The Core Model Trail

In this trail, you will learn some core concepts used within CST in order to understand the basic building blocks required for building your own Cognitive Architecture using CST. Our emphasis will be in understanding the core classes used for building the most elementary kind of Cognitive Architecture. Every cognitive architecture built upon CST will follow a set of guidelines, using just a few classes as support. More sophisticated architectures using additional CST classes might be learned further, following the other trails in our Tutorials page. Once you are able to understand these core concepts and how they are meant to be used using CST classes, you will be able to start building your first Cognitive Architecture using CST.

Basic Notions

The CST toolkit is designed using some basic notions coming from the work of Hofstadter & Mitchell (1994). In their historical "Copycat Architecture", Hofstadter & Mitchell defined their view of a cognitive architecture as based on the work of many micro-agents called codelets, which will be responsible for all kinds of actions in the architecture. Codelets are small pieces of non-blocking code, each of them executing a well defined and simple task. The idea of a codelet is of a piece of code which ideally shall be executed continuously and cyclically, time after time, being responsible for the behavior of a system's independent component running in parallel. The notion of codelet was introduced originally by Hofstadter and Mitchell (1994) and further enhanced by Franklin (1998). The CST architecture is codelet oriented, since all main cognitive functions are implemented as codelets. This means that from a conceptual point of view, any CST-implemented system is a fully parallel asynchronous multi-agent system, where each agent is modeled by a codelet. CST's codelets are implemented much in the same manner as in the LIDA cognitive architecture and largely correspond to the special-purpose processes described in Baar's Global Workspace Theory (Baars 1988). Nevertheless, for the system to work, a kind of coordination must exist among codelets, forming coalitions which by means of a coordinated interaction, are able to implement the cognitive functions ascribed to the architecture. This coordination constraint imposes special restrictions while implementing codelets in a serial computer. In a real parallel system, a codelet would simply be called in a loop, being responsible to implement the behavior of a parallel component. In a serial system like a computer, the overall system might have to share the CPU with its many components. In CST, we use multi-thread to implement this implicit parallelism. Each codelet runs in its own thread.

Figure 1 below illustrates the general conception of a cognitive architecture, according to CST.

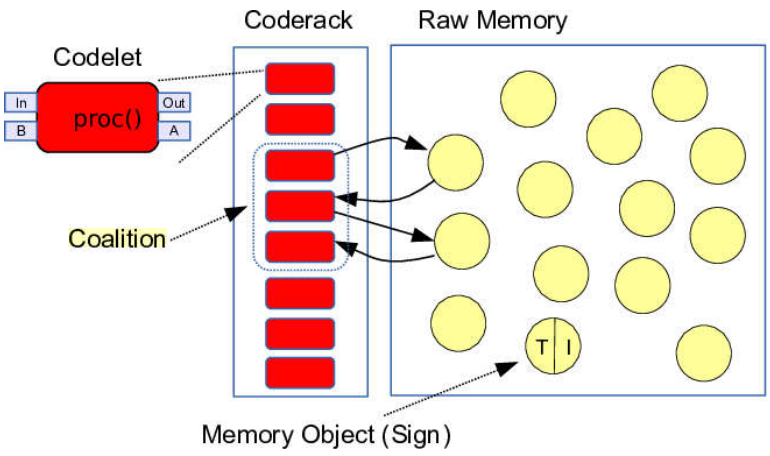

Figure 1 - The CST Core

A ***Mind*** is modelled in terms of a ***RawMemory***, where many ***MemoryObjects*** are stored, and a ***Coderack***, containing a set of ***Codelets***. ***Codelets*** might have ***MemoryObjects*** as input, and also other ***MemoryObjects*** as outputs. Different ***Codelets*** might form ***Coalitions***, by sharing the ***MemoryObjects*** they use as input and output.

A codelet has two main inputs (which are characterized as In and B in the figure), a local input (In) and a global input (B). The local input is used for the codelet to get information from memory objects, which are available at the Raw Memory. The global input is used for the codelet to get information from the Global Workspace mechanism (Baars & Franklin 2007; 2009). The information coming from the Global Workspace is variable at each instant of time, and usually is related to a summary, an executive filter which select the most relevant piece of information available in memory at each timestep. The two outputs of a codelet are a standard output, which is used to change or create new information in the Raw Memory, and the activation level, which indicates the relevance of the information provided at the output, and is used by the Global Workspace mechanism in order to select information to be destined to the global workspace. Using this Core, the CST toolkit provides different kinds of codelets to perform most of the cognitive functions available at a cognitive architecture, as indicated in figure 2.

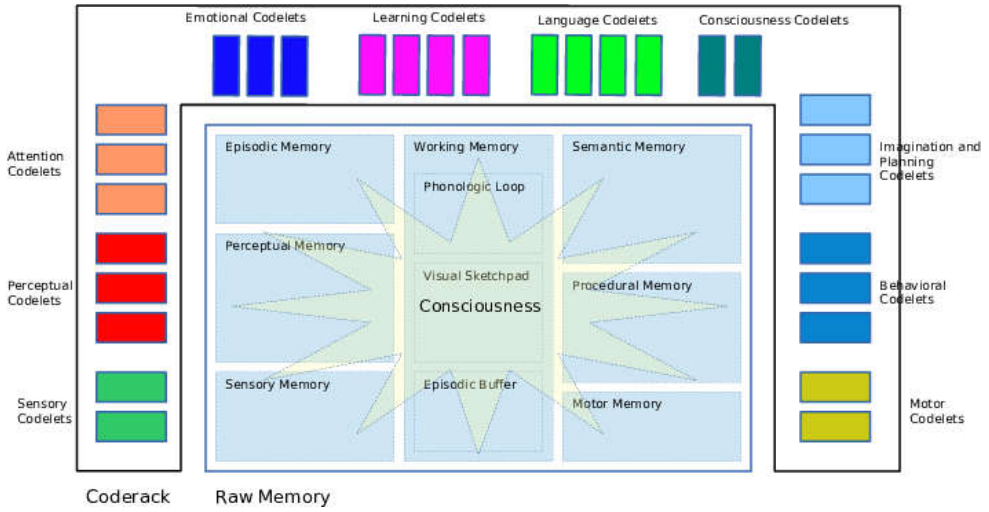

Figure 2 - The CST Overall Architecture: Codelets

Also, memory objects are scattered among many different kinds of memories. The Raw Memory is so split into many different memory systems, which are used to store and access different kinds of knowledge. Using the available codelets, different cognitive architectures, using different strategies for perception, attention, learning, planning and behavior generation can be composed in order to perform the role necessary to address a specific control problem. These codelets are constructed according to different techniques in intelligent systems, like neural networks, fuzzy systems, evolutionary computation, rule-based systems, Bayesian networks, etc., which are integrated into a whole control and monitoring system. The definition and choice of a particular cognitive architecture is constructed using a composition of different kinds of codelets, according to the control problem under analysis. Depending on the problem to be addressed, different strategies might be necessary or useful, depending on the problem constraints.

The Main Classes

The main core classes in the CST are presented in the UML diagram of figure 3. Each of them corresponds to one of the concepts presented earlier.

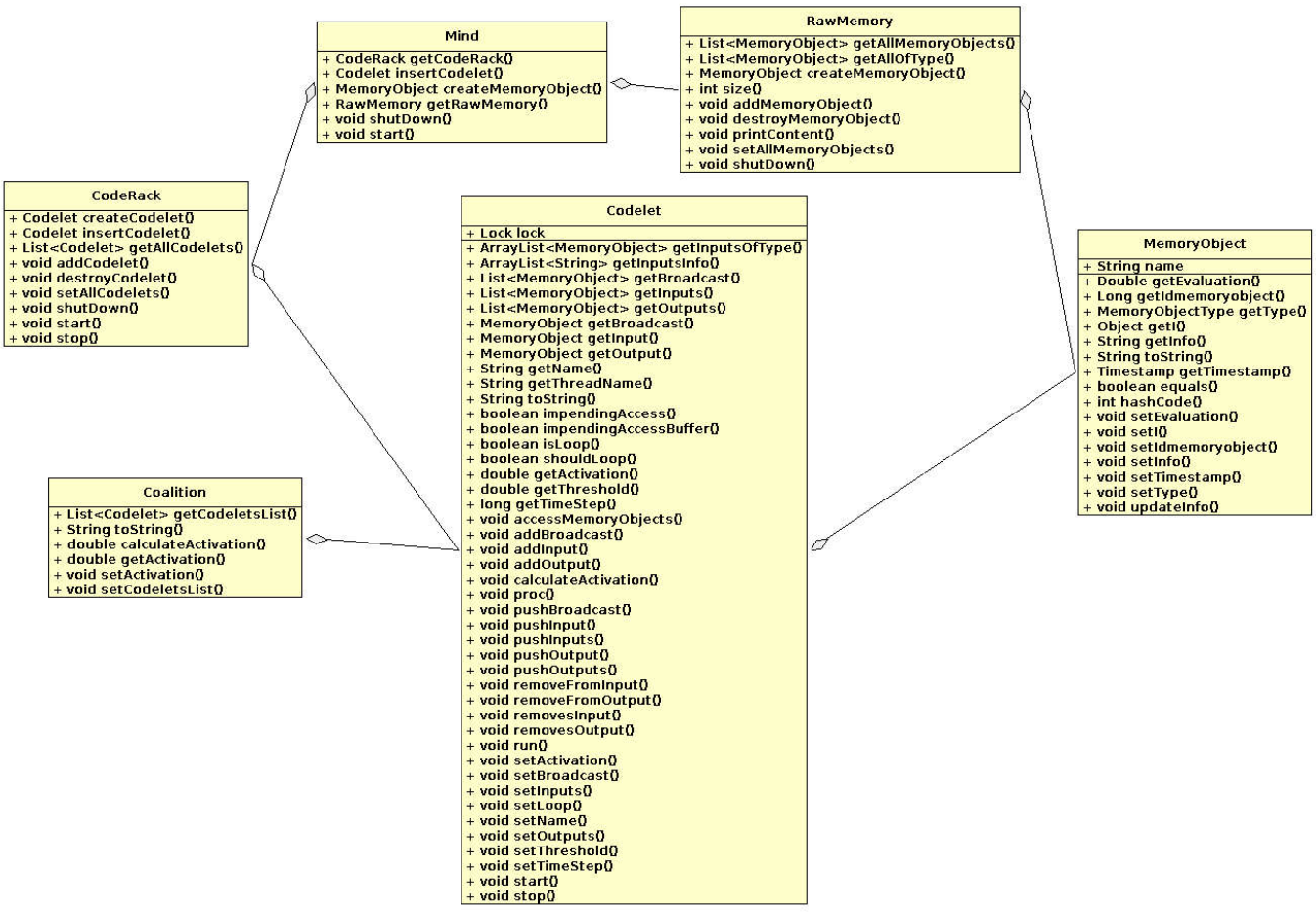

Figure 3 - The Main Core Classes in CST

In order to build a cognitive architecture, the designer might rely basically on those classes. It is possible to build a complete Cognitive Architecture simply by using these main core classes. Basically, the designer might have to create different classes inheriting from the ***Codelet*** class, where each codelet is responsible for a small fragment of processing. These codelets will operate on **MemoryObjects**, affecting other **MemoryObjects**. The designer might either create an instance of the ***Mind*** class, or create a new class inheriting the ***Mind*** Class. By doing this, it is creating both a ***Coderack*** and a ***RawMemory***. Using the **Mind** class (or a subclass), the designer might then create all the codelets and

MemoryObjects for his/her cognitive architecture and insert them within the ***Mind***. After that, the designer might call the **start**() method from ***Mind*** and the Cognitive Architecture should be operational.

The WS3DApp Application

In this trail, we will be building the WS3DApp, a very simple Cognitive Architecture to control de Robot in the WS3D Virtual Environment. More information about the Virtual Environment, and the source code necessary to build this application can be found in the WS3D Example page. A sketch of the architecture is given in Figura 4 below:

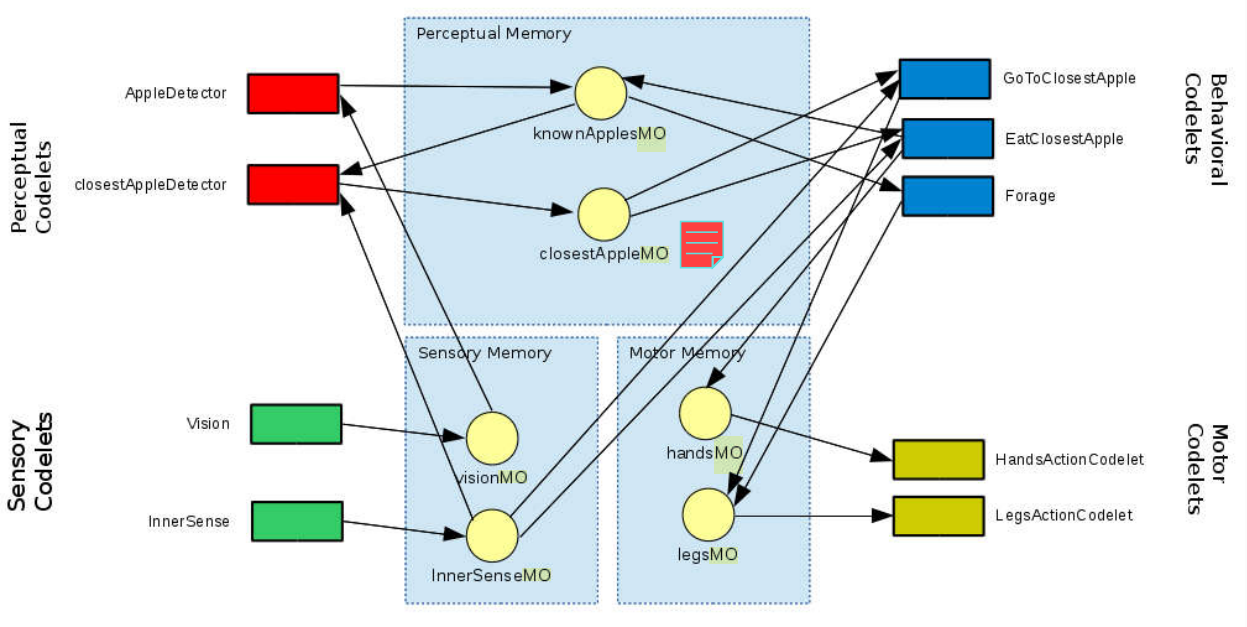

Figure 4 - The Demo Cognitive Architecture for the WS3D Robot Control

This demo architecture is very simple. We have two sensory codelets: ***Vision*** and ***InnerSense***. These sensory codelets produce the ***VisionMO*** and the ***InnerSenseMO***, the first a memory object containing a list of objects being seen by the creature in its line of sight, and the second a memory object containing self-referencial information, like the agent position, velocity, and other information. We have then two perceptual codelets processing such information. The first perceptual codelet is the ***AppleDetector*** codelet. This codelet is responsible for updating the ***KnownApplesMO*** memory object, which is a list containing all apples already seen, but not eaten. The ***ClosestAppleDetector*** codelet, then, using the information from both ***KnownApplesMO*** and ***InnerSenseMO*** can then generate the ***ClosestAppleMO*** memory object, containing the information regarding the closest apple. Among the behavioral codelets, we have codelets responsible for three different possible behaviors: the ***GoToClosestApple*** codelet, the ***EatClosestApple*** codelet and the ***Forage*** codelet. The ***Forage*** codelet will act only when the list of ***KnowApplesMO*** is empty. It will create a random walk, by choosing a random destination and providing the ***LegsMO*** with the information with this location. The ***GoToClosestApple*** codelet will act only when there is something at the ***ClosestAppleMO***, and the ***InnerSenseMO*** detects the Robot is far from the closest apple. It then generates information at the ***LegsMO*** in order to the Robot to proceed to the location of the closest apple. The ***EatClosestApple*** acts only when there is something at the ***ClosestAppleMO*** and the ***InnerSenseMO*** detects that the Robots position is in a reachable vicinity of the closest apple. It then order the ***HandsMO*** memory object, to give an order for the Robot to eat the apple. Finally, the motor codelets process the orders given by ***HandsMO*** and ***LegsMO***. The ***HandsAction*** codelet check if it is there an order to eat an apple, and sends it to the Virtual Environment, and the ***LegsActionCodelet*** checks if there is an order to go to some position and sends it to the Virtual Environment.

Let's take a look on the tree of classes for the WS3DApp application, given in figure 5:

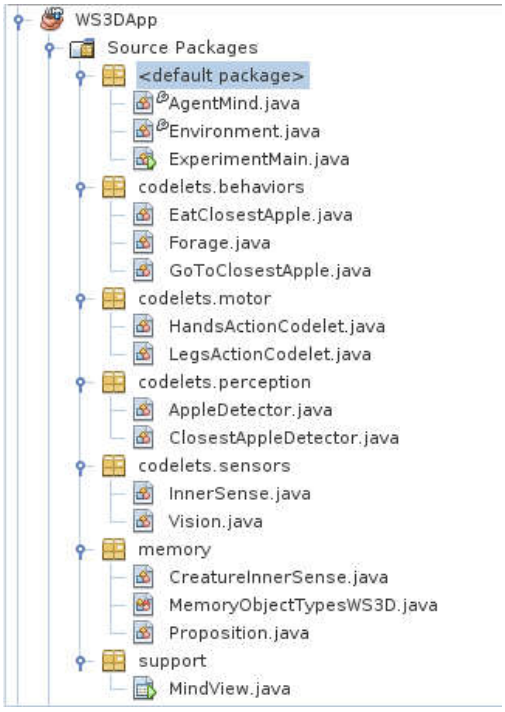

Figure 5 - The Classes Tree for the WS3DApp Application

We have basically a set of packages where we organize our classes in terms of their functionality. In the default package, we have the ***AgentMind*** class, where all the architecture is defined, the ***Environment*** class, responsible for populating the WS3D environment with a robot and some apples (and increasing new apples from time to time), and the ***ExperimentMain*** class, which is the application main class. Then we have a hierarcy of codelets packages, where the different codelets are defined, a memory package with auxiliary classes used within MemoryObjects and a support package, with supporting classes (in this case, the ***MindView*** class, which prints out what is within each MemoryObject in the architecture.

The ExperimentMain Class

Let's take a look on the code for the ExperimentMain class (use the sliding bar at the right for viewing the full code):

```
import java.util.logging.Logger;

public class ExperimentMain {

        public Logger logger = Logger.getLogger(ExperimentMain.class.getName());

        public ExperimentMain() {
                // Create Environment
                Environment env=new Environment(); //Creates only a creature and some apples
                AgentMind a = new AgentMind(env);  // Creates the Agent Mind and start it
        }

    public static void main(String[] args) {
        ExperimentMain em = new ExperimentMain();
```

This code is very simple. Basically, on the main method the application is initialized, and during the creation of the ExperimentMain object, we create an instance of the Environment class and an instance of the AgentMind class.

The Environment Class

The code for the Environment class can be viewed below:

```
import ws3dproxy.CommandExecException;
import ws3dproxy.WS3DProxy;
import ws3dproxy.model.Creature;
import ws3dproxy.model.World;

public class Environment {

    public String host="localhost";
    public int port = 4011;
    public String robotID="r0";
```

```
    public Creature c = null;

    public Environment() {
```

In this code, we use classes from the *WS3DProxy* to get access to the WS3D Virtual Environment, reset the ***World*** if it was already initialized from a previous simulation (this sets up the Environment back to ground), create 3 apples on selected locations and create a ~~new~~ creature. Then the command **c.start()** triggers the beginning of the simulation at WS3D.

The AgentMind Class

The code for the AgentMind class can be seen below:

```
import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import br.unicamp.cst.core.entities.Mind;
import codelets.behaviors.EatClosestApple;
import codelets.behaviors.Forage;
import codelets.behaviors.GoToClosestApple;
import codelets.motor.HandsActionCodelet;
import codelets.motor.LegsActionCodelet;
import codelets.perception.AppleDetector;
import codelets.perception.ClosestAppleDetector;
import codelets.sensors.InnerSense;
import codelets.sensors.Vision;
import java.util.ArrayList;
import java.util.Collections;
```

It is in this class that the overall Cognitive Architecture is set up. Initially, pay attention that the ***AgentMind*** class extends the ***Mind*** class from CST. This means it becomes automatically equiped with a ***RawMemory*** and a ***Coderack***. Then, we start building up the architecture. We first declare all ***MemoryObjects*** and start creating them one by one. The ***String*** being passed to the **createMemoryObject()** method (it is a method of the ***Mind*** class, which ***AgentMind*** inherits) is tag (a name), which uniquely identifies the MemoryObject, and can be used to recover access to it from the ***RawMemory***, if it is necessary. After creating all ***MemoryObjects***, we then create the ***MindView***, which is a window to trace the contents of all ***MemoryObjects***, and insert in it all ***MemoryObjects*** we want to trace. Then we start creating all the codelets in the architecture. We start with the sensor codelets, going through actuator codelets, perception codelets, and behavior codelets. Finally, after the architecture is set up, we **start()** the cognitive cycle.

The Memory Objects

MemoryObjects are just wrappers for Java objects which hold the valuable information to be used by codelets and to be written by them. In our case, we have the following mapping:

- legsMO = String
- handsMO = String
- visionMO = ArrayList
- innerSenseMO = CreatureInnerSense
- closestAppleMO = Thing
- knownApplesMO = ArrayList

The class ***Thing*** is a class from the WS3DProxy package, which provides the connection of our application to the WS3D Virtual Environment. The class ***CreatureInnerSense*** is the only memory class which has to be provided by our app.

The CreatureInnerSense Code

Basically, the CreatureInnerSense is just an encapsulation of the inner sense variables for the creature.

```
package memory;

import java.awt.Polygon;
import ws3dproxy.model.WorldPoint;

public class CreatureInnerSense {
    public WorldPoint position;
    public double pitch;
    public double fuel;
    public Polygon FOV;

    public String toString() {
        if (position != null)
            return("Position: "+(int)position.getX()+","+(int)position.getY()+" Pitch: "+pitch+" Fuel: "+fuel
```

The Anatomy Of A Codelet

Every codelet in a CST architecture must declare the methods:

- ***proc()***
- ***accessMemoryObjects()***
- ***calculateActivation()***

The following code is a skeleton to be used for creating any kind of codelet

```
import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.List;

public class MyCodelet extends Codelet {

        private MemoryObject mo1;
        private MemoryObject mo2;
        private List bmos;


        /**
         * Default constructor
         */
```

Let's take a look on each of these methods. The first one which is called every time the codelet is timed is the ***accessMemoryObjects()*** method. In this method, all the memory objects to be used in the ***proc()*** method might be declared and captured using the ***getInput, getOutput*** or ***getBroadcast*** methods. There are different versions of these methods available in the Codelet class. In the example, the input and the output are capture by means of their labels which were declared in their initialization at the ***AgentMind*** class.

The Behavior Codelets

In our application we have 3 behavioral codelets:

- ***GoToClosestApple***
- ***EatClosestApple***
- ***Forage***

Let's take a look in the code for each of these classes:

The GoToClosestApple Codelet

The ***GoToClosestApple*** codelet basically reads information from the ***closestAppleMO***, holding the position of the closest apple, the ***selfInfoMO***, which bring the information from the inner sense of the creature, and write a command for the legs in the ***legsMO***. This command is written with the help of JSON.

```
package codelets.behaviors;

import java.awt.Point;
import java.awt.geom.Point2D;

import org.json.JSONException;
import org.json.JSONObject;
import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import memory.CreatureInnerSense;
import ws3dproxy.model.Thing;

public class GoToClosestApple extends Codelet {
```

The EatClosestApple Codelet

The ***EatClosestApple*** codelet basically uses the information from the ***innerSenseMO*** to get the creature's position, and if the creature is just besides the apple, send the command to hands to eat it. After that, it destroys its internal reference to the apple from known apples.

```
package codelets.behaviors;

import java.awt.Point;
```

```
import java.awt.geom.Point2D;

import org.json.JSONException;
import org.json.JSONObject;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import memory.CreatureInnerSense;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
```

The Forage Codelet

The ***Forage*** codelet basically checks if there is no known apples. If there is no known apples, it sends the command to forage to the legs actuator.

```
package codelets.behaviors;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.List;
import org.json.JSONException;
import org.json.JSONObject;
import ws3dproxy.model.Thing;

public class Forage extends Codelet {

        private MemoryObject knownMO;
        private List<Thing> known;
        private MemoryObject legsMO;
```

The Motor Codelets

In our demo app, we have 2 motor codelets:

- ***HandsActionCodelet***
- ***LegsActionCodelet***

Let's take a look on the code for these 2 classes

The HandsActionCodelet Code

The ***HadsActionCodelet*** is an actuator responsible for eating, picking or burying things at the environment. If it receives any of these commands, it executes it, sending the corresponding command to the server.

```
package codelets.motor;

import org.json.JSONException;
import org.json.JSONObject;
import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.Random;
import ws3dproxy.model.Creature;

public class HandsActionCodelet extends Codelet{

        private MemoryObject handsMO;
        private String previousHandsAction="";
        private Creature c;
```

The LegsActionCodelet Code

The ***LegsActionCodelet*** is an actuator responsible for the creature mobility. It can goes to a specific point given in a command, or decides to rotate, in order to look for apples.

```
package codelets.motor;

import org.json.JSONObject;
```

```
import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.Random;
import org.json.JSONException;
import ws3dproxy.model.Creature;

public class LegsActionCodelet extends Codelet{

        private MemoryObject legsActionMO;
        private double previousTargetx=0;
```

The Perception Codelets

In our app, we have 2 Perception Codelets:

- ***AppleDetector***
- ***ClosestAppleDetector***

Let's take a look on the code of these classes

The AppleDetector Code

The ***AppleDetector*** codelet basically scans the visual range sensor, and in finding an apple includes it in the list of known apples.

```
package codelets.perception;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import ws3dproxy.model.Thing;

public class AppleDetector extends Codelet {

        private MemoryObject visionMO;
        private MemoryObject knownApplesMO;
```

The ClosestAppleDetector Code

The ***ClosestAppleDetector*** codelet basically scans the list of known apples and actualize the ***closestAppleMO***, based on the current position for the creature, detecting the closest apple.

```
package codelets.perception;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.Collections;
import memory.CreatureInnerSense;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import ws3dproxy.model.Thing;

public class ClosestAppleDetector extends Codelet {

        private MemoryObject knownMO;
        private MemoryObject closestAppleMO;
```

The Sensor Codelets

In our app, we have 2 Sensor Codelets:

- ***InnerSense***
- ***Vision***

Let's take a look on the code of these classes:

The InnerSense Code

The ***InnerSense*** codelet, is a sensor codelet which basically reads the creature's inner sense and update the ***innerSenseMO*** with the newest information.

```
package codelets.sensors;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import memory.CreatureInnerSense;
import ws3dproxy.model.Creature;

public class InnerSense extends Codelet {

        private MemoryObject innerSenseMO;
        private Creature c;
        private CreatureInnerSense cis;

        public InnerSense(Creature nc) {
```

The Vision Code

The ***Vision*** codelet is a sensor which basically uses the ***WS3DProxy*** routines to get a list of apples under the current view of the creature.

```
package codelets.sensors;

import br.unicamp.cst.core.entities.Codelet;
import br.unicamp.cst.core.entities.MemoryObject;
import java.util.List;
import ws3dproxy.model.Creature;
import ws3dproxy.model.Thing;

/**
 * Vision codelet is responsible for getting vision information
 * from the environment. It returns all objects in the visual field
 *  an its properties.
 */
```

Conclusion

By concluding this trail, you have studied the Core basics of CST. Most cognitive architectures built with the aid of CST will require just these features shown in this trail. In the next trails, more sophisticated mechanisms will be described and exercized.

References

(Baars 1988) Bernard J. Baars. A Cognitive Theory of Consciousness. Cambridge University Press, 1988.
(Baars & Franklin 2007) Bernard J. Baars and Stan Franklin. An Architectural Model of Conscious and Unconscious Brain Functions: Global Workspace Theory and IDA. Neural Networks, 20:955–961, 2007.
(Baars & Franklin 2009) B.J. Baars and S. Franklin. Consciousness is Computational: The LIDA model of Global Workspace Theory. International Journal of Machine Consciousness, 01(0101):23, 2009.
(Hofstadter & Mitchell 1994) D. Hofstadter and M. Mitchell. The Copycat Project: A Model of Mental Fluidity and Analogy-making, pages 205–268. BasicBooks, 10, East 53rd Street, New York, NY 10022-5299, 1994.
