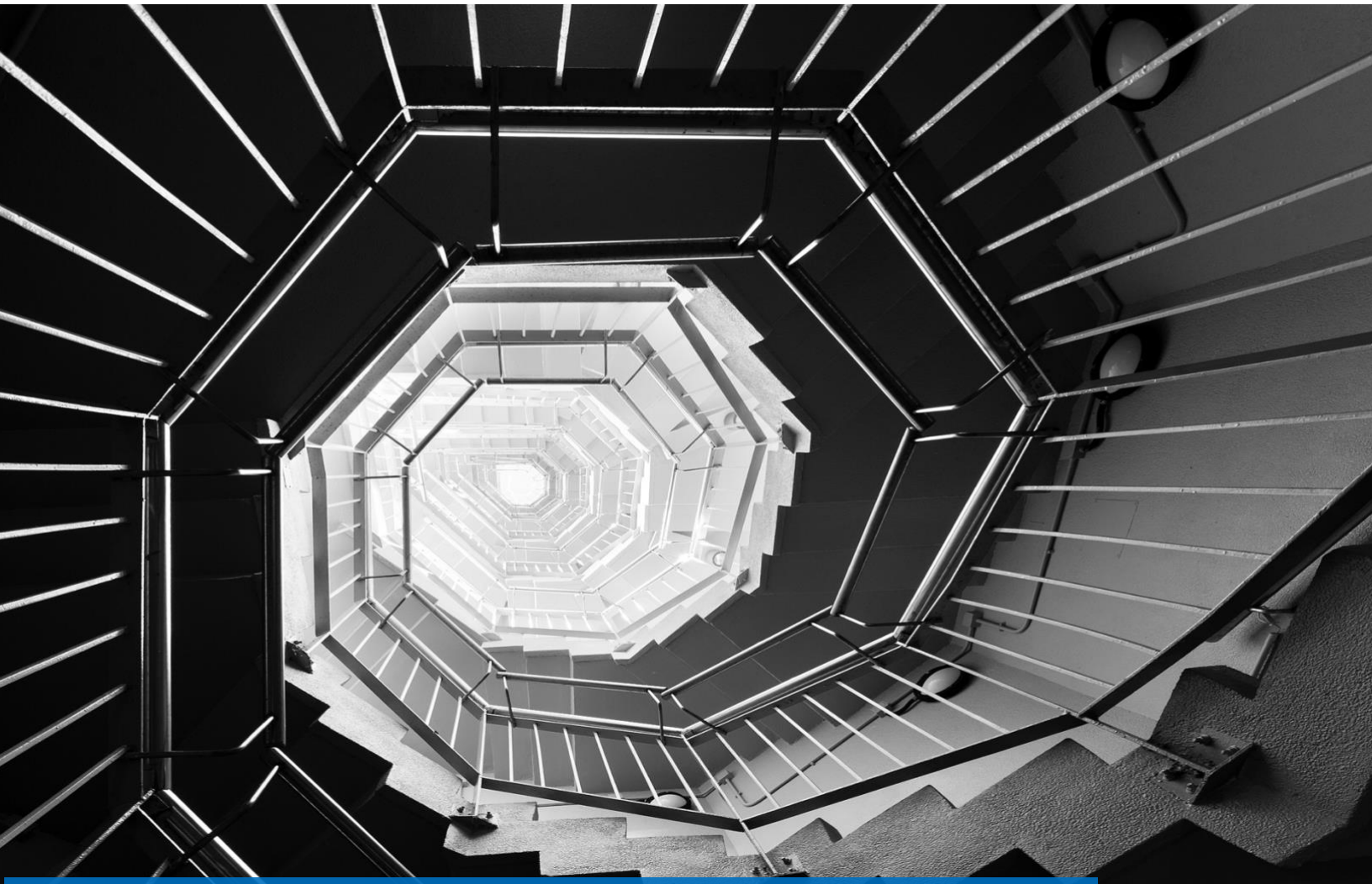

Building Cognitive Applications

Atlas System Guide

TABLE OF CONTENTS

Getting started

Welcome to the Atlas Cognitive System (Atlas) Guide.

Atlas offers multiple co-dependent components (capabilities) available through C++ APIs.

This document gives a general introduction to the Atlas Cognitive System which is broken down into the following sections:

- The environment.
- Components of the environment.
- Residents of the environment.
- Describing knowledge.
- Learning knowledge.
- Applying knowledge.

Atlas system documentation is intended for developers and subject matter experts seeking to improve cognitive applications, services, and experiences.

To learn how to develop cognitive simulations using Atlas, we recommend starting with this document “Atlas System Guide” followed by the “Atlas Process Guide” which is also part of the Atlas Development Kit.

ATLAS

System Overview

System overview

Atlas is part of a new variety of intelligent systems that combine different parts of cognitive computing.

Atlas is an integrated cognitive and immersive multi-agent system built from the ground up on a set of proprietary technologies that provides developers with the ability to intuitively configure their simulations to acquire, represent, access, modify and portray knowledge at scale.

In order to simplify the understanding of how Atlas works, we have devised a model analogous to a typical theatrical production.

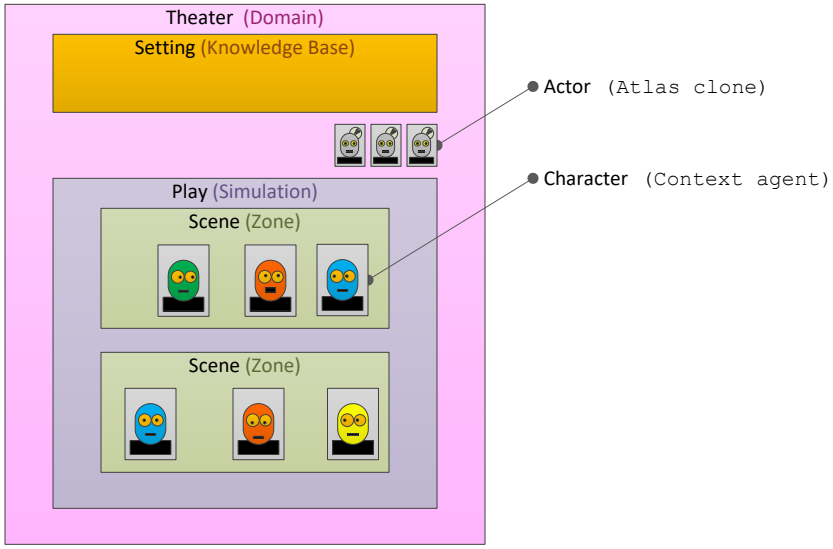

Fig 1. Mainting a domain is like managing a small theater production. The setting is a knowledge base. It dictates how the world behaves and a simple back story. The play is a simulation of the setting in a particular situation. This simulation involves characters, known as context agents, to be played by actors, known as Atlas clones. Each character is aware of the history and environment of the scenes to which they belong.

System structure

Atlas is a cognitive system comprised of multiple modules that interact with one another to simulate cognitive skills and processes.

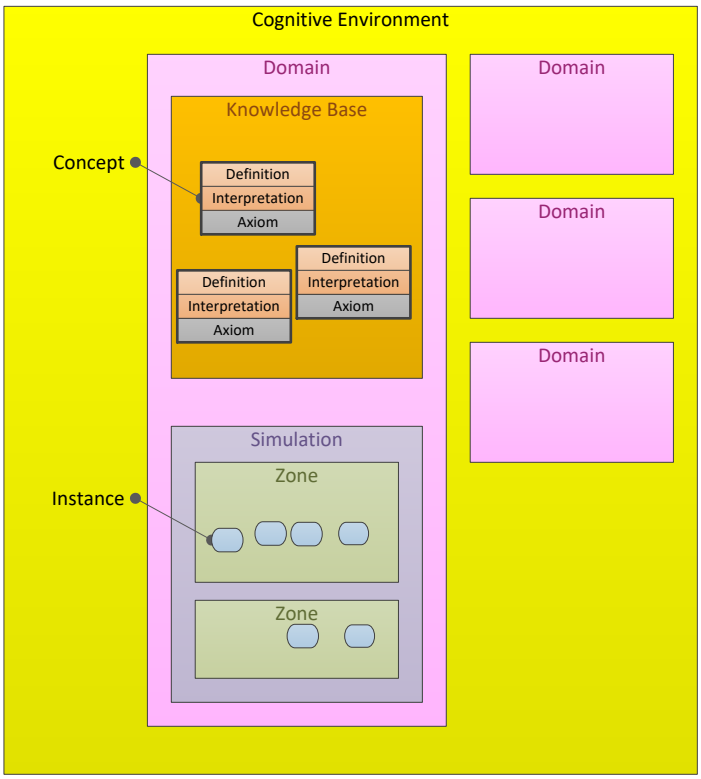

Fig 2. A cognitive environment with several domains: The expanded domain on the left exhibits a knowledge base containing three concepts and a simulation containing two zones. Each zone has been entered by multiple instances.

I. Cognitive Environment

Cognitive Environments (CEs) are layered, hierarchical spaces in which information is abstracted and structured gradually from raw data, represented as binary combinations of 1s and 0s, to higher representations called **concepts**.

Cognitive environments host domain-specific knowledge (Knowledge Base) and how this knowledge is applied in a situation (Simulation).

> *Cognitive simulations can be produced to emulate a physical world with multiple interactions.*

Cognitive environments are partitioned into logical spaces called **domains.**

A. Domain

Annotation:

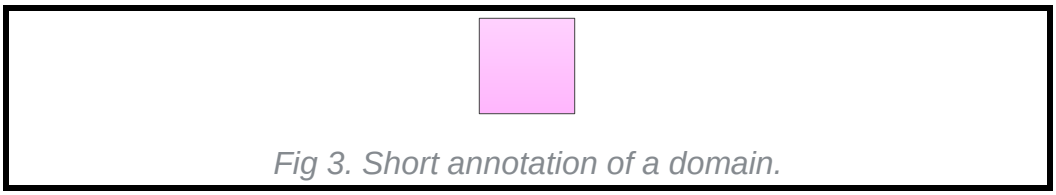

Fig 3. Short annotation of a domain.

i. Structure

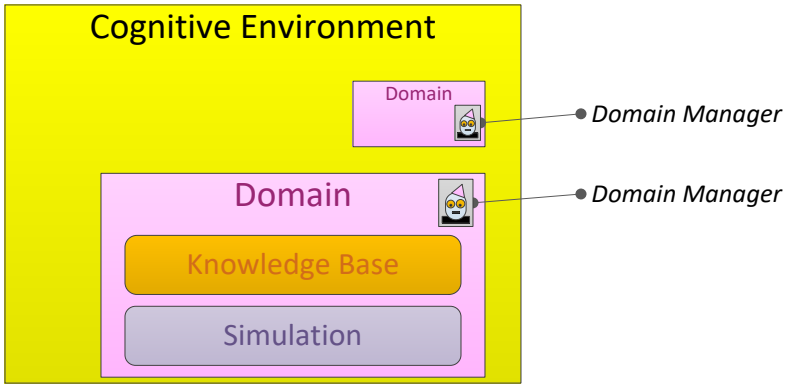

Fig 4. A cognitive environment hosting two domains: Each domain has its corresponding managing agent, depicted here with pink hats.

ii. Description

Domains offer a fencing mechanism for a knowledge base and its associated simulations.

Domains have two main parts:

- **Knowledge Base (KB):** for learning the rule set.
- **Knowledge Simulation (SIM)**: for applying the rules to an instanced situation.

The root domain in any cognitive environment is called "**Titan.**"

NOTE: Knowledge is not shared between domains but the Cognitive Exchange (Cx) allows communication between them.

iii. Management

Domains are managed by one or multiple administrators.

Domain Managers (DM) manage domains and control access to them.

DMs typically reside within the domains they are managing.

DMs intercept domain requests to allow or deny the following rules for their managed domains:

- Entering/Exiting a domain by a context agent.
- Creating/Destroying a simulation by a reference.
- Creating/Destroying a zone by a reference.
- Learning/Forgetting a concept.

Annotation:

Fig 5. Short annotation of a domain manager.

B. Zone

Annotation:

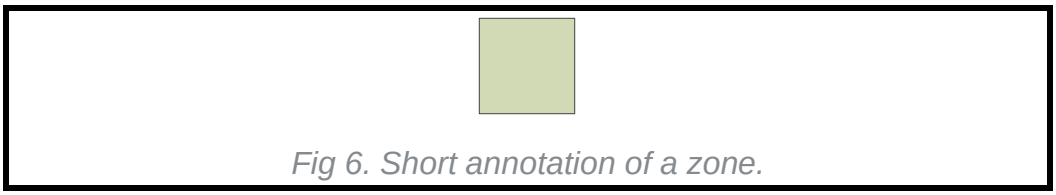

Fig 6. Short annotation of a zone.

i. Structure

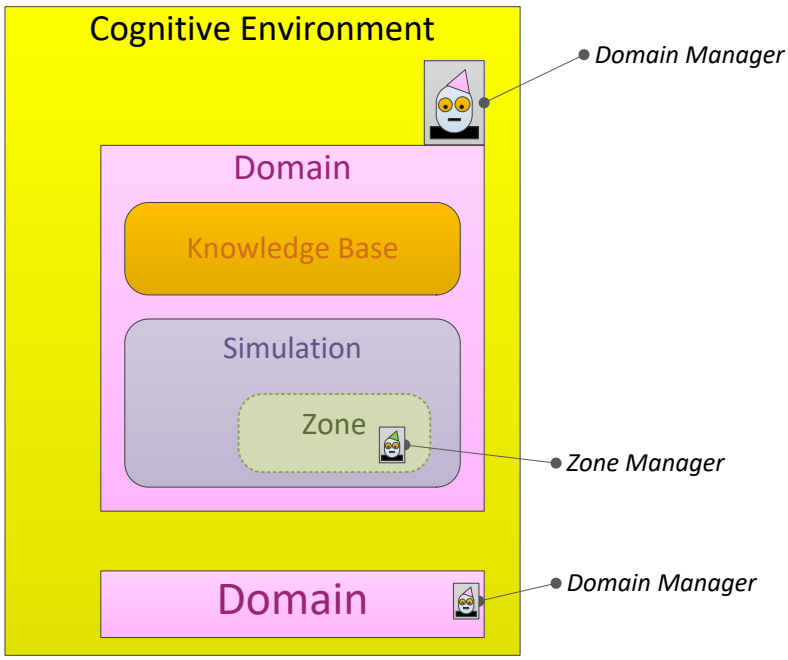

Fig 7. A cognitive environment with two domains: Each domain has its own manager, depicted with a pink hat. Zone managers are depicted with green hats.

ii. Description

Zones are non-exclusive visibility areas entered by agents for the purpose of interacting with a subset of the cognitive environment. They are analogous to chat rooms in a server.

All entities entering a domain must also enter at least one zone.

The root zone in the root domain, Titan, is "**Home**."

iii. Management

Zones are managed by one or multiple administrators.

Zone Managers (ZM) manage zones and control access to them. Similar to DMs, ZMs typically reside within the zones they are managing.

ZMs intercept zone requests to allow or deny the following rules for their managed zones:

- Managing entering/exiting by an agent.
- Communicating between two or more agents.
- Observing an agent's actions by another agent.

Annotation:

Fig 8. Short annotation of a zone manager.

II. Cognitive Agent

Annotation:

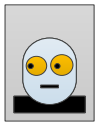

Fig 11. Short annotation of an agent.

i. Description

Agents are goal-oriented entities that enter an environment with the objective of interacting with information or other agents.

Agent activity inside a domain is limited to the zones in which they have permission.

Multiple agents can communicate if they reside in at least one common domain.

Atlas offers a resident agent capable of performing cognitive tasks on behalf of other agents described later in this document.

Atlas agent is ubiquitous and belongs to all domains.

ii. Management

Agents can trigger the following events:

- *Enter*/*Leave* a domain.
- *Enter*/*Leave* a zone.
- *Tell* other agents an expression (in natural language).
- *Ping* other agents a directive (directives discussed in later chapters).
- *Send* other agents a datagram message.
- Host concept references and instances (references discussed in later chapters).

A. Atlas Clone

See Atlas Process Reference for more detail.

B. Context Agent

See Atlas Process Reference for more detail.

III. Cognitive Container

In Atlas, a Knowledge Base (KB) is a repository for the underlying states, state changes and state relations representing an idea. This information is encapsulated into a repeatable, multi-layered, abstracting data structure called a knowledge concept, or simply **a concept.**

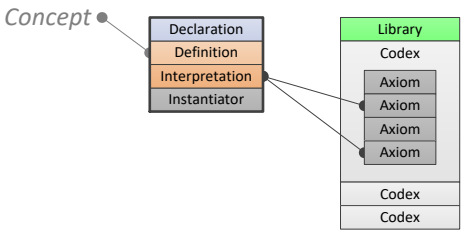

Fig 12. A concept and its components: A concept is encoded in a container that describes the name of the concept (Declaration), the general common knowledge of the concept (Definition), and a particular interpretation of the concept by one or multiple developers (Interpretation). The instantiator is the memory manager of the instances of the concept.

A. Concept Overview

Annotation:

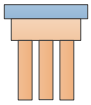

Fig 13. Short annotation of a concept.

Concepts are the elemental blocks for representing knowledge in Atlas.

Concepts can be combined with other concepts to form complex ideas.

Concepts are organized into a four-layer structure.

- Concept label (Declaration)
- Concept representation in natural language (Definition)
- Concept representation in code (Interpretation)
- Concept representation in data (Instance)

B. Concept Learning -Do, Be, Have

i. Definition:

Definitions give a broad idea of what a concept can be, how it behaves, and how it relates to other concepts.

All concepts must be defined.

A concept is defined in Atlas by answering the following questions:

- What is it? What else can I call it?
  - This question hints to the name(s) of the concept.
- What can it be?
  - This question hints to the state(s) of the concept.
- What can it do?
  - This question hints to the action(s) of the concept.
- How does it relate to other concepts?
  - This question hints to the relation(s) of the concept.

The answers to the above questions are provided in the form of a plain-text file that describes the definition of one concept and references definitions of other concepts.

The definition of a concept is learned by Atlas through its definition file.

The process of learning occurs when the definition is added to the knowledge base.

Annotation:

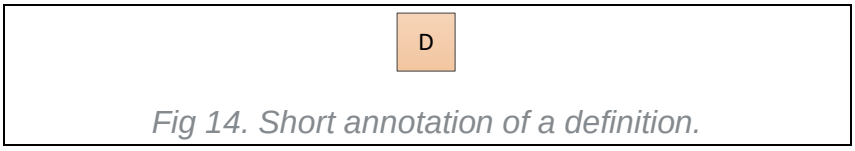

Fig 14. Short annotation of a definition.

Example:

Let's take a look at an example:

Consider a simple definition of a desk lamp which is comprised of a lamp shade, lamp base and a bulb, where the bulb is further broken down into a filament of a particular color as shown in the figure below.

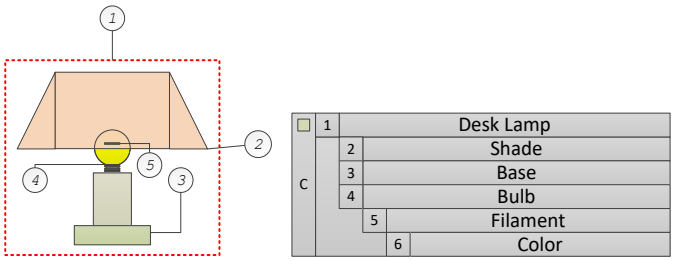

Fig 15. Desk lamp concept with its subordinate concepts.

The corresponding ATLAS top-level definition file for this lamp would be:

```
[Base/MyLamp]                    # Unique identifier of the definition
 Info:                           # Info block
 Name: My Lamp                   # Official name of this definition
 Author: Me                      # Author of this definition
 Version: 1.0                    # Version of this definition
 Declaration: Desk Lamp          # Name of the declaration that we are
                                     defining
 Synonyms: {Light, Furniture}    # Soft synonyms of this definition


 Be:                             # BE block of the definition
 On:                             # Lamp can be on
 Off:                            # Lamp can be off
 Broken:                         # Lamp can be broken


S.Do:
 Turn on:                        # Lamp can turn on (lamp is the subject)
 Turn off:                       # Lamp can turn off (lamp is the subject)
O.Do:
 Turn on:                        # Lamp can be turned on (lamp is the object)
 Turn off:                       # Lamp can be turned off (lamp is the object)


Have:
 Shade: Base/MyLampShade         # Lamp has a shade defined in Base/MyLampShade
 Base: Base/MyLampBase           # Lamp has a base defined in Base/MyLampBase
 Bulb: Base/MyLightBulb          # Lamp has a bulb defined in Base/MyLampBulb
```

Definition parameters:

Bracket [ ] Identifiers:

All definition files must start with a unique identifier for the definition that will be added to the knowledge base.

Identifiers are delimited by square brackets, [ ], as shown in the first line of the file.

The subsequent blocks, shown by colored headers above, are the main parts of the definition. These blocks can come in any random order.

The **bolded and underlined** text represents the main keywords of the definition, whereas text in *italic* is the parametric text that is definition dependent and is provided by the definition's author.

Info:

The **Info** block holds the metadata about the definition's official name, authorship, and version. It is represented by the tags **Name**, **Author**, and **Version** respectively.

Declaration:

The **Declaration** block is the linguistic representation of the definition in Atlas' lexicon, where the declaration field is the official linguistic label for the definition (i.e. Desk Lamp). This field answers the question "What is it?"

The **Synonyms** field answers the question "What else can I call it?" as it indicates all the different ways this lamp could be invoked in a sentence – this field is loosely related to the true synonym of the definition. Due to this nuance, the labels in this field are also known as *soft synonyms*.

Be:

The **Be** block answers the question "*What can it be?*" All fields in this block represent one possible state of the defined concept (e.g. On, Off, Broken).

S.Do/O.Do:

The **Do** block answers the question "*What can it do*?" Fields in this block represent functions of the defined concept (e.g. Turn On, Turn Off).

The prefix **S.** in **S.Do** indicates that the defined concept is the subject of the functions advertised, i.e., what can this concept do to others?

The prefix **O.** in **O.Do** indicates that defined concept is the object of the functions advertised, i.e., what can others do to this concept?

Have:

The **Have** block answers the question "*What is it made of?*" or "*How does it relate to other concepts?*" Each field in this block represents one link between this definition and other definitions in the knowledge base (e.g. *Shade, Base, Bulb)*. This relationship representation is only one-layer deep. Deeper layers of relationships should be defined in the referenced definitions of these fields recursively. The values of these fields are the identifiers of the subordinate concepts (e.g. *Base/MyLampShade*). The highest parent definition in a hierarchy of concepts is called a *Top Concept*.

All definitions should be stored under the subtree ***./KB/Concept*** with the file extension ***.D***

A visual representation of the lamp we have just defined would be the diagram below:

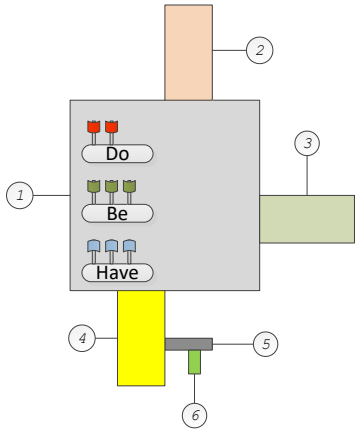

*Fig 16. Desk lamp (1) has two **Do** methods, three **Be** methods, and 3 **Have** methods corresponding to the definition we gave Atlas. The desk lamp also references other definitions: shade (2), base (3), and bulb (4). The bulb definition references a filament (5), and the filament references a color (6).*

ii. Interpretation

Once a definition is described, Atlas needs to know how to simulate it. As the name suggests, the interpretation of a definition interprets what might have been conveyed.

In our example of a desk lamp we have described to Atlas that a lamp can "*Turn ON"* and "*Turn OFF*." Next we need to describe "*Turn ON*" in terms that a computer understands.

Describing *"Turn ON"* means we have to reach a common language that Atlas understands beyond just the label *"Turn ON."* In this case, we use a programming method that Atlas fully understands and holds as self-evident truth. Code, C++ in this case, is axiomatic; it no longer requires further interpretation.

In Atlas cognitive environments, all methods that represent states, functions, or relationships that are defined in code are called **axioms.**

Interpretations are added to the knowledge base through a plain-text file that describes the interpretation of a particular definition of one concept.

Annotation:

Fig 17. Short annotation of an interpretation.

Example:

Let's look again at our desk lamp example from figure 15:

The corresponding ATLAS top-level interpretation file for this lamp would be:

```
[Base/MyInterpretedLamp]     # Unique identifier of the interpretation

Info:                                # Info block
Name: My Lamp Interpretation         # Official name of this interpretation
Author: Me                           # Author of this interpretation
Version: 1.0                         # Version of this interpretation


Links:                               # Links (to axioms) block
   FLamp: {MyLampLibrary, FunctioningLamp, Desk Lamp}
   NFLamp: {MyLampLibrary, NonFunctioningLamp, Desk Lamp}


Definition: Base/MyLamp              # Definition we are interpreting
   FLamp: 0                          # Axiom (FunctioningLamp)
   NFLamp: 0                         # Axiom (NonFunctioningLamp)


Have:
   Shade: Base/MyInterpretedShade # The lamp shade interpretation
   Base: Base/MyInterpretedBase   # The lamp base interpretation
   Bulb: Base/MyInterpretedBulb   # The lamp bulb interpretation
```

Interpretation parameters:

Bracket [ ] Identifiers:

All interpretations files must start with a unique identifier for the interpretation that will be added to the knowledge base. This identifier is delimited by square brackets, [ ], as shown in the first line of the file.

The subsequent blocks, shown by colored headers above, are the main parts of the interpretation. These blocks can come in any random order.

The **bolded and underlined** text represents the main keywords of the interpretation, whereas text in *italic* is the parametric text that is interpretation dependent and is provided by the interpretation's author.

Info:

The **Info** block holds the metadata about the interpretation's official name, authorship, and version. It is represented by the tags **Name**, **Author**, and **Version** respectively.

Links:

The **Links** block is the resource locator of the axioms to be attached to the knowledge base on behalf of this interpretation.

Each subsequent field under this block represents one link to a dynamic library (DLL, .so) that will be handling some, or all, of the functionality of the advertised interpretation. The fields are further broken down into the following format:

<Link Name>: <Library Name, Codex Name, Axiom Name>

Link Name is the name of the link that will be later referenced within this interpretation.

Library Name is the name of the library from which the axiom will be loaded. The library will be loaded from the *./KB/Codex* folder based on the name specified here.

Codex Name is the name of the codex that will be handled by the library's axiom instantiator.

Codices are a convenient way of binding multiple correlated axioms together.

It is possible to put all axioms of a knowledge base under the same codex in one library, but we recommend splitting them apart as a good software development practice.

Axiom Name is the name of the axiom that will be mapped to this link.

Definition:

The **Definition** block describes the definition this interpretation is handling.

All subsequent fields under this block represent the links to the axioms that collectively interpret the referenced definition.

Multiple independently developed libraries can collaborate on interpreting the same definition.

The relationship between the interpretation and the code library is not one-to-one.

The same library can interpret multiple definitions and the same definition can be interpreted by multiple libraries.

In the example above, we have two axioms of the same name (*Desk Lamp*), but from different codices (*FunctioningLamp*, *NonFunctioningLamp*) of the same library (*MyLampLibrary*) collaborating on interpreting our *Base Lamp* definition.

The second parameter of these fields (0) is the optional parameter that is sent to the axiom during the *KB_Attach* stage.

This parameter is sent as a string and is useful if the axiom is polymorphic, common between multiple interpretations, has internal parameters that are dependent on the interpretation it is handling, or has a proprietary handshake needed to initialize (See *KB_Attach*).

Have:

Like its definition counterpart, the interpretation's **Have** block references further interpretations of the sub-concepts described in the original definition.

All interpretations should be stored under the subtree ***./KB/Concept*** with the file extension ***.I***

A visual representation of the lamp we have just interpreted would be the diagram below:

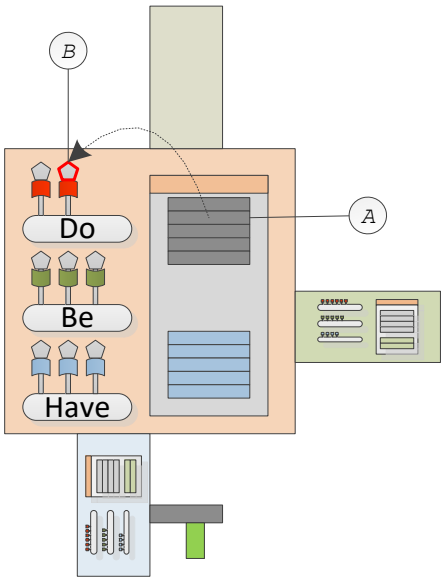

*Fig 18. A Desk Lamp interpretation is attaching to the Desk Lamp definition. In this example "Turn Off" (B) is being interpreted by the second axiom (A) referenced by the interpretation. This mapping of A to B is performed during the attach stage of the knowledge base through the method **C3**. (See **C3**:**Do**)*

iii. Instance

Once a concept is learned, its application in the cognitive space is performed in the simulation. The interpretation is tasked with supplying examples of its knowledge to the simulation; these examples are known as simulation instances, or simply **instances**. An instance can be a singular block of memory or it can have multiple elements (e.g. arrays, lists, streams). Atlas references an instance by supplying the hosting interpretation with two values:

- One value for instance data.
- One value for element data.

The notation for this ordered pair is **{I: *instance_ID*, E: *element_ID*}**. When no element ID is referenced, then the pair can be reduced to **{I: *instance_ID*}** where *element_ID* is understood by Atlas to be 0.

Example:

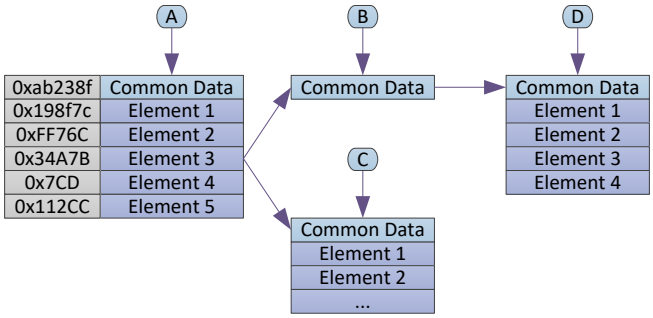

Fig 19. A hierarchy of simulation: Instance A is the top concept instance with 5 elements. Element {I:0xab238f, E:0x34A7B} of the top instance has two sub-instances (B, C) as defined by the concept-tree, where instance B has no elements but has a sub-instance D. Instance C has an undefined number of elements (i.e. stream).

iv. Axiom

Axioms

In this release, an axiom is an object-code interpretation of the concept behavior that is implemented as a C++ class inherited from the class *TitanAxiom*. An axiom can have one or multiple methods within it. It is by definition an OOP class.

Axiom Libraries:

So far, the interpretation has told Atlas that in order to understand the claims of the definition, it needs to reference axioms. These axioms come in packages we call libraries.

A library in Atlas is a shared object code that can be:

- A 64-bit Dynamic Linked Library (.DLL) for the Windows operating system.
- A 64-bit Shared Object (.so) for the Linux operating system.

The axioms are generated by using the SDK that accompanies this document. After it has been compiled, the generated axioms library must be stored in the *./KB/Codex* folder.

Once in the codex folder, the interpretation can reference the library in the *Links* block described in the previous section.

In the interpretation file example above, we have pointed out that there are two main axioms simulating the desk lamp, one for a functioning lamp and one for a non-functioning lamp.

- FLamp: {MyLampLibrary, FunctioningLamp, Desk Lamp}
- NFLamp: {MyLampLibrary, NonFunctioningLamp, Desk Lamp}

The functioning lamp axiom is referenced in *MyLampLibrary* under codex *FunctionalMap*. The non-functioning lamp is referenced in the same library, *MyLampLibrary*, but under a different codex *NonFunctionalLamp*.

Axiom Codices:
A shared object or library typically has a flat hierarchy of methods. Atlas introduces the concept of a codex.

A codex is a virtual folder within a library that groups axioms of common functionality or behavior together.

When Atlas requests an axiom method from library, it references the codex first, then the axiom's name. It is left to the axiom developer to decide how to group their codices within their own library. If the links in the interpretation reach their intended axiom methods, then Atlas' requirements will be satisfied.

C. Concept Loading -Manifest

Annotation:

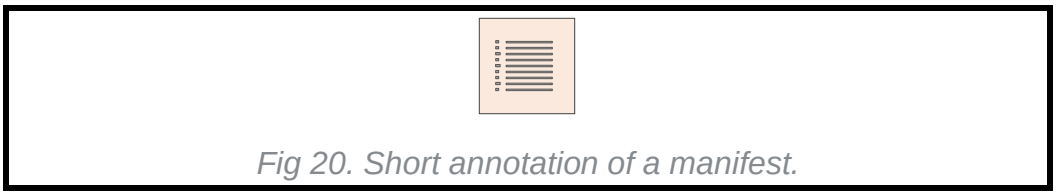

Fig 20. Short annotation of a manifest.

Once a concept's definition and interpretation have been described in .D and .I text files respectively, we need to load them into the knowledge base. All concepts are loaded into an existing knowledge base domain (See *Domain*) through a text file we call a *manifest*. An example manifest file looks like this:

```
/Learn "Office Lamp" "MyInterpretedLamp" "Base/MyInterpretedLamp"
```

/Learn:

The manifest command **/Learn** teaches Atlas about a concept, where:

- The first parameter is the name of the new concept we will interpret (*Office Lamp*).
- The second parameter is the interpretation path on disk under ./KB/Concept. The file loaded will be *<interpretation_path>.I*. In this case, *MyInterpretedLamp* is expected to be found in *./KB/Concept/MyInterpretedLamp.I*.
- The third parameter is optional but can be useful if the path is unknown but the Identifier is known. In this case, we will use the identifier we gave to the desk lamp interpretation "Base/MyInterpretedLamp".

Note that the definition file is pulled directly from the interpretation file upon learning. There is no need to explicitly call for a */Learn* on the sub-components of the Desk Lamp as they are pulled automatically from the interpretation during the learning process. Essentially all the sub-concepts of a learned concept will also be learned to satisfy a simulation.

D. Concept Instantiation -Script

Annotation:

Fig 21. Short annotation of a script.

Once a concept of an office lamp has been learned, instantiating this concept into the cognitive space can be done in three ways:

- Through a script file.
- Through an agent expression (See *Method AB*).
- Through an API calls.

Script files are a convenient way of expressing multiple statements in a single pass. Any complete natural language sentence (expression) can be put in the script as an action, a statement, or a query.

Expressions are separated by a new line character or a period.

There are also keyword-specific expressions such as the */Create* command:

```
/Create "Office Lamp" "Lamp 01" 1 "Model 003"
/Create "Office Lamp" "Lamp 02" 1 "Model 263"
```

/Create:

The script command **/Create** instantiates an example of a learned concept, where:

- The first parameter is the name of the concept we will instantiate (*Office Lamp*).
- The second parameter is the official name of the instance (*Lamp 01*).
- The third parameter is the number of elements to be allocated within the instance (1).
- The fourth parameter is optional, but when available, it is sent to the instance manager as an initialization parameter. In this case we are loading a particular lamp model (*Model 003*) that the “Office Lamp” is expected to handle. This parameter allows multiple versions of a concept to be represented by a common axiom. (**See *I1:Create*, *I2:_Create***)

Note that there is no need to create the sub-components of the lamp (i.e. Shade, Base, etc.); they are expected to be automatically instantiated by the axiom managing the top

concept. Essentially, each instance is responsible for creating its sub-concept instances during instantiation (I1, I2).

IV. Cognitive Exchange

Atlas Cognitive Exchange also known as the Atlas Cognitive Platform is the ecosystem where multiple cognitive agents can semantically search for information and collaborate on solutions through recursive interaction and delegation. We will discuss Atlas platform services in more detail in future documents.