2011

The LIDA Tutorial

J. Snaider

R. McCall

S. Strain

S. Franklin

The LIDA Tutorial

Version 1.0
Cognitive Computing Research Group
The University of Memphis

Javier Snaider – LIDA Framework main designer, developer, and team leader
Ryan McCall – LIDA Framework co-developer and designer
Steve Strain – LIDA Tutorial first draft author
Stan Franklin – CCRG director and the LIDA Model founder

I. Introduction

In order to successfully develop agents with the LIDA Framework, a basic understanding of the LIDA Model and its analogues in the Framework is necessary. The LIDA Model is both a conceptual model of minds and a computational model that provides an architectural design for autonomous agents. The LIDA Framework is a partial implementation of the LIDA Model in the form of a software framework using the Java programming language. Throughout our discussion, the reader should bear in mind the differences between the Model and the Framework. A concept may have a different meaning in the context of the Model than it does in the context of the Framework. The Framework constitutes one, but not the only, way of implementing the Model.

This LIDA Tutorial introduces the user to the basics of designing and implementing agents based on the LIDA Model using the LIDA Framework. This document begins with a high-level description of the LIDA Model and Framework. A more detailed description can be found in the paper "The LIDA Framework as a General Tool for AGI" (Snaider, McCall & Franklin, 2011). Next some specifics of the Framework's modules, processes, and fundamental data structures are detailed. A brief description of how an agent's architecture is specified declaratively by an XML file follows. Then, a high-level description of the Framework's default implementations of modules, processes, data structures, etc. is given. Finally, several Appendices appear, including a glossary, and additional details about the Framework.

a. The LIDA Model

The LIDA Model (Franklin and Patterson 2006; Ramamurthy et al. 2006; Baars and Franklin 2009) is a broad, conceptual and computational model covering a large portion of human cognition. Based primarily on Global Workspace theory (Baars 1988, 2002), the model implements and fleshes out a number of psychological and neuropsychological theories. The LIDA computational architecture is derived from the LIDA cognitive model. The LIDA Model and its ensuing architecture are grounded in the LIDA cognitive cycle. Every autonomous agent (Franklin and Graesser 1997), be it human, animal, or artificial, must frequently sample (sense) its environment and select an appropriate response (action). More sophisticated agents, such as humans, process (make sense of) the input from such sampling in order to facilitate their decision-making. The agent's "life" can be viewed as consisting of a continual sequence of these cognitive cycles. Each cycle consists of units of sensing, attending and acting. A cognitive cycle can be thought of as a moment of cognition, a cognitive "moment."

We will now briefly describe what the LIDA Model hypothesizes as the rich inner structure of the LIDA cognitive cycle. More detailed descriptions are available elsewhere (Baars and Franklin 2003, Franklin et al. 2005). During each cognitive cycle the LIDA agent first makes sense of its current situation as best as it can *by updating its representation of its current*

situation, both external and internal. By a competitive process, as specified by Global Workspace Theory, it then decides what portion of the represented situation is most in need of attention. Broadcasting this portion, the current contents of consciousness, enables the agent to chose an appropriate action and execute it, completing the cycle. Learning also takes place with the broadcast.

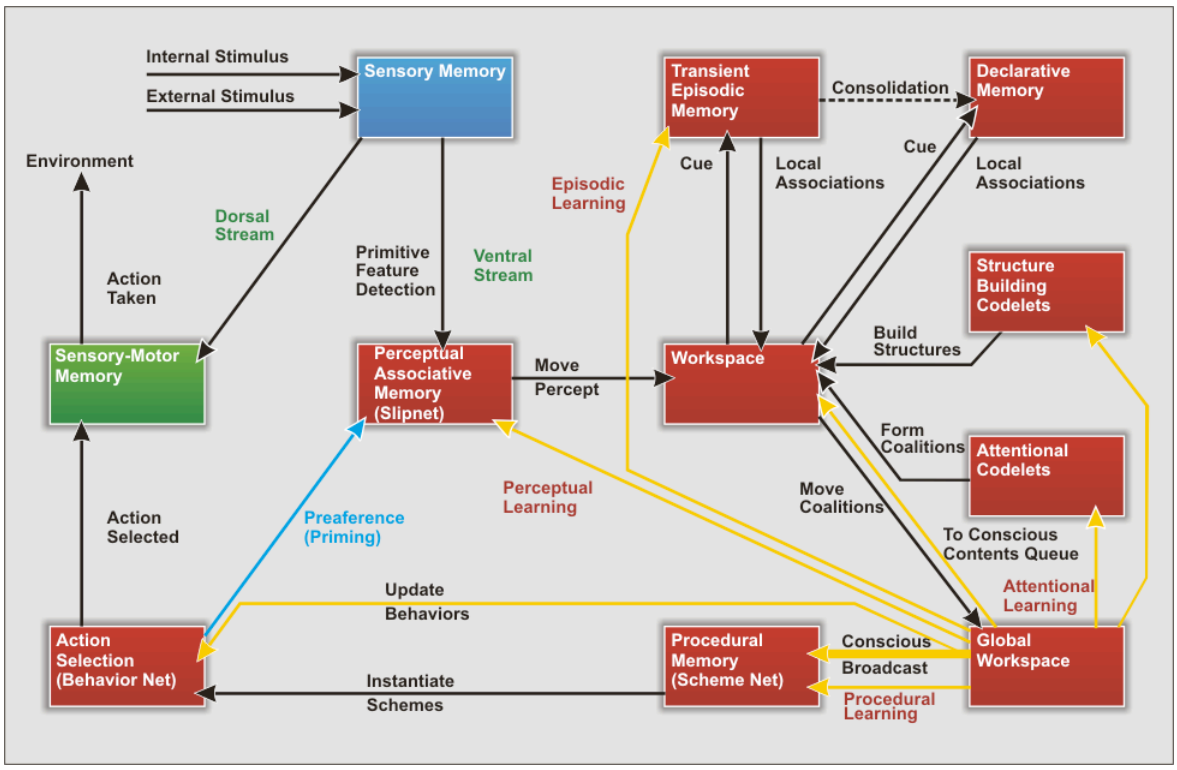

Figure 1. The LIDA Model Diagram

b. The LIDA Framework

Why use a software framework?

Intelligent software agents can be exceedingly complex systems and, as such, are difficult to implement and to customize. Frameworks have been applied successfully in large-scale software engineering applications. A framework constitutes the skeleton of the application, capturing its *generic functionality*. Frameworks are powerful as they promote code reusability and significantly reduce the amount of effort necessary to develop customized applications.

Features of this framework

Main features of the framework include default implementations of the major LIDA modules and several abstract classes for the generic parts of an agent architecture. Thus the framework can be used to implement other agent architectures besides LIDA. These generic parts are

described briefly as follows: A *module* is a cohesive collection of representations (data) and the processes the operate on them. A *task* is a short algorithm that can run repeatedly implementing a small process. Several data structures implement *common internal representations*. A dedicated *task manager* supports the multithreaded execution of tasks. A *configurable GUI* displays the current internal state of the processes and data in the system. A *factory* is used to create common data structures and *strategies*, which encapsulate common algorithms. Finally an *XML parser* is used to load and create an agent from a declarative XML file.

Relationship between the LIDA Framework and the LIDA Model

The Framework allows easy customization of an agent's architecture. Each code element in the Framework is encapsulated so that its functioning is coupled as loosely as possible with other code elements. Thus, all modules and processes depicted in the LIDA Model diagram may, but need not, be implemented. Moreover, the chosen modules and processes may be implemented using the default implementations provided with the Framework, or with custom code written by the user.

While default implementations are provided for many modules and processes, the user will need to develop domain-specific code for the agent's environment (or at least a sensory interface to its environment), its Sensory Memory module, and likely its Sensory-Motor Memory module as well. The relationship between the Framework and the Model is summarized in the following bullet points. See Section IV for details.

- The generic idea of a **module** in the Model is specified by the **FrameworkModule** interface
- Particular modules in the Model are typically specified by an interface, e.g., the Procedural Memory module is specified by the `ProceduralMemory` interface.
- **Module processes** in the Model are implemented using **tasks** of various types, which are themselves implementations of the `FrameworkTask` interface
- **Time** in the Model is represented by the Framework's internal time mechanism measured in a unit called the **tick**
- **Asynchronous execution** in the Model is implemented by concurrent execution of task s, which is managed by the `TaskManager`
- **Communication between modules** in the Model is implemented using listeners (observer design pattern) implementing the `ModuleListener` interface
- **Nodes and Links** in the Model are implemented using **NodeStructures** which, consisting of **Nodes** and **Links**
- The **activation** of Nodes, Links and other elements of the LIDA Model is achieved by an implementation of the `Activatible` interface. This implementation also includes **Strategies** that control the decay and excitation of these elements.

Consider the following example: in the LIDA *Model*, a conscious broadcast is sent from the Global Workspace during each cognitive cycle. This broadcast is sent to most modules in the Model, as illustrated by the arrows in the LIDA Model diagram (Figure 1). However, modules in a Framework agent will not receive conscious broadcasts unless they implement the `BroadcastListener` interface and register themselves as listeners of the `GlobalWorkspace` module.

II. Basics of the LIDA Framework

The LIDA Framework provides a generic structure for the creation of agents using the architecture defined by the LIDA Model (see Figure 1). It is worth noting, however, that the Framework's flexibility will allows it to possibly support other architectures as well. Agents created with the Framework are composed of modules, listeners, and tasks. Tasks are run using TaskSpawners, which work closely with the `TaskManager`. In this section we describe each of these basic features of the Framework.

a. Modules and Listeners

Modules

The LIDA Model has numerous modules as part of its conceptual architecture. Examples include Sensory Memory, Perceptual Associated Memory, and the Workspace. Some of the modules are highly domain-dependent (e.g. Sensory Memory and Sensory-Motor Memory), while others are very much domain-independent (e.g. the Workspace).

The LIDA Framework provides concrete default implementations for the domain-independent modules, which are easily customizable. Abstract default implementations of the domain-dependent modules and tasks are also provided. These classes must be extended for each particular agent implementation.

Modules may contain submodules, for instance, the `Workspace` contains the Current Situational Model as submodule. Each submodule is itself a module.

In the LIDA Framework, modules are specified generically by the `FrameworkModule` interface. A default implementation, the abstract class `FrameworkModuleImpl`, is provided. Modules extending this implementation inherit a name, zero or more submodules, and a `TaskSpawner` to help run tasks. This default implementation specifies a *structure* for the decay of the module and its submodules and specific modules must implement their own algorithm to decay the module. Similarly `FrameworkModuleImpl` defines a default method for adding listeners, and specific module implementations should override it and handle the specifics. Specific module implementations must also implement methods and processes necessary to achieve their specific functions.

Each module in the LIDA Model has a corresponding interface in the Framework that specifies the required methods for that module's implementation. Custom module implementations can be used provided they extend from `FrameworkModuleImpl` and implement the appropriate interface. For instance, an implementation of Sensory Memory will extend `FrameworkModuleImpl` and will implement the `SensoryMemory` interface.

Many default implementations of modules and tasks in the framework inherit from the few key abstract classes (see Appendix 2).

Listeners

Modules must pass information to each other in order to have a connected system. These communication channels are indicated by arrows in the LIDA Model Diagram. The LIDA Framework utilizes a design pattern known as the Observer Pattern as a generic way to implement communication between modules. To this end, many modules in the Framework provide a Listener interface, which specifies the methods necessary to receive content from that module. Any module wishing to receive the communications sent by this module must implement this interface (i.e. it must implement all of the methods in the Listener interface) and register itself to the sending module. The Listener methods allow the producer module to send information to all of its registered listeners in a generic way, without needing to know specific details about the listening modules themselves.

An example of the use of the Observer Pattern is illustrated by the relationship between the `GlobalWorkspace` module and the `BroadcastListener` interface. The Model demands that most modules receive the conscious broadcast from the `GlobalWorkspace`. Thus these modules in the Framework must implement the `BroadcastListener` interface. Each receiving (consumer) module must then register itself with the `GlobalWorkspace` module using its `addBroadcastListener` method. When the `GlobalWorkspace` needs to send a broadcast, it simply calls the `receiveBroadcast` method of each of the registered listeners.

Listeners are not the only mechanism to create a relationship between modules. Any module can have one or more *associated modules*. In this relationship the associated module is more tightly coupled than if it were added as a listener. Modules are associated using the `addAssociatedModule` method in the `FrameworkModule` interface. This allows for close relationships between modules that may be required for some modules' functionality. For instance, the `AttentionCodeletModule` needs to directly access to both the `Workspace` and `GlobalWorkspace` modules. Using codelets this module will check and retrieve content from the `Workspace`. Also its codelets will be adding coalitions directly to the `Global Workspace` (and only this module).

b. Tasks, TaskSpawners, and the TaskManager

Tasks

In the LIDA Framework, the processes of the LIDA Model are implemented using *FrameworkTasks,* or simply "tasks". A task can be thought of as a "demon" as in Jackson's Pandemonium Theory (1987), or as a codelet a la Hofstadter (1995). Every agent application has a single `TaskManager` that is the heart of the LIDA Framework's simulation engine. The `TaskManager` uses a pool of threads for the execution of tasks. Modules do not directly use the `TaskManager`; rather, each module in the Framework has an associated `TaskSpawner` that works with the `TaskManager`. Next we describe additional aspects of the `TaskManager`.

The `TaskManager` controls the application's internal time. The unit of this time is termed the *tick*. All tasks are scheduled for and executed at a particular tick. The *current tick* is an integer starting at zero, which is incremented as the execution of an agent simulation progresses.

Each task has an algorithm contained in its `runThisFrameworkTask` method, a scheduled tick (i.e., the tick at which it will be executed next), and a status. Tasks can be run a single time or repeatedly. For instance, to pass activation to a Node (see below), an `ExciteTask` is executed once. On the other hand, a feature detector is implemented as a task that runs every few ticks. The run frequency is determined by the `ticksPerRun` attribute of the task. Additional features exist for the fine tuning of this mechanism.

During a task's execution, its status may change, and, after execution, the task is returned to the `TaskSpawner` that originally ran it. The `TaskSpawner` then has an opportunity to process the task's result, and, based on the task's status, determine whether the task should be scheduled for further execution.

The TaskManager

The `TaskManager` is in charge of the execution of tasks and the application's internal time. It receives tasks to be run from TaskSpawners. It schedules these tasks for execution in a queue called the *task queue*. The task queue is ordered by tick and each position in the queue contains the tasks to be executed during at that tick.

The `TaskManager`'s main loop cycles through the following sequence: 1) Decay all modules; 2) Execute all tasks scheduled for the current tick *and* wait for them to finish; 3) Possibly refresh the GUI; and finally, 4) increment the current tick. The `TaskManager` has public methods to start, stop, and resume the execution of this sequence. The Start/Pause button in the toolbar of the Framework GUI uses these methods to control the execution of an agent simulation.

The *tick duration* is the minimum amount of real time it takes for a single tick to occur. This can be set by the user to control the *apparent* speed of a simulation. The actual duration of a given tick may exceed this minimum because the execution of the scheduled tasks may take

longer than the specified duration. (This depends on the speed of the microprocessor and the number and complexity of the tasks scheduled for that tick.) If tick duration is set to 0, task execution will proceed as quickly as possible (this speed will be machine-dependent).

Stated another way, the execution of any task always takes exactly one tick in Framework time. Depending upon the size of the `TaskManager`'s thread pool, multiple tasks can be executed concurrently, but still within a single tick. Some tasks may take longer (in real time) than others to execute, and the execution of many concurrently scheduled tasks may take longer than the execution of a few, so due to this variability, ticks do not in general have a uniform length in units of real time.

In the agent declaration file (see Section IIIa), the `TaskManager` can be configured using two parameters, `tickDuration`, and `maxNumberOfThreads`. The former parameter is the starting tick duration and the latter determines the size of the thread pool that the `TaskManager` uses to execute tasks. Additionally, the tool bar in the Framework's GUI provides a means to adjust tick duration interactively.

c. Nodes, Links, and NodeStructures

Nodes and Links are major data structures in the LIDA Framework. Both have activation, a measure of current salience, which can be excited in several situations and which decays over time. A `Node` can represent features, objects, events, concepts, feelings, actions, etc. Every `Node` is grounded in a `Node` in PAM called a `PamNode`. Nodes are instantiations of PamNodes. Every `Node` in a simulation has a reference to its originating `PamNode` in PAM, as well as an *id* that uniquely identifies the `Node`.

A `Link` connects a `Node` to either another `Node` or another `Link`. A `Link` that connects two Nodes is known as a simple Link. On the other hand, a complex Link connects a source `Node` to a simple Link. For example, given two Nodes, B and C, connected by a simple `Link`, L2, a complex Link would be one that connects a third `Node`, A, to the simple Link L2. Additionally, Links have a `LinkCategory` attribute that specifies the nature of the relationship represented by the `Link`. For instance, a red ball can be represented by a "ball" `Node` connected to a "red" `Node` by a `Link` with `LinkCategory` "feature". Both Nodes and Links implement the `Linkable` interface, and have an *`ExtendedId`* that uniquely identifies any `Linkable` element.

The Framework allows the specification of different subtypes of Nodes and Links. In this way custom Nodes and Links can have their own implementation class that customizes their properties and functionality. All `Node` and `Link` instances, both default and custom types, are created dynamically by the `ElementFactory` (discussed below).

A `NodeStructure` is a graph-like structure composed of Nodes and Links, and is the "common currency" for information exchange between most modules. The `NodeStructureImpl` implementation provides methods to add, remove, or retrieve, and manage the Nodes and Links contained inside it. When a `Node` or a `Link` is added to a `NodeStructure`, a copy of the element is added (and returned) rather than the element itself.

This prevents the same Java object from existing in multiple NodeStructures, a problematic situation. While the objects are different, the new `Node` retains (at least to start) the same attributes as its original `Node` including its id. This also allows the new element to have different parameters and/or functionality than the original element. For example, when a `Node` in PAM is added to the Workspace's Perceptual Buffer, the new `Node` may have its own activation and different mechanism for its excitation and decay.

d. Activation and Strategies

Nodes, Links, and other elements including coalitions, codelets, schemes, and behaviors have activation. Activation is represented as a real number between 0.0 and 1.0 inclusive. In general activation represents the salience of an element. The elements mentioned above all have a current activation, which measures their salience at the current moment. Other elements have an additional activation called base-level activation, which is used to implement learning. The Framework provides the `Activatible` and `Learnable` interfaces for the implementation of elements with these functionalities.

An `Activatible` element has `excite` and `decay` methods to regulate its activation appropriately. Elements are typically removed if their activation decays below a certain *removal threshold*. The `Activatible` must specify two Strategies to determine how its activation level changes when its `excite` and `decay` methods are called. In this way, the excitation and decay functions of elements can easily be configured.

A `Learnable` element, implementing the `Learnable` interface, is an extension of `Activatible`. In addition to a current activation, it additionally has a base-level activation to implement learning. A `PamNode` is one example of a `Learnable` element; the base-level activation might correspond to the past usefulness of the concept represented by a `PamNode`. The current version of the LIDA Framework does not yet implement learning algorithms in its default implementations. In order to implement learning at this time, custom module implementations could be developed that implement a learning method. Such an algorithm would modify the base-level activation of `Learnable` elements.

As mentioned above, the LIDA Framework uses the Strategy design pattern to implement excitation and decay of `Activatible` elements. This approach allows for fine control, easy modification, and reuse of activation mechanisms. Default implementations for linear excitation, sigmoid excitation, linear decay and sigmoid decay are provided with the LIDA Framework. Strategies are obtained from the `ElementFactory` (see next section).

e. Framework Tools

ElementFactory

New instances of many elements in the LIDA Framework, such as Nodes, Links, tasks and Strategies can and *should be* created by requesting them from the `ElementFactory`. This utility implements the Singleton and the Factory design patterns. In general factories provide valuable encapsulation of the constructor of the various elements it produces. This also allows for the addition of new elements types using different *classes* such as new `Node` and `Link` types, tasks (which include feature detectors, attention codelets, and others), and strategies (which include excite, decay, and others). Additionally element types can specify particular activation, strategies, and parameters. For example several `Node` types can be defined that all use the same underlying class, e.g. `NodeImpl`, but have different excite and decay strategies and initial activations. Each element type definition includes a *name*, which is used to request a new instance of that type from the `ElementFactory`. The element types that the `ElementFactory` can produce can be configured in the factoryData.xml file. Type definitions include the name, class and parameter values that establish the factory settings for each element type.

The Graphical User Interface (GUI)

The LIDA Framework includes a customizable GUI that allows the real-time display of module contents, parameter values, running tasks, and other variables of interest during the operation of a LIDA agent. A "GUI Panels" properties file allows the addition of default or custom panels and configures the appearance of the GUI Panels in the Framework's GUI.

III. Framework Initialization

The Framework's `initialization` package contains the classes involved in the run-time setup of an agent simulation. The class `AgentStarter` provides methods to run an agent simulation. It also provides a main method that can be executed from the command line with a parameter specifying the path of the *primary configuration file*. The `start` method of `AgentStarter` performs the following steps:

1) It loads the definitions of the element types found in a secondary configuration file known as the *factory data file*. This file is specified in the `lida.elementfactory.data` property of the primary configuration file (the default name of which is `lidaConfig.properties`).

2) An instance of the `Agent` class is created based on the contents of the *agent declaration file*. This file is specified in the `lida.agentdata` property of the primary configuration file. See the following sections for further details.

3) If the `lida.gui.enable` property in the primary configuration file is set to `true`, the framework's graphical user interface is created using the information found in the *GUI configuration files*, specified by the `lida.gui.panel` and `lida.gui.commands` properties of the primary configuration file.

4) The agent is loaded

5) The GUI is loaded and displayed.

In the following sections, some of the details of this process will be described.

a. The Agent Declaration File

The file specified in the `lida.agentdata` property of the primary configuration file contains the declaration of the components of the agent, and is known as the agent declaration file. This is an XML document that contains the root tag, `<lida>`, and has the following first level nested tags.

- `<globalparams>` — an optional tag that contains parameters to be used during the module initialization process
- `<taskmanager>` — contains parameters for the configuration of the TaskManager , as explained in a previous section
- `<taskspawners>` — contains subtags of type `<taskspawner>` that declare the TaskSpawner types available to the agent's modules
- `<submodules>` — contains subtags of type `<module>` that specify the classes and parameters necessary to create the modules of the agent; optionally, the <module> subtags may contain nested tags of type `<associatedmodule>`, `<initialtasks>`, `<initializerclass>`, or even `<submodules>`, the latter allowing for declaration of nested modules
- `<listeners>` — contains subtags of type `<listener>` that declare the listener connections between the agent's modules

A formal specification of these tags can be found in the file `LidaXMLSchema.xsd`.

b. AgentXMLFactory

The `AgentXMLFactory` class loads the agent declaration file, parses it, creates the components declared in the file, and assembles them into an `Agent` object. The `getAgent` method performs the following sequence of actions.

1) The global parameters, if any, are read and added to the `GlobalInitializer` class.
2) The `TaskManager` is created.

3) The TaskSpawners are created.
4) The modules are created. At this time, the `init` method of each module is called.
5) The `ModuleListener` connections are established.
6) Optionally, the modules are associated, if this was specified in the module declarations.
7) Each module that specifies an initializer class in its declaration is re-initialized using the specified class.
8) For each module that has as initial tasks specified in its declaration, the specified tasks are created, and the module's `TaskSpawner` schedules them for execution.

c. The Factory Data Definition File

The file specified in the `lida.elementfactory.data` property of the primary configuration file contains the definition of the `Strategy`, `Node`, `Link`, and task types to be added to the `ElementFactory` at startup, and is known as the factory data definition file. Note the `ElementFactory` internally defines several default types not specified in this file including "NodeImpl", "PamNodeImpl", "noDecayPamNode", "LinkImpl", "PamLinkImpl", "noDecayPamLink", "defaultDecay", "defaultExcite", "noDecay", "noExcite".

This XML document that contains the root tag, `<LidaFactories>`, and has the following first level nested tags.

- `<strategies>` — contains subtags of type `<strategy>` that define the `Strategy` types to be loaded into the `ElementFactory`
- `<nodes>` — contains subtags of type `<node>` that define the `Node` types to be loaded into the `ElementFactory`
- `<links>` — contains subtags of type `<link>` that define the `Link` types to be loaded into the `ElementFactory`
- `<tasks>` — contains subtags of type `<task>` that define the task types to be loaded into the `ElementFactory`

d. The Initializable Interface

Several elements in the Framework implement the `Initializable` interface, including the default implementations of `Node`, `Link`, and `Strategy`. This interface provides a uniform way to initialize elements with parameters. Subclasses that users will typically work with need only override the `init` method specified by this interface to perform custom initialization of the element. Within the overriding `init` method the `getParam` method should be used to retrieve parameters set in the configuration files.

The `FullyInitializable` interface extends `Initializable` and additionally allows the association of modules using the `setAssociatedModule` method. `FrameworkModule` and `FrameworkTask` are examples of `FullyInitializable` elements.

IV. Default Implementations of LIDA Modules

Default implementations of the following LIDA modules are included in this version:

- Environment (abstract)
- Sensory Memory (abstract)
- Perceptual Associative Memory
- Transient Episodic Memory
- Declarative Memory
- Workspace
- Structure-Building Codelets
- Attention Codelets
- Global Workspace
- Procedural Memory
- Action Selection
- Sensory-Motor Memory

There are several details to mention regarding the default module implementations:

The Environment module is designed to be one that the user provides. The framework provides a basic abstract class, `EnvironmentImpl`, which should be extended implementing either an environment for the agent in Java or serving as an interface between a non-Java environment implementation and the agent. Users' Environment module implementations should, at a minimum, implement the methods `getState` and `processAction`, which allow the environment to be sensed and process agent actions respectively.

The Sensory Memory module is currently provided in the form of an abstract class, as sensory memories typically vary greatly from domain to domain. Thus users should extend from the default implementation, `SensoryMemoryImpl`, and implement the domain-specific parts. There are plans for a Sensory Memory implementation with object recognition algorithms in future framework versions.

The default Action Selection implementation provided in the current version does not implement all the features that are described by the LIDA Model. An enhanced Behavior Network (Maes 1989) implementing these additional features is planned for the next version of the Framework.

The two episodic memories describe by the LIDA Model, Transient Episodic Memory and Declarative Memory, are both implemented using a sparse distributed memory (Kanerva, 1988).

Finally, several learning algorithms specified by the LIDA Model have not yet been implemented for any default module in this version.

Acknowledgments

We want to thank the members of the CCRG for their invaluable help and insightful discussions, the members of the computational LIDA group for their participation in the development and testing of the framework, and especially Stan, for his unconditional support and inspiration.

Appendix 1: Glossary of Terms

Action
An action is a conceptual memory of a procedure the agent can perform, e.g., grab a cup, which can have several associated algorithms for execution in Sensory-Motor Memory.

Activation
Activation can represent different things, but, generally, it represents the salience or usefulness of an element. Nodes, Links, and other elements such as coalitions, codelets, and behaviors, have activation.

Aggregate-Coalition-Activation Trigger
A broadcast trigger that fires when the sum of the activations of all coalitions in the Global Workspace is greater than a certain threshold.

Attention Codelet
An Attention Codelet is a type of task that tries to bring particular Workspace content in the Current Situational Model to the Global Workspace. Upon finding such content it creates a coalition containing the content and adds it to the Global Workspace to compete for consciousness.

Autonomous agent
A system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda, and so as to affect what it senses in the future. Most animals and software agents are examples.

Background Task
A task, typically internal to a module, which implements a process integral to the module's functionality. Such tasks run repeatedly "in the background". An example is the `SensoryMemoryBackgroundTask` that runs the agent's sensors each time it is executed.

Base-Level Activation
Activation that represents the general usefulness and relevance of an element in the past. Its value can range from 0.0 to 1.0 inclusive and is modified by learning.

Behavior
A behavior is an instantiated scheme. Schemes are instantiated in Procedural Memory. Instantiation binds variables in the scheme's context and result, based on the current conscious

broadcast. Instantiated schemes (behaviors) are then sent to Action Selection where they compete with each other for selection and execution.

Behavior Network
An implementation of the Action Selection module inspired by Maes' (1989) Behavior Net. It selects a behavior to execute at the conclusion of each cognitive cycle. This selection is based on the activation of the Behavior, which is itself dependent on the current contents of consciousness as well as the relationship the Behavior has with other Behaviors currently in the Behavior Network.

Bit Vector
Bit vectors are typically large vectors with values of either 0 or 1. They are used in the Sparse Distributed implementation of Episodic Memory as a means to store episodic memories.

Broadcast Trigger
A process that monitors the Global Workspace for a particular condition. Upon detecting that condition it prompts the Global Workspace to hold a competition for consciousness.

Coalition
A coalition contains a portion of the Current Situational Model that is brought to the Global Workspace as a unit along with the attention codelet that created it. Coalitions compete for consciousness based on their activation derived from both the activation of the content and of the attention codelet. The winning coalition's contents are broadcast throughout the system.

Cognitive Cycle
In the LIDA Model, a sequence of processes that occur over time beginning with the sensing of stimuli, proceeding with understanding, then continuing with attention (consciousness), and ending with action selection and execution. Thus this cycle can be conceived of as having three main phases: understanding, attention (consciousness), and acting. Multiple cognitive cycles may be occurring in parallel (cascade): as one cognitive cycle has reached the action phase another could be in the attention phase while a third is in the understanding phase. A cognitive cycle can be thought of as a cognitive "atom" — the basic unit on which higher-level cognitive processes are built. The agent's "life" can be viewed as consisting of a continual cascade of these cognitive cycles.

Complex Link
A Link that has a Node for its source and a simple Link for its sink. Complex Links cannot be sinks.

Conscious Contents Queue
The Conscious Contents Queue holds the contents of the last several seconds of broadcasts and allows LIDA to ground and produce time-related concepts including duration.

Current Activation
An activation that represents an element's current salience. Can take a value from 0.0 to 1.0 inclusive.

Current Situational Model
The Current Situational Model is a buffer in the Workspace containing structures representing the agent's current situation. It contains a perceptual scene comprised a real and virtual part. The former consists of the current internal representation of the real world while the latter consists of the current internal rememberings, plannings or imaginings of the agent.

Decay Strategy
A strategy that decreases an activation value over time (in ticks). May be implemented using a sigmoid function for example.

Declarative Memory
Declarative Memory is a module in the LIDA Model, also called long-term episodic memory. Declarative Memory can be subdivided into the autobiographical memories of events, e.g., your last birthday celebration, and the semantic memory of facts like "Paris is the capital of France."

ElementFactory
The ElementFactory's main purpose is to create new element objects (e.g. Nodes) in a centralized, configurable way. This class implements the singleton and factory design pattern and thus provides an encapsulation of the constructors of basic framework elements. It is the preferred way to obtain new Nodes, Links, some tasks, and Strategies.

Episodic Memory
Episodic Memory is a memory for events, the what, the where, and the when of a given episode. In the LIDA Model it is comprised of Transient Episodic Memory and Declarative Memory.

Excite Strategy
A strategy that increases an activation value by a specific amount. May be implemented using a sigmoid function for example. Note that exciting activation value with a negative amount will likely decrease the activation.

Feature Detector Algorithm
A feature detector algorithm is a specific type of task. Its function is to detect a particular feature in Sensory Memory and excite a PAM Node representing this feature in Perceptual Associative Memory.

Global Workspace
A module where coalitions, created and added by AttentionCodelets, compete for consciousness. The content carried by the winning coalition becomes the current contents of consciousness and it broadcast throughout the system.

Global Workspace Theory
A theory of consciousness and cognition proposed by Baars, which describes consciousness functionally as a process by which many unconscious processors compete for access to a "Global Workspace." The winning processor has its contents broadcasted globally throughout the cognitive system. It is one important foundation theory of LIDA, which implements the Global Workspace and global broadcast.

GUI Command
GUI-generated events, e.g. a button press, that require a response in the model (the agent) are implemented as GUI commands. These are thus encapsulations of commands from the GUI to be performed in the model. GUI commands implement the Command design pattern.

Individual-Coalition-Activation Trigger
A broadcast trigger that initiates a competition for consciousness if there exists a coalition with activation above a certain threshold in the Global Workspace.

LIDA Cognitive Model
An integrated artificial cognitive system that attempts to model a broad spectrum of cognition in biological systems, from low-level perception/action to high-level reasoning. It is empirically grounded in cognitive science and cognitive neuroscience. In addition to providing hypotheses to guide further research, the model can support control structures for software agents and robots.

LIDA Software Framework
A software framework developed to allow a customizable implementation of the LIDA Cognitive Model. Specifically, an application programming interface (API) containing Java classes and interfaces that implement modules and processes according to the computational architecture of the LIDA Model. Also, due to its high degree of flexibility, this framework is suitable for wider use in developing other cognitive architectures.

Link
A connection between a source Node and a sink, which may be either a Node or a *simple Link*. A Link has a LinkCategory, which is the Link's conceptual meaning.

LinkCategory
LinkCategory is the conceptual meaning of a Link, e.g. membership, lateral, causal, parent, none, etc.

Listener
Modules need to communicate with other modules. To implement this, we use the observer design pattern. In this pattern a "listener" module registers itself with another "producer" module. Each time the producer has something to send, it transmits the information to all of its registered listeners.

Logger
Logger is a Java utility used to log messages about important events and warnings from specific framework components.

Module
Similar to a module in the LIDA Cognitive Model, we generically define a module in the Framework. Modules contain information of various kinds, and include processes and algorithms with which to manipulate, modify, and move that information. All modules implement a generic module interface (API) but each one also has its own API that defines its particular functionality.

Node
A Node is a major unit of representation in the framework, which can represent a feature, object, category, event, etc.

NodeStructure
NodeStructure is a graph structure, containing Nodes and the Links between them. It constitutes the main internal representation for many framework modules.

No-Broadcast-Occurring Trigger
A broadcast trigger that initiates a competition in the Global Workspace when no broadcast has occurred for a certain period of ticks.

No-Coalition-Arriving trigger
A broadcast trigger that initiates a competition in the Global Workspace when no new coalition has been added to the Global Workspace for a certain period of ticks.

Perceptual Associative Memory (PAM)
A module that makes sense of the incoming sensory information, creating additional understanding. Contains feature detector processes, which detect features in Sensory Memory and excite their associated PAM Nodes. The Node-and-Link memory of PAM is implemented as a semantic net with activation passing. Nodes propagate their activation to the other Nodes they are linked to. Those Nodes that gain activation above their threshold become part of the current percept, which becomes a part of working memory (enters the Workspace). PAM is essentially recognition memory.

Procedural Memory
A module where schemes are stored. Here schemes can be activated by each conscious broadcast, and, if this puts them over their threshold, are instantiated as behaviors and sent to Action Selection. Procedural Memory essentially remembers what to do in what circumstance to achieve a given result.

Receptive Field
The set of units from which a feature detector (PAM Node) takes as its input. The units can be other PAM Nodes or elements in Sensory Memory.

Scheme
Schemes are memories, stored in Procedural Memory, for various actions (procedures), which may be selected for execution. A scheme consists of a context, an action and a result. With some reliability, a scheme's result is expected to occur when its action is taken when its context is satisfied.

Sensory Memory
A module that senses the environment and holds the incoming sensory stimuli for a brief period. Low-level processing (e.g. color, texture, segmentation in the visual modality) of the sensory stimuli occurs here. Feature detectors have their receptive fields in Sensory Memory.

Sensory-Motor Memory
A module that stores algorithms (motor plans) for actuator execution. This module receives selected Behaviors from the Behavior Network and determines the appropriate algorithm for executing the action. The agent's actuators then execute the algorithm. During the execution it receives information from Sensory Memory in an implementation of the so-called "dorsal" stream.

Simple Link
A Link that connects two Nodes.

Strategy
Strategies are implementations of the Strategy design pattern. They are encapsulations of calculations, algorithms, etc. which may have several desirable implementations. This allows for flexibility, for example, either a linear function or a quadratic function could be used as a Node's excite strategy (provided they implement the ExciteStrategy interface).

Structure-Building Codelet
A Structure-Building Codelet is a specific type of task. It operates with representations in the submodules of the Workspace making associations among representations, creating new representational structures, and combining similar ones. They play a role in maintaining the agent's model of its current situation in the Current Situational Model.

Submodules
Modules can have submodules, which are modules themselves that are nested inside another module. For example, the Workspace module in LIDA has several submodules such as the Current Situational Model.

Task
Tasks are encapsulations of small, specialized, routines (algorithms), which typically run repeatedly at a specified rate. Tasks consist of an algorithm, an execution frequency, a time of next execution, and a status. Modules or even other tasks can create tasks to perform the small operations needed to achieve their specific functionalities. Tasks may run once or repeatedly. If they run repeatedly the period is based on their execution frequency.

TaskManager
The TaskManager controls the scheduling and execution of all tasks in the framework. It maintains the Framework's internal clock measured in *ticks*. Tasks are scheduled for execution in accordance with this clock.

Task Queue
A queue maintained by the TaskManager, as a schedule of the execution of tasks. Each position in the queue represents a tick and can contain 0 or more tasks. The TaskManager advances one position at a time, executing all tasks scheduled at that position (for that tick). It does not proceed to the next position until all tasks scheduled for the current position have finished running.

TaskSpawner
A TaskSpawner is an assistant to the TaskManager, which manages tasks' status and determines what to do with a task after each time it runs.

Tick
Each position in the task queue represents a discrete instant in the application's time. These positions are termed 'ticks'. All tasks scheduled for a particular tick are executed concurrently and the TaskManager does not advance to the next tick until all tasks scheduled for the current tick finish.

Total Activation
An activation value that represents the total salience of an element. It is a function of both current activation and base-level activation.

Transient Episodic Memory
Transient Episodic Memory is a module in the LIDA Model. It is the episodic memory that decays after a few hours or up to a day. Declarative memories are formed offline from transient episodic memories by a process called consolidation.

Workspace
A module that receives and stores content from several modules including current percepts from PAM, recent local associations from Episodic Memory, and the recent contents of consciousness from the Global Workspace. Structure-building codelets operate on Workspace content, integrating structures, creating new Nodes, and building structures representing higher-level phenomena (events, plans, imaginations, etc.).

Appendix 2: Classes and Interfaces

The table below maps several framework classes to the broad concepts of the Framework described in this tutorial. However, this is not a comprehensive description of the classes in the Framework. For detailed description of all of the classes in the Framework see its Javadoc (http://ccrg.cs.memphis.edu/assets/code/doc/index.html).

Framework Modules	interface FrameworkModule abstract class FrameworkModuleImpl
Framework Tasks	interface FrameworkTask abstract class FrameworkTaskImpl interface TaskSpawner class TaskSpawnerImpl class TaskManager
Activation	interface Activatible class ActivatibleImpl interface Learnable class LearnableImpl
Node, Link, and NodeStructure	interface Node class NodeImpl interface Link class LinkImpl interface NodeStructure class NodeStructureImpl class ExtendedId
Strategies	interface Strategy interface ExciteStrategy interface DecayStrategy
Initialization	interface Initializable interface FullyInitializable interface Initializer
Default Module Implementations	abstract class EnvironmentImpl abstract class SensoryMemoryImpl class PerceptualAssociativeMemoryImpl class WorkspaceImpl class WorkspaceBufferImpl class BroadcastQueueImpl class EpisodicMemoryImpl class AttentionCodeletImpl class StructureBuildingCodeletImpl class GlobalWorkspaceImpl class ProceduralMemoryImpl class BasicActionSelection class BasicSensoryMotorMemory
Default Task Implementations	abstract class BasicDetectionAlgorithm abstract class MultipleDetectionAlgorithm abstract class AttentionCodeletImpl abstract class StructureBuildingCodeletImpl

Appendix 3: XML Configuration Basics

Agent declaration file

In this discussion, `agent.xml` will refer to an xml file containing the definitions needed by `AgentXMLFactory` to build the agent in the LIDA Framework. Any valid file name can be used for this file, but the proper path and file name must be specified in the `lida.factory.data` element of the `lidaConfig.properties` file (see also Section III). The agent.xml file contains the `<lida>` tag as its root element. This tag contains the following nested tags:

Tag	Nested Tags	Contents
<**globalparams**>	(node)	global parameters available to Initializers during setup
<taskmanager>	(none)	tick duration, max. number of threads
<taskspawners>	<taskspawner>	name, Java class
<submodules>	<module>	name, Java class, (associated modules), TaskSpawner, (initial tasks*)
<listeners>	<listener>	type, sending module, receiving module

() — indicates an optional element

* — `<initialtasks>` contains one or more `<task>` definitions as defined in the `factoryData.xml` file (see Section IIIc).

<taskmanager>

Two TaskManager parameters can be configured in this section:

`tickDuration` is the length of one tick in milliseconds of actual time

`maxNumberOfThreads` is the maximum number of threads the `TaskManager` will maintain.

Let's discuss the structure of these parameters, and for parameters in the `agent.xml` file in general. Here is a parameter example:

```
<param name="taskManager.tickDuration" type="int">50 </param>
```

All parameters appear within <param> tags. Parameters should have a name, which will be parsed as a String. It is convention to differentiate parameters across modules, tasks, etc. by beginning the name with the module or task name. The variable types "int", "double", "boolean", or "string" can be specified. If type is specified the AgentXMLFactory will parse a variable of that type. Finally, the value of the parameter goes inside the parameter tag.

<taskspawners>

Several TaskSpawners can be defined here for modules to use. All definitions must include a name and a class name. Optionally parameters can be defined too. The name will be used to refer to the taskspawner in other places in the declarative agent file, e.g., when modules declare their taskspawner.

<submodules>

One or more <module> tags can be specified inside this tag. The tag's name is "submodules" because all these modules will technically be submodules of a special FrameworkModule called *Agent*.

Every <module> must have a String name, a class name, and a taskspawner specified for it. Optionally the module can have one or more associatedmodule specified by module name (as defined in this same file). The optional <initializerclass> tag specifies the class name of the Initializer to be run at startup to configure this module. Also optionally, one or more tasks may be specified in an <initialtasks> tag. These will be created and added to the module's taskspawner at startup. Modules declared by name within a <submodules> tag will be stored as submodules inside the module. Finally a <module> can have several parameters defined for it was described in the previous section.

Here is an illustrative example of a <module> declaration, which uses some, but not all, of the possible tags:

```
<module name="AttentionModule">
<class> attentioncodelets.AttentionCodeletModule</class>
      <associatedmodule>Workspace</associatedmodule>
      <associatedmodule>GlobalWorkspace</associatedmodule>
      <taskspawner>defaultTS</taskspawner>
      <initializerclass>init.AttentionInitializer</initializerclass>
</module>
```

As mentioned, initial tasks can be specified for modules. Every <task> in <initialtasks> must specify a name parameter, as well as a class name. Optionally the

declaration may specify `ticksperrun` – how often, in ticks, the task will be run. Additionally `<associatedmodule>` and parameters can be defined exactly as was done for other xml nodes.

Here is an example initial task declaration for a module:

```
<initialTasks>
      <task name="cueBackground">
          <tasktype>CueBackgroundTask</tasktype>
          <ticksperrun>15</ticksperrun>
          <param name="cueThreshold" type="double">0.4</param>
      </task>
</initialTasks>
```

<listeners>

In this xml node the inter-module communication (implemented using the observer pattern) of the agent is created. Specifically, those modules implementing a `ModuleListener` interface can be added to a specified module. The xml declaration for a listener specifies three things: first the type of Listener, then the transmitting module, then, finally, the listener module. For example, if the `PerceptualAssociativeMemory` module should have the `Workspace` module as a `PamListener` then there should be the following declaration:

```
<listener>
      <listenertype>edu.memphis.ccrg.lida.pam.PamListener</listenertype>
      <modulename>PerceptualAssociativeMemory</modulename>
      <listenername>Workspace</listenername>
</listener>
```

An easy way to read this is that the first module will transmit content to the second module. The "protocol" for the communication is the `listenertype` and the second module must implement this interface to receive the communication.

Bibliography

Baars, B. J. (1988). *A Cognitive Theory of Consciousness*. Cambridge: Cambridge University Press.

Baars, B. J. (2002). The conscious access hypothesis: origins and recent evidence. *Trends in Cognitive Science, 6*, 47–52.

Baars, B. J., & Franklin, S. (2003). How conscious experience and working memory interact. *Trends in Cognitive Science, 7*, 166–172.

Baars, B. J., & Franklin, S. (2009). Consciousness is computational: The LIDA model of Global Workspace Theory. *International Journal of Machine Consciousness, 1*(1), 23-32.

Franklin, S., Baars, B. J., Ramamurthy, U., & Ventura, M. (2005). The Role of Consciousness in Memory. *Brains, Minds and Media, 1*, 1–38.

Franklin, S., & Graesser, A. C. (1997). Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents *Intelligent Agents III* (pp. 21–35). Berlin: Springer Verlag.

Franklin, S., & Patterson, F. G. J. (2006). The LIDA Architecture: Adding New Modes of Learning to an Intelligent, Autonomous, Software Agent *IDPT-2006 Proceedings (Integrated Design and Process Technology)*: Society for Design and Process Science.

Hofstadter D., & Mitchell M., (1995). "The copycat project: A model of mental fluidity and analogy-making," in Hofstadter and the Fluid Analogies Research group: "Fluid Concepts and Creative Analogies," Basic Books, Chapter 5, pp. 205-267.

Jackson, J. V. (1987). "Idea for a Mind," ACM SIGART Bulletin Issue 101.

Kanerva, P. (1988). Sparse Distributed Memory., Cambridge MA: The MIT Press.

Maes, P. (1989). How to do the right thing. *Connection Science* 1:291-323.

Ramamurthy, U., Baars, B. J., D'Mello, Sidney K., & Franklin, S. (2006). LIDA: A Working Model of Cognition. In D. Fum, F. Del Missier & A. Stocco (Eds.), *Proceedings of the 7th International Conference on Cognitive Modeling* (pp. 244–249). Trieste: Edizioni Goliardiche.

Snaider, J., McCall, R., & Franklin, S. (2011). *The LIDA Framework as a General Tool for AGI*. Paper presented at The Fourth Conference on Artificial General Intelligence, Mountain View, California, USA.