# MUSE API Details

In this section we will present the seven public classes that are available to the user.

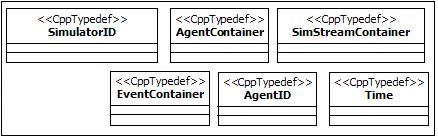


Figure 12: DataTypes header

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 clearly understandable, for example:

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

I purposely chose uninformative variable names and most of the times this is how 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.



Figure 13: Simulation Class

Figure 13 above shows all the available method from 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*. 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. At this point is when 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 as discussed earlier. The simplest step, which gets the entire simulation started is done with the *start()* method. Lastly, you need to make sure that all agents and internal resources are freed. Calling the *finalize()* methods handles taking all of the internal resources and most of 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.



Figure 15 : The Agent Class

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 and the destructor 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 and 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 time according to the kernel where this agent resides. *GVT* (global virtual time) is the least time throughout all the kernels. Most operation just need to call *getTime()*, because the *LVT* is sufficient. 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 declared virtual is the *setState(state)* method. There is one good reason to make this method virtual.

We will talk about the State class next, but in 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.



Figure 17: 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; an example of this was given earlier. Therefore, you should 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.



Figure 18: The Event class



Figure 19: The oSimStream class



Figure 20: The SimStream class