# MUSE Installation and API Details

In this section we will present the seven public classes that are available to the user. MUSE has the following classes available for the user:

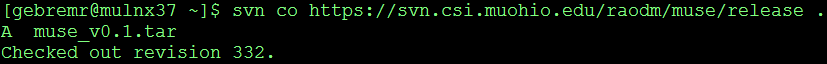
* muse::DataTypes
* muse::Simulation
* muse::Agent
* muse::State
* muse::Event
* muse::oSimStream
* muse::SimStream

The header files for these classes are available *MUSE\_ROOT\_DIR/include* directory. The best way to learn MUSE API is by creating a simple simulation. This example will serve two reasons. The first is to show how easy it is to get setup and going with MUSE. The second reason is we get to demonstrate how to use each available class within the example. After we describe how to configure and install MUSE, we will describe the ping-pong simulation example. While we build the example from the ground up, we will describe the different classes as we use them.

## Configuring and Installing MUSE

There are a couple of steps before you can actually use MUSE. However, we have made getting started with MUSE as painless as possible. In this section we describe how to download, configure and install MUSE. First, we must grab MUSE. As of this writing MUSE is not publicly available (password protected), however it will soon be available through SVN. The current stable release is MUSE beta version 0.1. To get the latest release make sure you have SVN client and execute the following command in your shell.

*svn co https://svn.csi.muohio.edu/raodm/muse/release* ***.***

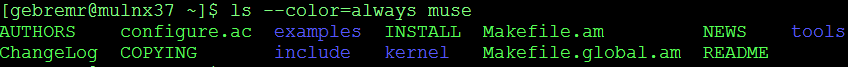


The following command above will download the latest tarball of MUSE beta version 0.1, which is stored in the release directory. Next we want to expand the tarball and rename the directory to *muse*.

**

*mv muse\_v0.1 muse*

The two commands above will give you a directory called *muse*, which will house MUSE source code and examples.



MUSE has a couple of dependencies. The following are what we tested and developed with for MUSE beta 0.1.

* GCC version 3.3.4
* MPI version 1.2.5
* GNU AutoConf version 2.59

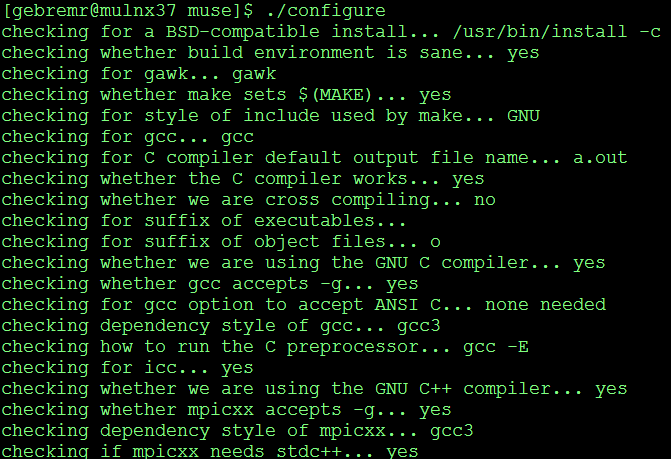
If you are installing MUSE of Miami’s Redhawk cluster then all of the dependencies are already installed. These tools are freely available and installation of these tools is out of the scope of this thesis.

Once we have the directory and all its content we are ready to configure MUSE. To configure MUSE first, change into the *muse* directory and run *autoreconf*, which will generate a configure script.

autoreconf -i -v

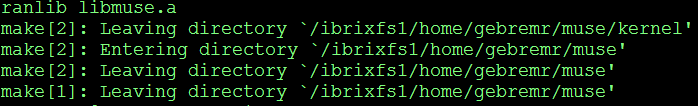
**./**configure

Executing the configure script will check for needed dependencies and created the make files for MUSE and all the examples. The figure below shows the script in action.



When the script is complete and your system has all the needed dependencies all that is left to do is run the *make* command and MUSE source will compile. Run the following command.

*make*



The figure above show what you should see if MUSE compiled correctly. At this point you have installed, configured, and compiled MUSE and the provided examples.

## Background on Ping-Pong Simulation

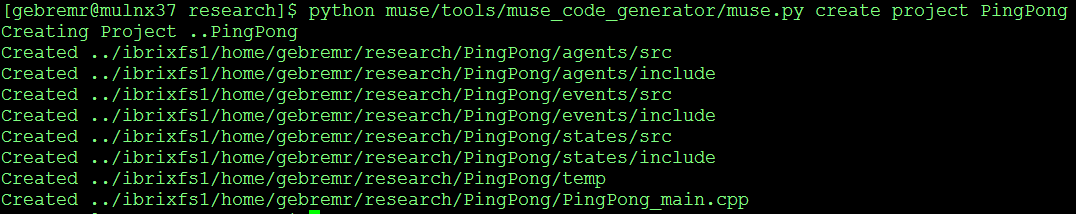
In this section we describe the case problem in more detail. Ping-Pong simulation is meant to be an easy simulation that we can implement and at the same time use as a learning tool to understand MUSE API. We will be simulating a rally between to ping-pong players. To make things simple, we will have no random variables. When a player receives a ball, the player will return the ball to the opposite player. The simulation will go one for a given amount of time. At the end of the simulation we will want to know how many times each player hit the ping-pong ball. Hence, for this simple simulation we will need a ping-pong player, a ping-pong ball and we will need to maintain the number of balls each player returns. Now we can implement and learn about the API in the next section.

## Implementing Ping-Pong Simulation

Implementing a simulation with MUSE is very easy and enjoyable. One of our requirements was ease of use. We want to make simulation development with MUSE very intuitive. To get started we must first setup our project, here we can use the MUSE code generator. The MUSE code generator was explained earlier, so create the project and call it *PingPong* with the following command.

*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create project PingPong*

The above command, replace *MUSE\_ROOT* with the path to directory that holds MUSE source code and examples. The following figure shows what happens after the *create project* command is executed.



When we described MUSE code generator, we also discussed the ability to create *makefile* on the fly. We change into the PingPong directory, which is the project directory and the following command was used to create the *makefile* for our example.

*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create makefile MUSE\_ROOT*

With two commands, MUSE code generator has created and setup a MUSE project for us. Now we can compile the code by running *make* command. Once we compiled the project, we can run the main stub class. This will verify everything went well and the following figure is the output you should see.



Most of the hassle with most simulation framework is getting start and setup. MUSE handles all of the setup for you, so you can just get started on development. From here on, we will develop from bottom up. We will start by creating the class that holds the information for each ping-pong player. In MUSE, this is represented with the *State* class. We create a class called *PingPongState*. This class inherits from the base class *State* and will house our information about the player.

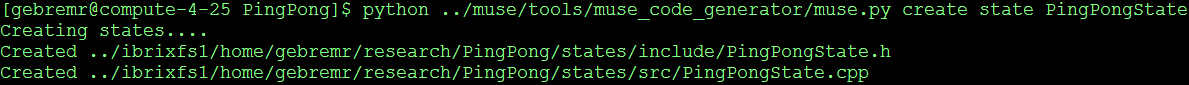


Figure 1: The State class

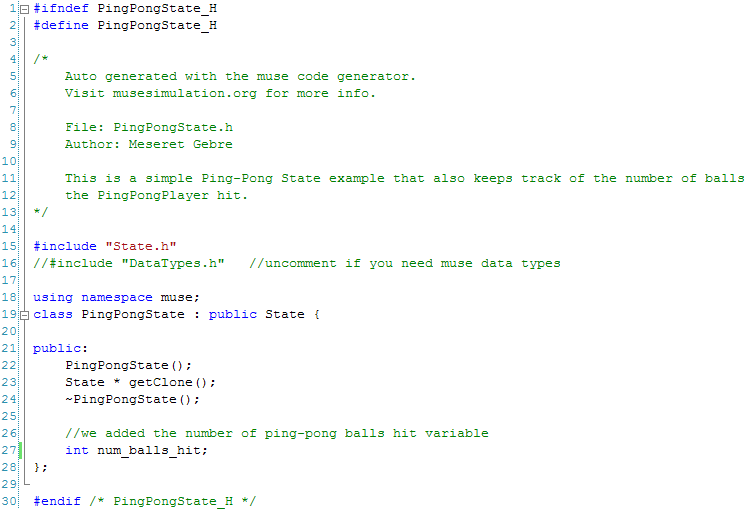
The state can be seen as everything that we need to know about an agent at any given time. The state by definition should not be anything that is static and can change at any time. The amount of information in the state can shrink or grow. Therefore, you should put any data that you need to modify in the state. There are only two public methods in the *State* class. The information stored in the state can change, so we need a way to record at what time the information was changed. The MUSE kernel automatically handles this, but you can get the time stamp of the state by invoking the *getTimeStamp()* method. The most important method, which is heavily used by the kernel is the *getClone()* method. This method is declared virtual and must be implemented by the subclass. Not implementing this method will give unknown behaviors, which will cause MUSE to abort. Typically for classes that have primitive types only, a shallow copy is sufficient, however class with pointers or objects as variables should implement deep copy to return a proper clone. Once you subclass from the *State* class, feel free to add any data type you need. A good rule of thumb is to try and minimize the information you need for the time it is needed. You can really improve your simulation time by wisely using different versions of the same state. If you have static data, refactor it to the agent class, if the data never changes there is no sense in having multiple copies. The *getClone()* method also must return a pointer to a heap allocated object. If the kernel calls for a clone it will handle disposing the memory, however, if the user calls for a clone the user must remember to release the memory. State cloning is very important; the kernel depends on these clones for storage purposes. If there is ever a rollback, MUSE can revert to a safe state from the past.

Reverting back to our example we can create a stub State class with the MUSE code generator. The following command will generate *PingPongState* class for us.

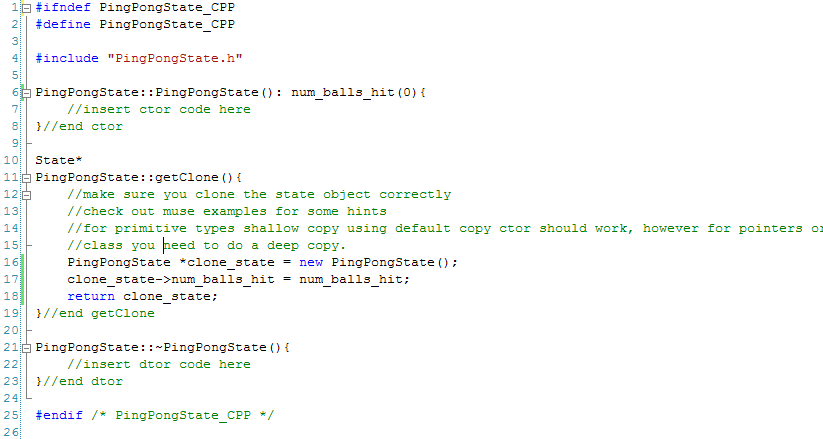
*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create state PingPongState*



In the header file for our PingPongState class, we must add a variable to keep track of the number of balls each ping-pong player hits. Figure below is what the header file looks like.



The only thing we added was the variable (shown by the green arrow above) the rest was generated for us. We move on to the source file and implement the *PingPongState::getClone()* method and initiate the *num\_balls\_hit* variable to zero. The resulting figure is what the code looks like.



With only five lines of code that we added, we have completed the *PingPongState* class. Now that we have a way to store the player stats, we need to create a ball for the player to hit back and forth. Essentially the ball will be an event in a MUSE simulation. In the simulation players will send each other events and when players receive an event, it can be thought of as a ping-pong ball.

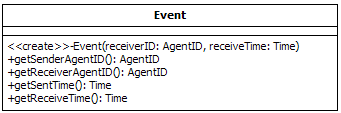


Figure 2: The Event class

The *Event* class has four public methods for the user. The only to create an *Event* object is to heap allocate it. Once an Event object is create you can probe the object for certain types of information. To get the *AgentID* of the agent that sent the event, use the *getSenderAgentID* method. Likewise, we can figure out who the receiving agent of the event is by invoking the *getReceiverAgentID* method. You can also get the sent time or receive time of the event by invoking the *getSentTime* or *getReceiveTime* method. However, one thing that still left unexplained is the custom MUSE defined primitive types. For example, the *getSenderAgentID* and *getReceiverAgentID* methods return an *AgentID* type. The *getSentTime* and *getReceiveTime* methods return a *Time* type. All of MUSE types can be used if we include the *DataTypes* header file.

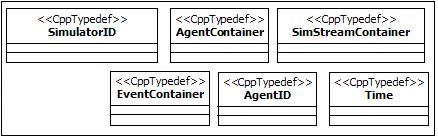


Figure 12: DataTypes header

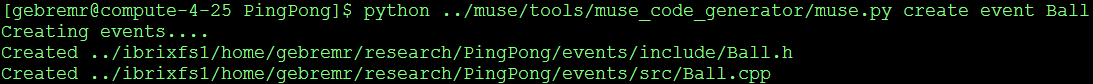
Figure 12 above shows the available MUSE defined data types. *SimulatorID* is used to identify kernels in the simulation. When you initialize the kernel, it automatically assigns itself a *SimulatorID*. *AgentContainer* is used to store agent pointers. The *Simulation and Scheduler* class uses this to contain the registered agents. We discussed how *Agent* classes can write to any *SimStream* based class. The *Agent* class uses the *SimStreamContainer* to store registered *SimStream* based classes. When it is time for an agent to execute its events for any given time, it is passes an *EventContainer*. These are used to store events for processing. It is up to the agent to iterate through the container and process each event accordingly. All the containers are just typedef STL containers and can be used just like the STL containers. As of this writing all the container discussed are of type *std::vector* which hold pointers to the class they contain. *AgentID* are just like *SimulatorID*, but they are used to identify agents. All IDs should be globally unique! We leave this to the user to define. *Time* is the last data type, this is used to describe the time in the simulation. Benefits of MUSE defined data types are very clear when you view the code. Parameters are very clear and understandable, for example:

1. void foo(Time t1, AgentID id1, SimulatorID id2);
2. void foo(double t1, int id1, int id2);

I purposely chose uninformative variable names and most of the times this is how naive developers code. However, with the first example you can clear understand what each variable represent, because the data types are themselves informative. The second example leaves a lot to the code reader to try and guess. This is a very simple example there are methods that take many parameters and that’s when you truly see the benefits.

In the ping-pong simulation we represent the ball by creating an *Event* class called *Ball*. To create the *Event* based class *Ball*, we used MUSE code generator and execute the following command in our shell prompt.

python ../muse/tools/muse\_code\_generator/muse.py create event Ball



For our example, the *Ball* event does not need to carry additional information. When the player agent receives the *Ball* event, the player must create a new *Ball* event and send it back to the opposite player. Also, all events that are in the simulation are automatically cleaned up by the kernel at the end of the simulation. The following two figures show what the source and header file should look like for the *Ball* class.

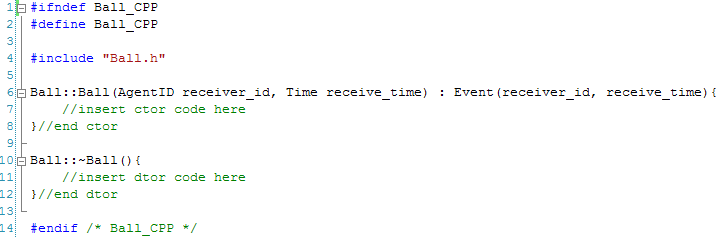


Figure 3: Ball class generated source file

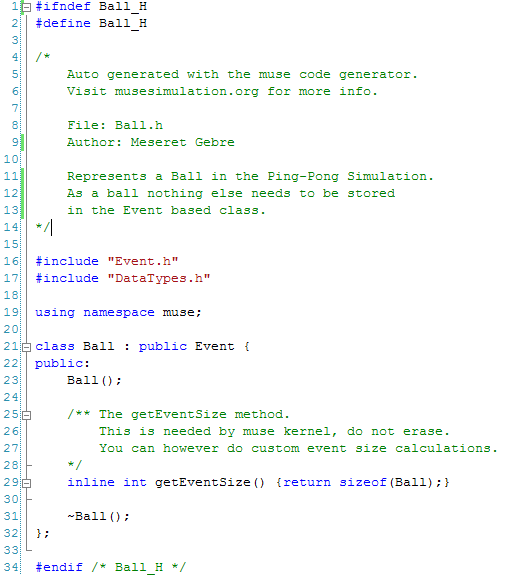


Figure 4: Ball class generated header file

We now have completed our *Ball* class and up to this point (besides commenting code) we have only added five lines of code. Next we create the ping-pong player. In term of MUSE, we can represent a ping-pong player with the *Agent* class.

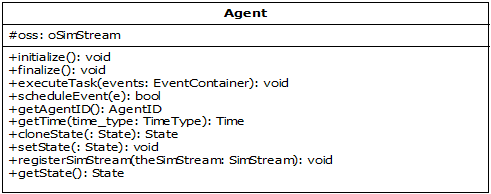


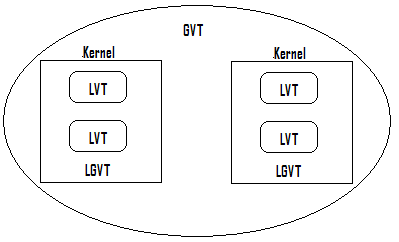
Figure 15: The Agent class public methods

The *Agent* class is a base class provided to represent agents in the simulation. Agents are autonomous and independent; this *Agent* class handles most of the heavy lifting for the user. There are a couple of important things to understand about the *Agent* class. The first three methods, the destructor, and the *cloneState* from figure 15 above are declared virtual methods and should be implemented by the subclass. The *initialize()* method should contain information and procedures to initialize the agent. When the simulation is started, the kernel will invoke all *initialize()* methods of all the agents that are registered. Likewise, the *finalize()* methods should store information and procedures to finalize and end the agent class. The kernel will call the *finalize()* method when it is finalizing. Figure 16 below visually shows this process.



Figure 16: Sequence of initializing and finalizing an agent

The most important method is the *executeTask(events).* This is the only way you communicate with the agent. In parallel simulation, we do not have the luxury of having pointers to the agent we want to communicate with. As the developer, the subclass should handle the event(s) it gets accordingly. The *Scheduler* class will inform the agent when it is time to process its next set of events and these are the event(s) the agent gets. When an agent creates an event, it must use the *scheduleEvent(event)*  to schedule that event. This method handles all the work of determining the receiver agent’s location and how to get it there. To get the identifier of the agent, use the *getAgentID()* method. Agent class also provides the user with time information. You can grab three different times, based on what parameter you pass into the *getTime(TimeType)* method. *TimeType* is an enumeration which contains *LVT, LGVT, and GVT*. Default parameter is the *LVT* (local virtual time). However, the agent can get the *LGVT* (local global virtual time), this is the least LVT of all agents registered to the kernel. *GVT* (global virtual time) is the least LGVT throughout all the kernels. Most operation just need to call *getTime()*, because the *LVT* is sufficient. The figure below visual explains the different time types. An option to get a clone of the agent’s state is available through the *cloneState(state).* To get a pointer to your current state, just call the *getState()* method. Another method that is available is the *setState(state)* method.



Agent’s LVT

Equals smallest LGVT

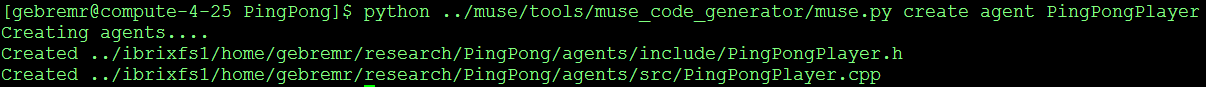
Equals smallest LVT

Figure 5 : getTime method time types

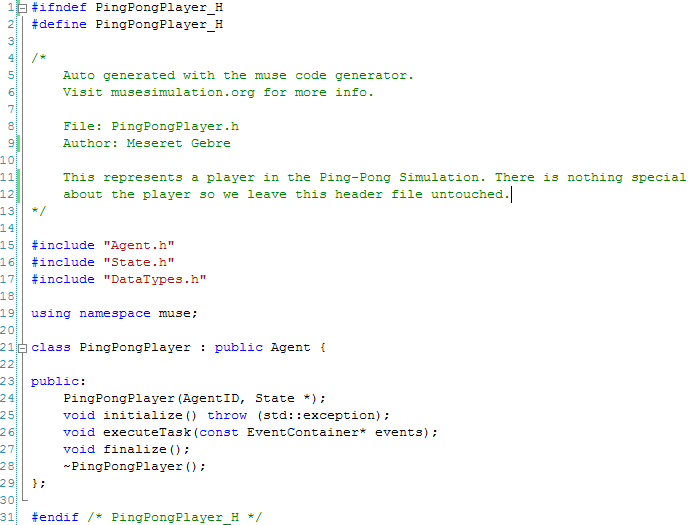
We already talk about the *State* class, but briefly the state of an agent is just a collection of data that can be modified through the life cycle of the simulation. Accordingly, there are cases when we do not need all the information at once. For example, if we had a person agent, we can run the simulation and the person as a baby, and therefore we would not need to store information about the person’s school grades or what type of car the person drives, yet. When it comes time to fast forward this persons age to say twenty-one then the information mentioned above become significant. Therefore, we can have many different types of states and we should be able to switch based on the need of the information. The advantage becomes evident with the space we are saving, which increases performance. The last method publicly available is *registerSimStream(SimStream).* Running simulations is about gathering data. MUSE allows the modeler to extract the data to any stream that has a stream buffer. We will discuss how to properly use the SimStream later in this section. That sums up the *Agent* class public API.

Now that we have explained the Agent class and MUSE data types, we can implement a ping-pong player as an agent. We call the agent *PingPongPlayer*. Using MUSE code generator, the following command will generate the header and source file for the *PingPongPlayer* class.

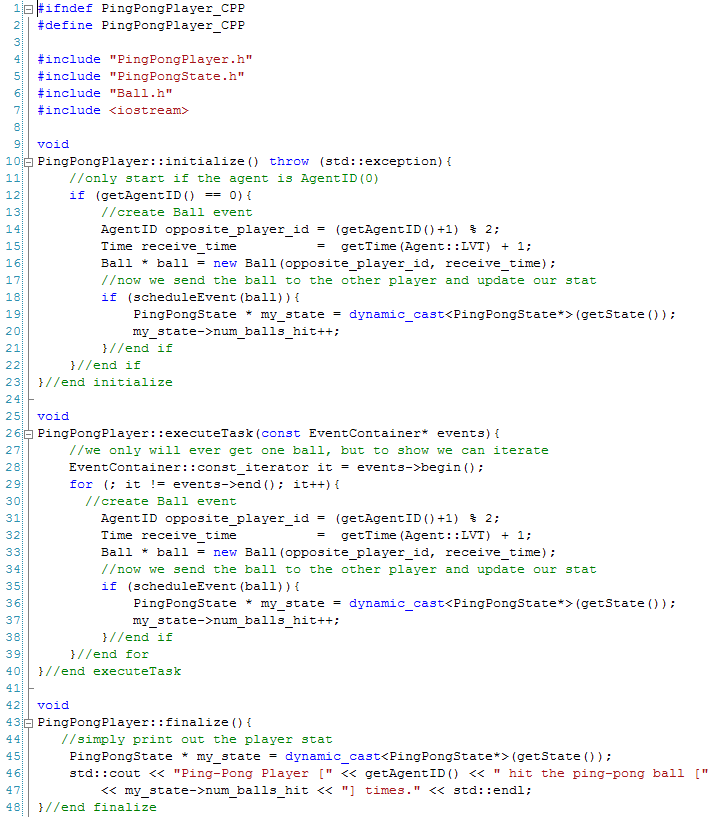
*python ../muse/tools/muse\_code\_generator/muse.py create agent PingPongPlayer*



The PingPongPlayer class is very simple; hence the default header file generated by MUSE code generator can stay unmodified. The following is what the implementation should look like.



The source file that was generated will have to be modified. In the simulation, there will be two players. The first player will have an *AgentID* assigned to zero, the second player *AgentID* we assign to one. The *initialize* method in *PingPongPlayer.cpp* will get the simulation started by letting the player with *AgentID* zero start by hitting the ball first. The next method to implement is the *executeTask* method. There are three actions that we must take care of in the executeTask method. First, we must grab a hold of the agent’s state so we can modify the player’s stats. Second, we must create a new *Ball* event. Lastly, we need to send the ball to the other agent. The *finalize* method will simply print out the player’s stat. This will be the number of time the player hit the ping-pong ball. Since this is a simple example we use the default implementation of the cloneState method in the *Agent* base class. The following figure is what the implementation should look like.



getState returns a pointer to State, we must cast this to a PingPongState pointer.

Don’t forget to include

The PingPongPlayer class is now complete and with the help of MUSE code generator we only added sixteen lines of code in the source code. The last piece we need in place to have a MUSE simulation is the main execution code. When we created the *PingPong* project earlier, MUSE code generator also created a main execution file called PingPong\_main.cpp for us. To understand what is contained in that file we must learn about the public methods in the *Simulation* class.

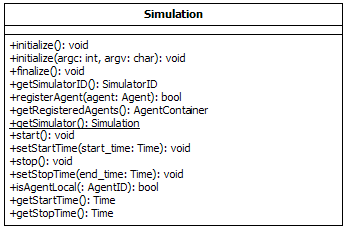


Figure 13: Simulation Class

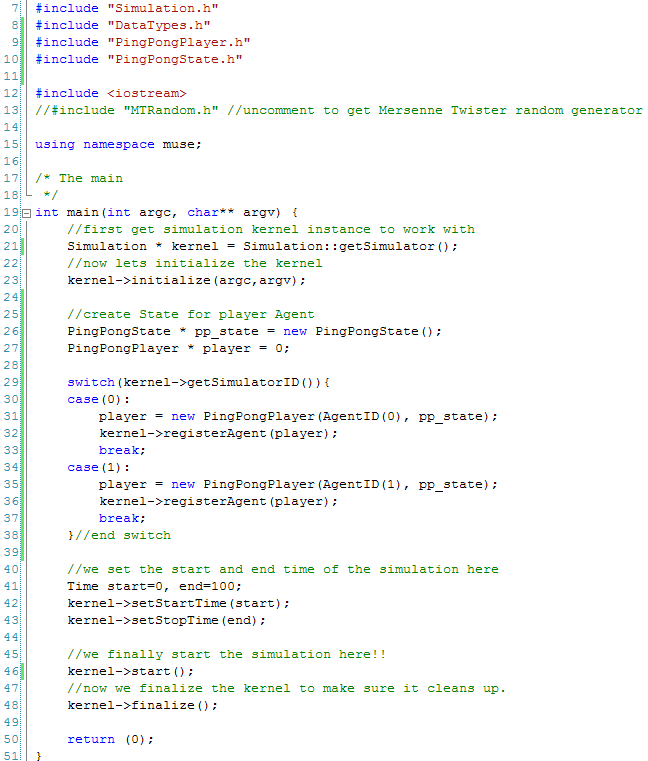
Figure 13 above shows all the public available method in the *Simulation* class. When you run a simulation with MUSE there is a common order of methods that must be called. First, you request an instance of the *Simulation* class. *Simulation* class implements the singleton pattern, so to get an instance you use the *Simulation::getSimulator()* method, this will return a pointer to the class. Once an instance is acquired you have to initialize the instance. This can be done with two methods. The first option you have is the *initialize()* method. The second is the *initialize(argc,argv)* this lets you pass in arguments from the main executable. The arguments are not used in anyway by the kernel, but they are passed in to init MPI. When the simulation kernel is initialized it will attain a valid *SimulatorID*. It is important to note that initialization should only happen once. After initialization is complete, you should set the start and stop time of the simulation. This can be done with the *setStartTime(Time start)* and *setStopTime(Time stop)* methods. After this point you should create and register your agents with the simulation kernel. The *registerAgent(Agent \* agent)* method is used to let the kernel know of agents that it is responsible for. The simplest step, which gets the entire simulation started is the *start()* method. Lastly, you need to make sure that all agents and internal resources are freed. Calling the *finalize()* methods releases all of the internal resources and external resources like the agents and events created. The remaining public methods are just getters, which are self explanatory. The following (figure 14) is a sequence diagram to visually show what was just described.



Figure 14: Sequence Diagram of starting a simulation

Keep in mind that the Simulation class calls other classes that were not shown, but we will see more sequence diagrams as needed.

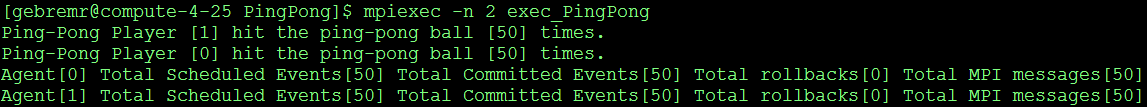
In our example, MUSE code generator already created and called most of the methods for us. All that is left is to register the two ping-pong players. We want to the players to reside on two different nodes. MUSE uses MPI, so when we call *Simulation::getSimulatorID* method we are actually getting a unique id to differentiate the nodes. Hence, we can use the SimulatorID to figure out which kernel to register each ping-pong player. The following is what PingPong\_main.cpp should look like.



Added Code

Don’t forget to include

We have now completed the main source code. We only added twelve lines of code to the main file. In total we have only implemented thirty-three lines of code. Hence, out of the 241 lines of code, we only truly wrote about 10% and the rest was done with MUSE code generator. This is yet another example how simple we made modeling with MUSE. To run the final simulation, we recreate the makefile, compile with the make command, and run the main execution file like we did earlier and the following figure is what you should expect to see.



Since we left the default stop time for the simulation we ran the simulation for 100 time steps. From the simulation we can see that each player hit the ball 50 times each. MUSE also prints out some statistics about each *Agent* class. For each agent, we know that there were a total of 50 events scheduled and 50 events were committed. There were no rollbacks, which makes sense and the total number of MPI messages used is 50. This is because the agents resided on two different nodes.