of your simulations.

Serializing (or Saving) Various Aspects of a Simulating Environment

This feature has been developed, however the documentation, guides, and tutorials for it are currently incomplete. If you would like to use this feature and have any questions on how to make use of it, feel free to contact us at clarion.support@gmail.com. In future releases, this section will contain additional information describing how to use this feature.

¹ Located in the *Clarion.Plugins* namespace

De-serializing (or Loading) Various Aspects of a Simulating Environment

This feature has been developed, however the documentation, guides, and tutorials for it are currently incomplete. If you would like to use this feature and have any questions on how to make use of it, feel free to contact us at clarion.support@gmail.com. In future releases, this section will contain additional information describing how to use this feature.

Interacting with Front-Ends

Remote Simulating Environments

This feature has been developed, however the documentation, guides, and tutorials for it are currently incomplete. If you would like to use this feature and have any questions on how to make use of it, feel free to contact us at clarion.support@gmail.com. In future releases, this section will contain additional information describing how to use this feature.

Warning: this plugin has not been thoroughly tested. If you run into any problems while attempting to use it, please contact clarion.support@gmail.com

Communicating via XML

This feature has been developed, however the documentation, guides, and tutorials for it are currently incomplete. If you would like to use this feature and have any questions on how to make use of it, feel free to contact us at clarion.support@gmail.com. In future releases, this section will contain additional information describing how to use this feature.

Communicating via JSON

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

The Keyboard and Mouse Plugins

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Using the “Built-In” Plugin Actions

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Remember, 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.

Advanced Customization Tutorial

© 2013. Nicholas Wilson

Table of Contents

Getting Started	1
ACS Structure	2
NACS Structure	3
MS Structure	3
MCS Structure	3
Interfaces and Templates	4
How to Implement a Custom Component	5
Requirements for Implementing a Custom Component	5
Implementing a “Factory”	5
Implementing a “Parameters” class	10
Local (per instance) Parameters.....	11
Global (static) Parameters.....	15
Factor # 1	15
Factor # 2	15
Factor # 3	16
Factor # 4	17
Factor # 5	17
Factor # 6	19
Factor # 7	20
Committing and Retracting	21
Using the <code>InitializeOnCommit</code> Property.....	23
How to Implement a Custom (Secondary) Drive	24
Implementing the Nested “Factory” Class	25
Implementing the Nested “Parameters” Class	27
Serializing a Custom Component (or Drive)	29
Specifying the <i>System.Runtime.Serialization</i> Resource	29
The <code>DataContract</code> Attribute	30
The <code>DataMember</code> Attribute	31
Pre/Post Serialization and Deserialization Attributes	32
Serializing the Global (static) Parameters	33

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](#) classes² 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 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 :
        ImplicitComponentFactory<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, let's 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 e1 = null;

    foreach (dynamic p in parameters)
    {
        if (p is int)
            numNodes = (int)p;
        if (p is EligibilityChecker)
            e1 = (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 (e1 != null)
        fpars.Add(e1);
    result.FactoryParameters = fpars.ToArray();
}
```

```
    return new SomeCustomComponent(numNodes, e1);  
}
```

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 actions³ to operate correctly⁴

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 accomplish 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 you 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.

Committing 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. Your 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), simlance (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 group¹², 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 namespace¹⁴ 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 deserialization 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