Tutorial 3
Object Tutorial

In this tutorial, we create an instance of an object based on an object we have in the database. Well also give the object a position and check to see what actions are associated with it. For this tutorial, well be using the Sink object.

In your main function, after initializing the Actionary, add the following line:

```
MetaObject* object = new MetaObject("Sink_0");
```

This creates a new MetaObject, which is the object system used throughout PAR. MetaObject can take two parameters, one of which is the object name. Notice how we use the name of the object we want to create followed by an underscore. When creating a MetaObject, the system determines the parent object by the root of the name (in this case, Sink). The system understands that this item is an instance through the use of the underscore. So, naming the sink instance Sink_blue or Sink_kitchen are all valid sink instances, and the program will fill in the information related to the object being a sink. The other optional parameter is a Boolean parameter to denote if the object is an agent. In our example, the default value of false is used. Agents are considered objects and can contain all of the properties other objects do. When we create an AgentProc instance, as we did in Tutorial 1, a corresponding MetaObject for the agent is automatically created.

Now that we have an object, lets give it a position. PAR uses a derivative of the standard C++ vector class, called Vector. We can create and populate a Vector object as:

```
        Vector<3> *pos = new Vector<3>();
        pos->v[0]=8.0f;
        pos->v[1]=1.0f;
        pos->v[2]=2.0f;
```

This creates a vector class instance and sets the position to (8,1,2); To assign the object to this position, we simply use the following command:

```
object->setPosition(pos);
```

Which changes the position to our object. Clearly the position of an object can be linked to an animation framework.

Finally, we query the Actionary about the actions that might be associated with a sink object:

```
        // Check to see what actions are associated with a sink
        cout << "The sink can be cleaned (true or false)? " <<
                object-> searchAffordance(actionary->searchByNameAct("Clean"),0) << endl;
        cout << "The sink can be eaten (true or false)? " <<
                object-> searchAffordance(actionary->searchByNameAct("Eat"),0) << endl;
```

The output will show that a sink can be a participant in a Clean action, but not an Eat action. The sink can be cleaned, but not eaten. The Actionary does not actually contain a capability record for Sink_0 nor, in fact, for Sink. These capabilities are inherited from the overarching Physical Object object. The position of the affordance also plays an important role in determining if an object can be a part of an action. Object positions within an action are counted from the zero position. Examining the included clean action (CleanPre.py), we see that two objects are used in the action. The first is the item to be cleaned (position 0) and the other is the instrument to clean with (position 1). We did not have the Sink_1 (or Sink) be an item that can be used to clean initially, but we can set it to using the command

```
actionary->setAffordance(actionary->searchByNameAct("Clean"),object,1);
```

This function allows all Sink instances to clean, even though we passed an instance. Setting an affordance adds the affordance to the database, and the database focuses on relationships of MetaObjects.

Finally, we can explore how objects relate to other objects. Specifically, MetaObject separates possessing and containing other objects. Possession is considered to be a mental ownership, such as a persons lunch. The person may have their lunch contained within another object (such as a refrigerator). With this example, the refrigerator contains the persons lunch, but does not own it. The functions for possessions and contents mirror each other. To set an object within another object, use the function

```
        // Creates a new object, and adds the sink to it's contents
        MetaObject* container= new MetaObject("Bookshelf_large");
        container->addContents(object);
```

Likewise, to add an object to the bookshelfs possessions, simply use the function addPossession. To search contents, use

```
        // Checks to see if the object is in the contents of the container (physcially)
        //        and the possession (mentally)
        cout<<"The sink is physially in the bookshelf(true or false)? "<<
                container-> searchContents("Sink_0")<<endl;
        cout<<"The sink is owned by the bookshelf(true or false)? "<<
                container-> searchPossession("Sink_0")<<endl;
```