GCM 4.0 Modeling Guide

# Chapter 1: Introduction

## Who is this for?

The General Computation Model (GCM) is a Java based simulation framework for building disease progression models. Users of GCM should have a general familiarity with Java and object oriented programming and would benefit from some exposure to event based modeling.

## High level overview

### Simulation

GCM is an event based simulation composed of data managers, actors and an event engine. The data managers contain the state of the simulation and generate events when that state changes. The actors contain the business logic of your model and act on the data managers. The engine transports events generated by the data managers to any data managers and actors that subscribe to those events.

### Plugins

Data managers and actors are organized into plugins. A GCM model is thus composed of the core simulation and a suite of plugins. The plugin architecture provides for the scalable reuse of concepts and capabilities between models. GCM is provided with a set of existing plugins that define many of the concepts useful to a broad range of models such as the management of people, their properties, social group structures and the like. The modeler is free to compose a model from their choice of plugins.

### Experiment

GCM also provides an experiment management system. Each plugin contains zero to many data objects that define the initial state of its actors and data managers. Each such data object may be altered freely. The complete set of all combinations (scenarios) of the variant plugin data objects form an experiment and a separate simulation instance is executed for each combination.

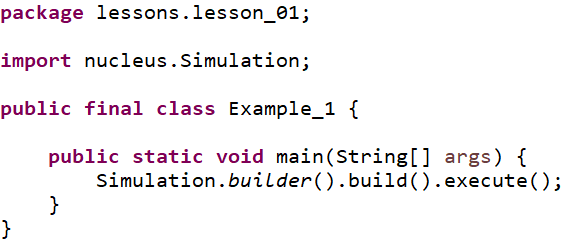
# Chapter 2: Getting Started

We start with a set of practical lessons that will help clarify the core concepts of GCM. The lessons generally build on one another and should be taken in order. You are encouraged to code along with the lessons.

## Lesson 1: Hello World

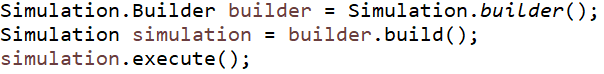
Our first lesson is a very reduced “Hello World” example where we will execute the simulation with one line of code.

***Figure 2.1.1***



With this one line we have created and executed a simulation. Since the simulation had no actors or data managers there was nothing to do and so it terminated immediately. Let’s analyze the line in a more drawn out form:

***Figure 2.1.2***

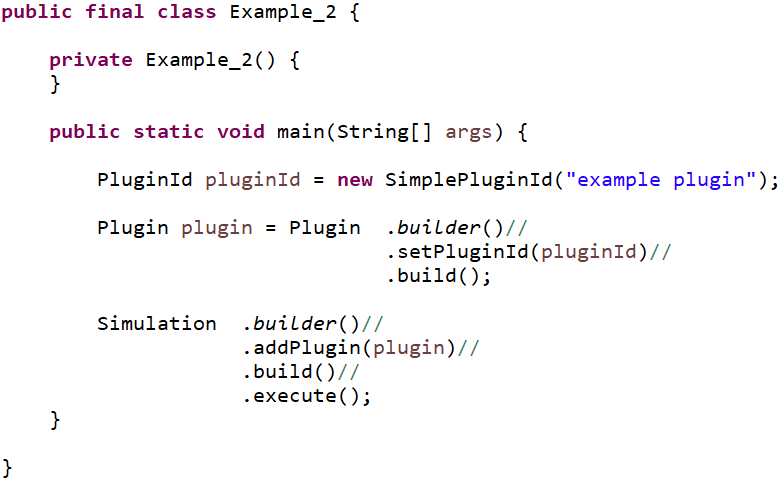


The simulation does not have a constructor. Instead it uses a static builder class that creates the simulation from various arguments. The builder is immediately capable of building a simulation instance so we will skip giving it any more information. The simulation is only capable of executing, so we execute it.

## Lesson 2: Plugins

Models are made of plugins. In this lesson we will add a single plugin to the simulation and execute it.

***Figure 2.2.1***



The first thing we will need to do to build a plugin is to identify it. The PluginId is a marker interface – it has no methods and serves to help differentiate between plugin id values and other identifiers. The SimplePluginId is a convenience implementor of PluginId and will wrap any object as an identifier. In this case we use the string “example plugin”, but you are free to implement them however best fits your needs.

Next we build the plugin. The Plugin class implements all plugins and you can provide several arguments to its builder to specify the contents and behavior of your plugin. A plugin is composed of four items:

1. An id
2. Dependencies on other plugins
3. Data objects used to initialize data managers and actors
4. An initializer to load the data into the simulation

For now, we will only need to add the plugin id and build the plugin.

Finally, we build the simulation by adding the plugin and then executing as usual. The result is the same as the previous lesson: nothing happens. However, internally, the simulation did add the plugin and found it had no information other than its id.

## Lesson 3: Actors

### Contexts

In all that follows, we will encounter various context objects. Contexts are interfaces into the simulation that are tailored to the thing using the context. For example, an ActorContext provides everything that an actor will need to interact with the simulation. Similarly, a DataManager context provides the capabilities needed by data managers.

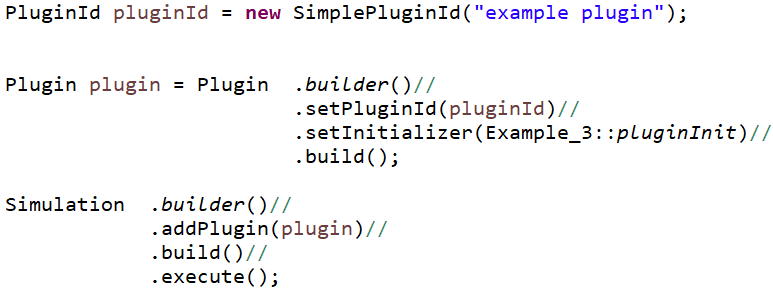
The first context we encounter is the PluginContext. It provides the plugin with the following abilities:

1. Add an actor to the simulation
2. Add a data manager to the simulation
3. Get plugin data

The PluginContext is passed to the plugin’s initializer and is used to add all data managers, all initial data and any actors that need to exist at the beginning of the simulation run.

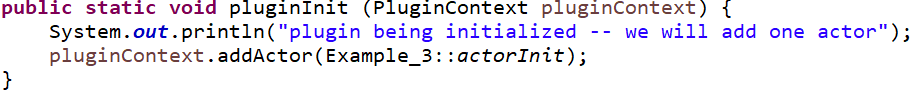
The next context will be the ActorContext. It provides actors with a wide array of capabilities that we demonstrate later. For now, the important takeaway is that being granted a context implicitly identifies the recipient as having a particular role in the simulation.

***Figure 2.3.1***



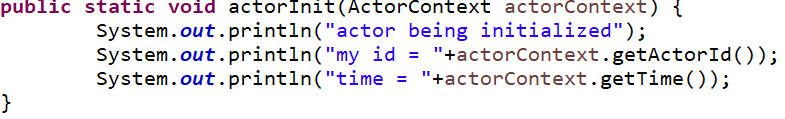
We are setting the plugin’s initializer. The initializer is a method that consumes a PluginContext and returns void. For this example, we use a static local method for our initializer:

***Figure 2.3.2***



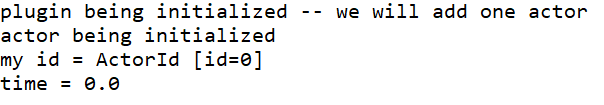
When the simulation starts up its execution, one of the first things it will do is to execute each plugin’s initializer to give the plugin an opportunity to add actors and data managers to the simulation before time and events begin to flow. Adding an actor is done with another consumer, but this time it is a consumer of ActorContext.

***Figure 2.3.3***



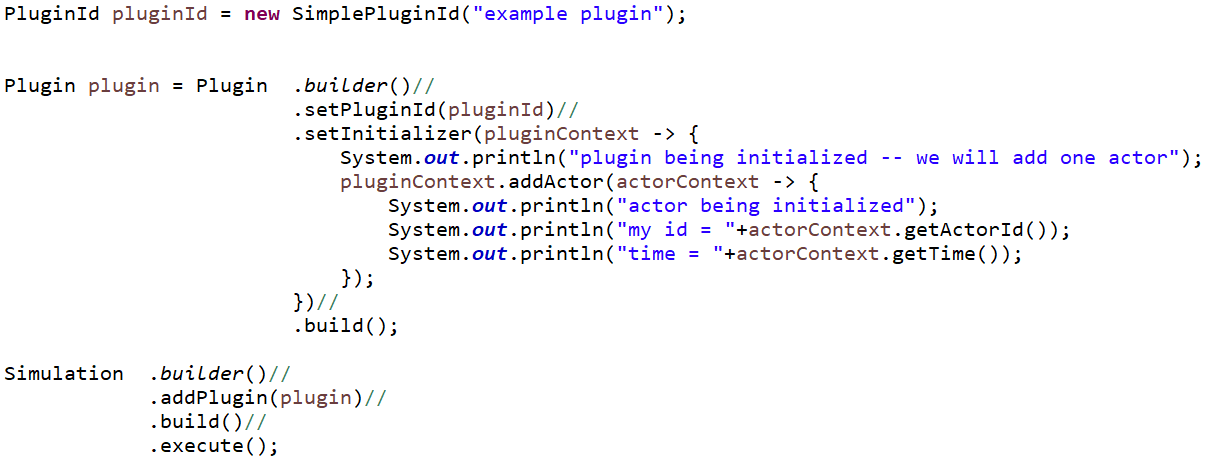
After the plugins are initialized, the actors and data managers are next. For this example, the actor is initialized and it prints a few statements and ceases activity. Here is the resulting console output:

***Figure 2.3.4***



We can replace the local method references above with lamdas to be more succinct.

***Figure 2.3.5***



## Lesson 4: Data managers

We extend the previous lesson by slightly altering the actor and adding a data manager. But first let’s list some of the attributes of data managers and actors to better understand the roles they play in the simulation.

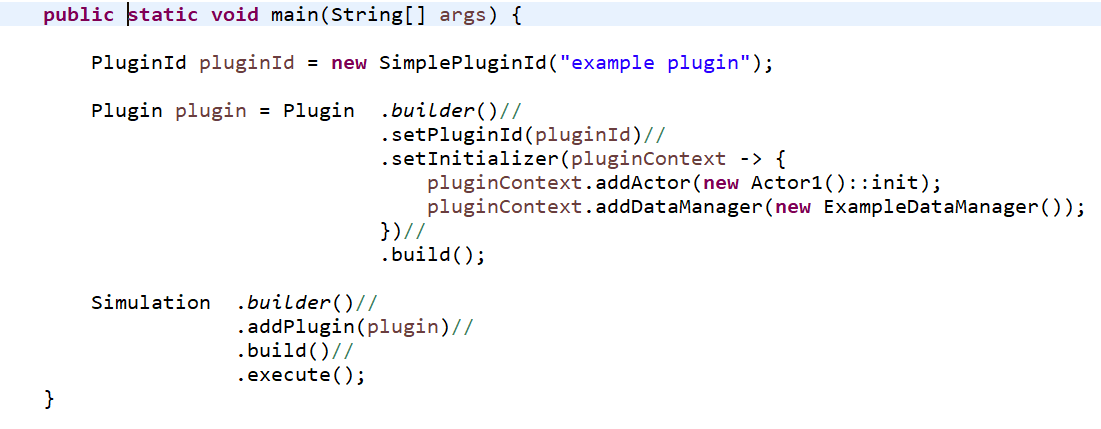
Data Managers

* Exist for the full duration of the simulation
* Contain and maintain the entire state of the world.
* Are highly stateful
* Produce events in reaction to state changes
* Interact with other data managers via events
* Do not have a set of objectives. They are not trying to achieve some particular state of the world
* Are narrowly focused on some particular aspect of the world, but are concerned with all instances of that aspect
* Are added as instances and are limited to a single instance per class type

Actors

* May be added and removed over time
* Are not considered to be part of the world
* Are generally stateless
* React to but do not produce events
* May access any data manager
* Have objectives. They contain the business logic of the model and are trying to achieve some particular state of the world
* Are concerned with many aspects of the world, but often focused on a particular subset of world
* Are added as consumers of ActorContext and may be composed of any such consumers

***Figure 2.4.1***



We add an instance of ExampleDataManager to simulation. Unlike the actor, where we pass a consumer of context, we need to provide an actual instance of a data manager. Note that the ExampleDataManager extends the base class DataManager. The base class provides the only init() method to override and you must include the super.init(dataManagerContext) call as its first line. This is done to ensure that each data manager is initialized exactly once by the simulation.

The ExampleDataManager has two (completely arbitrary) data fields alpha and beta and provides both getters and setters for each.

***Figure 2.4.2***



The actor is now specified via the ExampleActor class. Most actors contain enough code that we usually put that code into a separate class rather than a lambda statement as we did in the previous lesson. Note that the init() method has the correct method signature of being a consumer of ActorContext.

### Plans

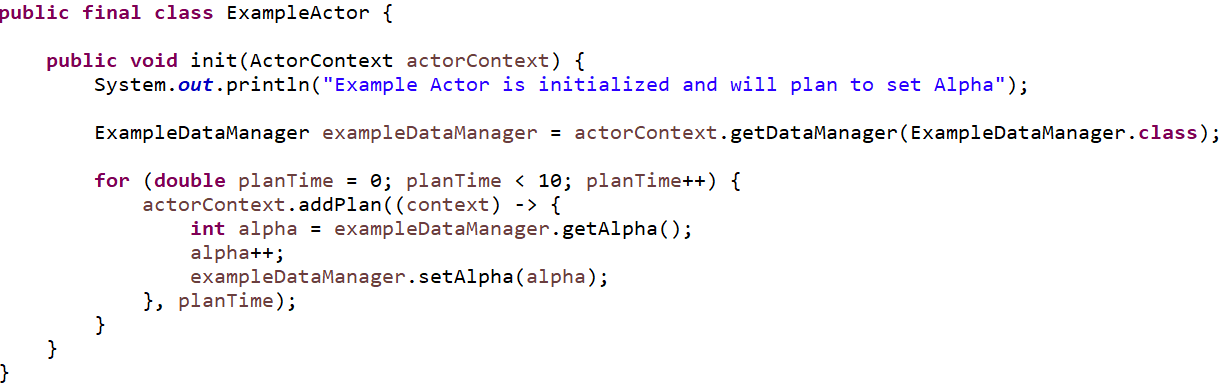
In GCM, an actor can do three things:

1. **Observe**: Observation can be done directly by gaining access to a data manager and then getting a value from that data manager. Observation can be done indirectly by subscribing to events. We will cover that option later.
2. **Act**: A mutation to some data manager’s managed data.
3. **Plan**: At some time in the future, the actor will take some particular action

Actions in GCM are always executed in the current moment in the simulation. Unlike many future event simulations where events are queued for future execution, GCM allows an actor to plan for an action or observation in the future. The plan is a consumer of ActorContext and can be a static method, member method or a lambda. The plan is registered with the simulation and is executed only when time has moved forward to the plan’s scheduled time. There is no requirement that the plan do anything at all. This allows the flexibility to re-evaluate the circumstances of the planned action and choose to take appropriate action at that time. Plans are queued in GCM by their associated planning times and it is this queue that dictates the flow of time. For example, suppose the simulation finds the first plan is scheduled for time= 2.4 days. The current time = 0 days and the simulation progresses time to 2.4 days and then invokes the plan. Plans are always privately managed by the actor that owns the plan and no other actor or data manager has any insight into those plans. See the planning chapter for more details on planning.

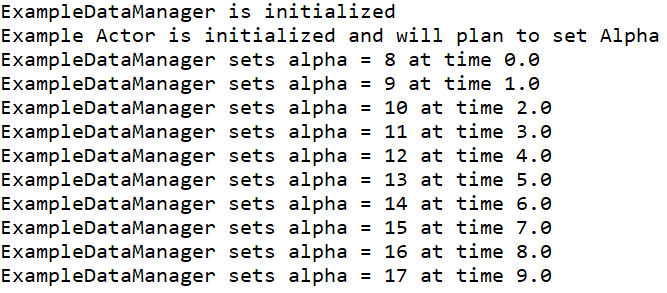
In this example, the actor is initialized at time= 0 and generates 10 plans to increment the value of the alpha in the ExampleManager. Each time the ExampleManager changes the value of alpha, it outputs to the console a description of the change.

***Figure 2.4.3***



The output from the simulation is:

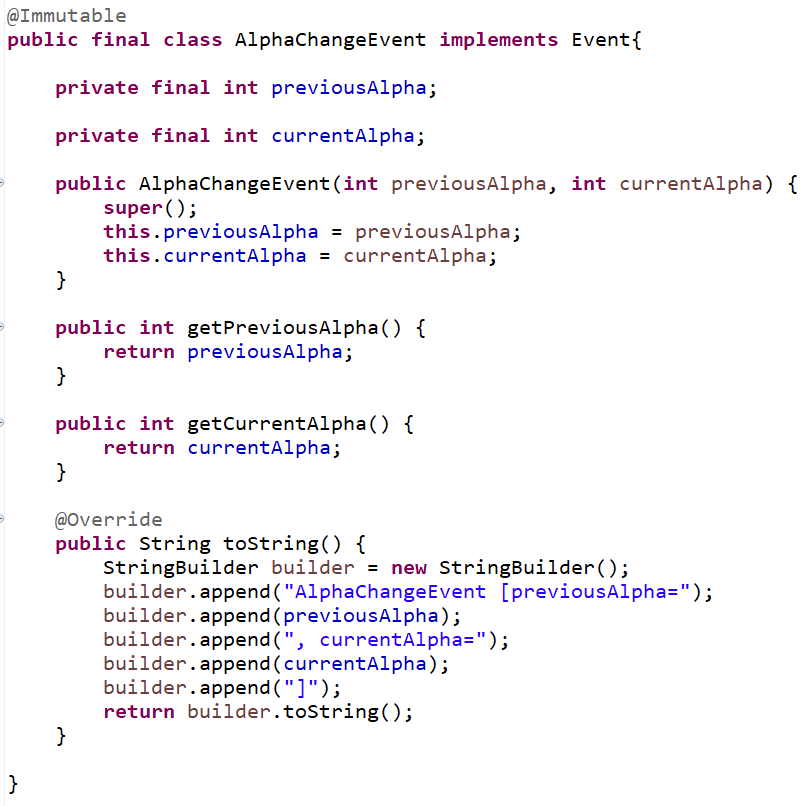
***Figure 2.4.4***



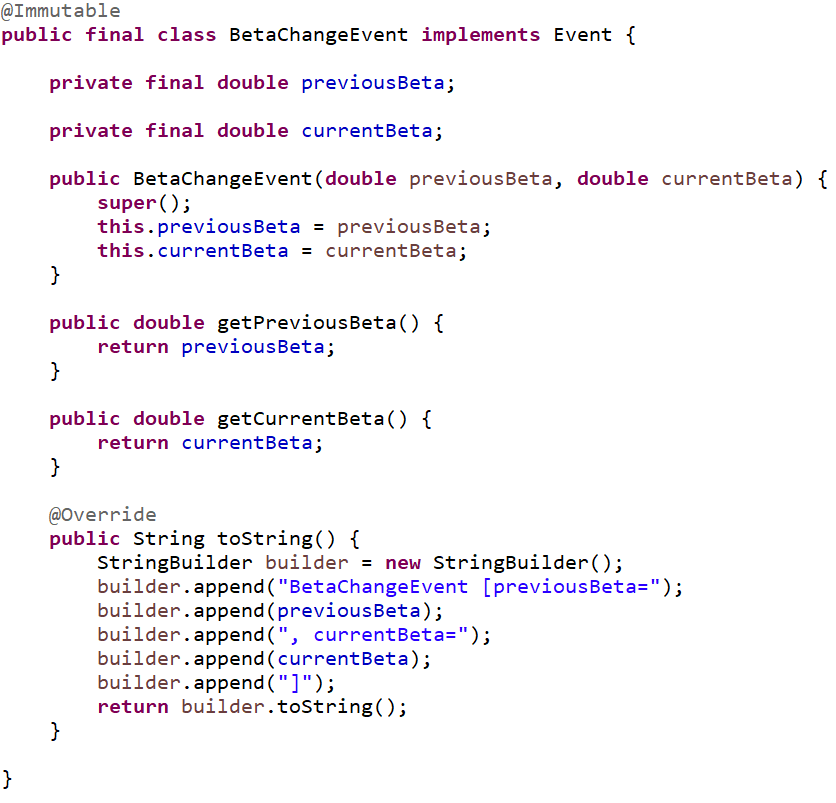
## Lesson 5: Events

An event in GCM is a notification of a data change to the state of a data manager. In this example we will introduce two events corresponding to the two changes to the ExampleDataManager. Both events document the previous value and current value (at the time when the event was generated) and are immutable data classes.

***Figure 2.5.1***

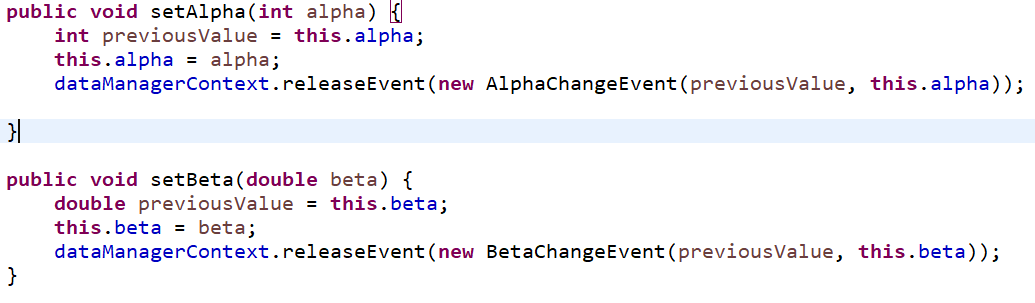


***Figure 2.5.2***



Each is generated by the ExampleDataManager when the alpha or beta values are mutated by releasing the events through the DataManagerContext to the simulation:

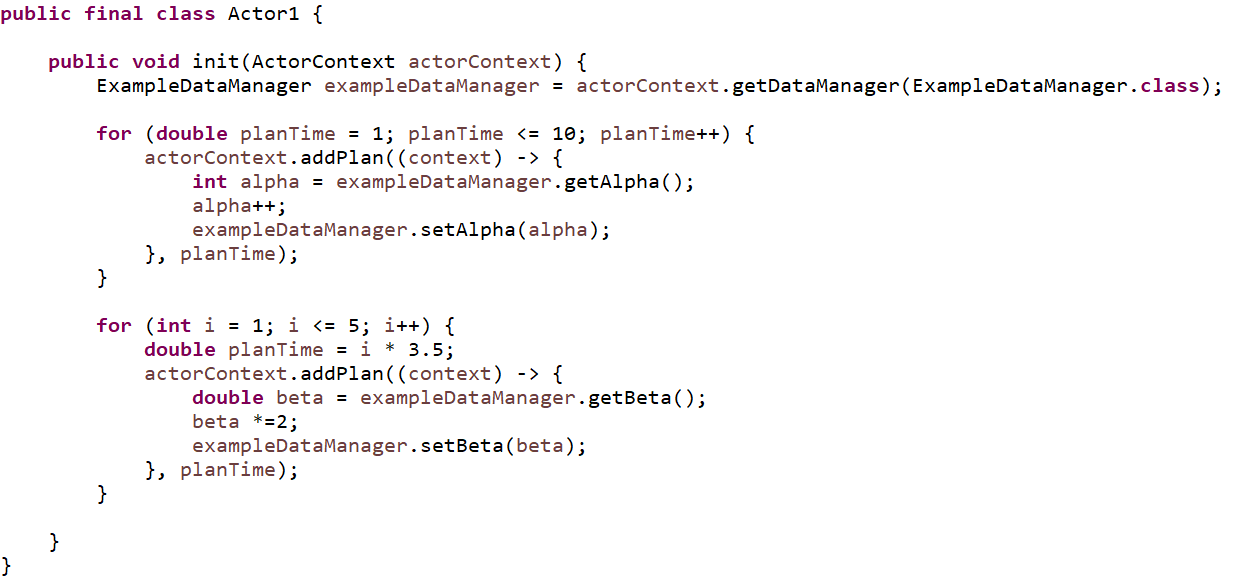
***Figure 2.5.3***



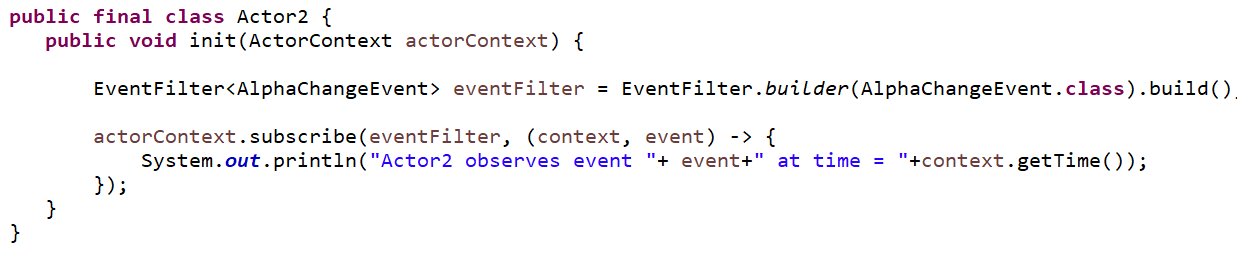
There are three actors in this example:

1. Actor1 makes changes to both the alpha and beta values at 1 and 3.5 day intervals respectively
2. Actor2 subscribes to AlphaChangeEvent events and reports to console what it receives
3. Actor3 does the same for BetaChangeEvent events

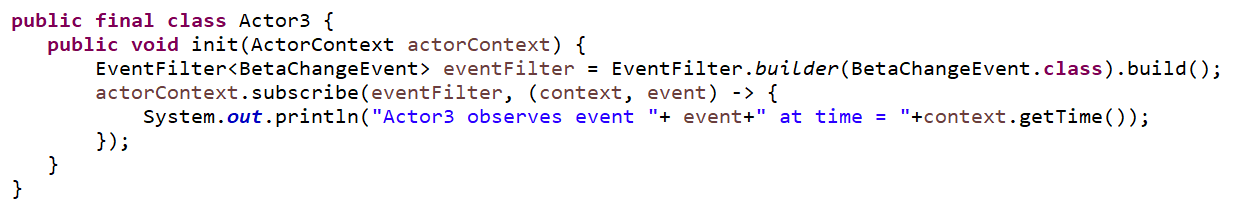
***Figure 2.5.4***

******

***Figure 2.5.5***

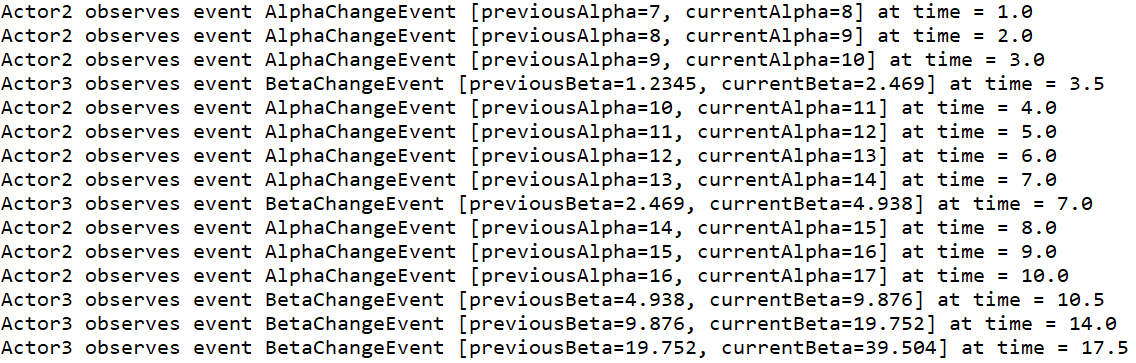


***Figure 2.5.6***



The resulting console output shows Actor2 and Actor3 observing the expected events at the expected times:

***Figure 2.5.7***



### Event Filters

Subscription to events for data managers and actors differ a bit. Data managers subscribe directly to the event type since they are generally interested in all events of some given type. Actors are often more selective and would like a predicate (in Java, the predicate is a function that returns a Boolean) to return true before they handle an event. For example, an actor wants to subscribe for person property change events, but is only interested in those events indicate a change to a particular person property. Since there will likely be dozens of person properties, the actor would get stimulated many times over, only to ignore the event most of the time. Unfortunately, a simple predicate added during the subscription process will not suffice since that predicate would have to be executed for each event and we will have gained little efficiency. Instead, GCM uses the EventFilter class that is essentially a predicate grouping mechanism that allows the subscription engine to group subscribers into a tree structure so that a single predicate execution might suffice to allow an event to be passed to multiple subscribers.

The event filter is logically composed of functions and target values as pairs. Each function takes in an event and releases a value. If that value is equal to the target value, then the event passes that function. An event passes the event filter if it passes all the functions that compose the filter.

The construction of the builder for event filters requires the event class reference. The addition of function-value pairs requires that the functions take in only events of the given class reference, but may return any non-null object value. The simple examples given so far have only specified the event class and thus every event of that type will pass the event filter.

The functions that compose the event filter are often non meaningfully comparable. For example, two functions that return the same values for every event may be separate instances of lambda code that are logically equal, but are not equal in from the point of view of Java. To get around this, each function is associated with an id value and the id and function pair are called an IdentifiableFunction. Two such functions will be equal if and only if their ids are equal without regard to what their functions actually do. Thus is it very important that the mapping of id to actual logical function be stable and the usual best practice is to manage that mapping in a curated manner via the data manager that is associated with the plugin that defines the event. As we examine plugins that define events, we will encounter event filters that are managed by data managers and will not generally generate event filters directly in the actor code.

## Lesson 6: Plugin dependencies

So far we have covered what actors and data managers do and that they are introduced into the simulation via plugins. Over the next lessons we take a closer look at the plugins. This lesson starts with creating a more realistic set of plugins each arranged into separate java packages.

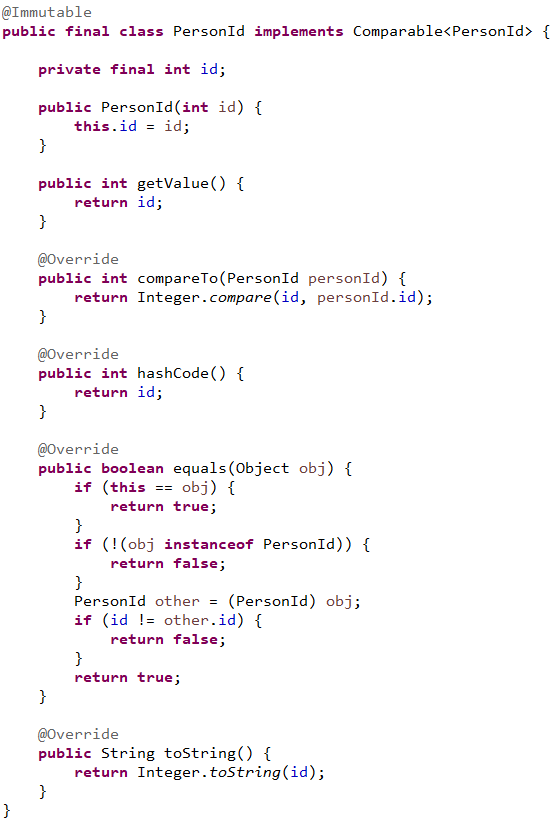
* People plugin
  + Defines a person id
  + Adds the PersonDataManager for tracking people
  + Adds events for the the addition and removal of people
* Family Plugin
  + Defines a family id
  + Adds the FamilyDataManager for grouping people into families
* Vaccine Plugin
  + Adds the VaccineDataManager for tracking which people have been vaccinated
* Model Plugin
  + Contains the ModelActor class to add people organized into family structures and vaccinate some of those people

Here are the classes that implement this example:

### People Plugin:

The people plugin defines a PersonId as a simple, immutable wrapper to an int value. The PersonDataManager tracks people via PersonId values and allows for the addition and removal of people. PersonId values are generated in order and never reused. Events are generated when people are added or removed.

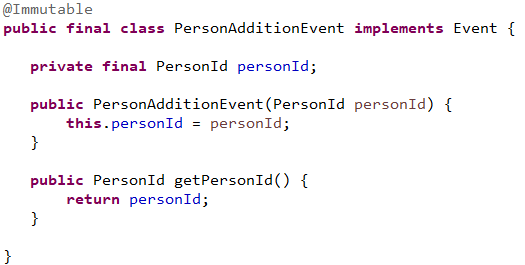
***Figure 2.6.1***



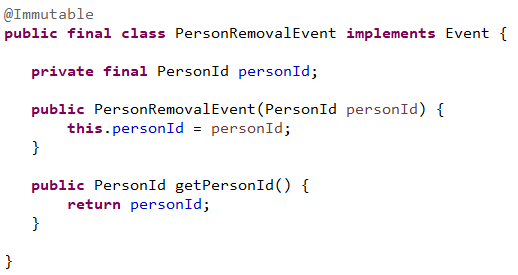
***Figure 2.6.2***



***Figure 2.6.3***



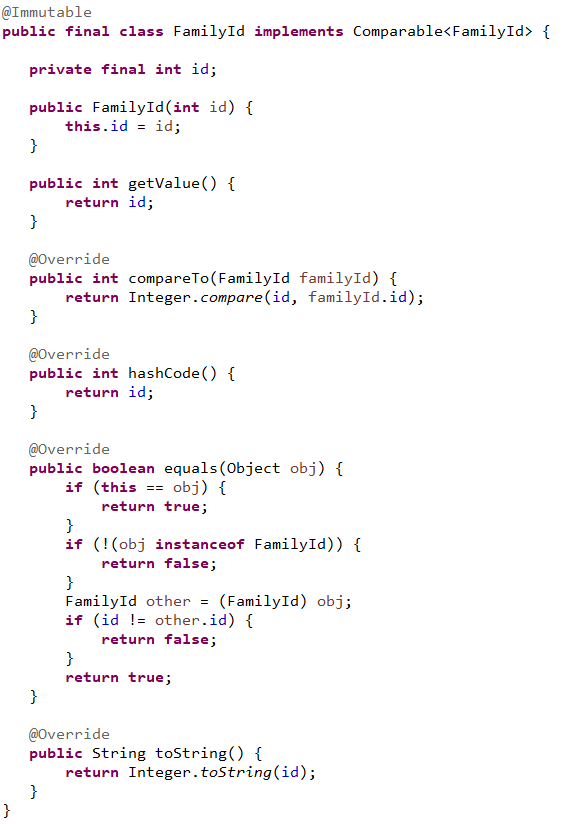
***Figure 2.6.4***



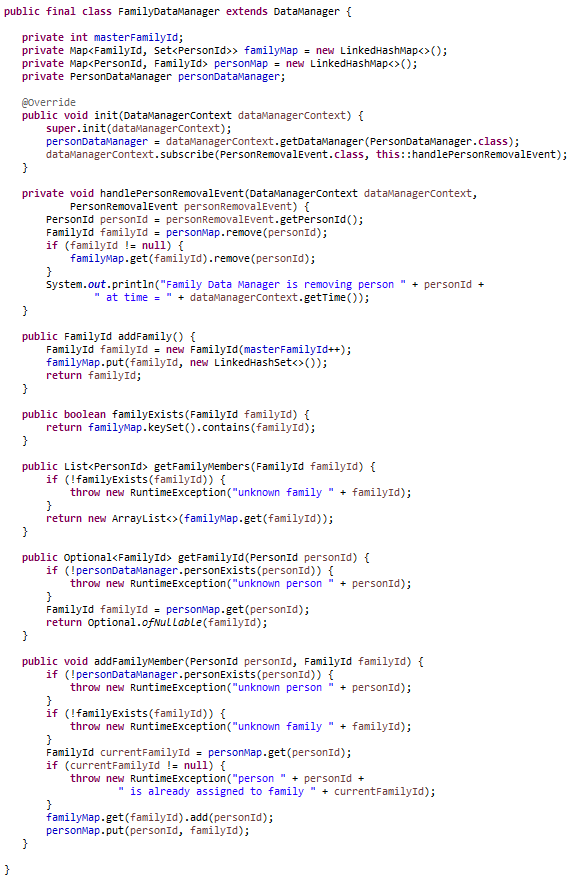
### Family Plugin

The family plugin defines a FamilyId as a simple, immutable wrapper to an int value. The FamilyDataManager tracks family membership via two-way mappings of PersonId to FamilyId. In this example families can only be added and people can only be added to families. However, people can be removed via the PeoplePlugin so the FamilyDataManager subscribes for PersonRemovalEvent and thus removes the people from families.

***Figure 2.6.5***



***Figure 2.6.6***



### Vaccine Plugin

The vaccine plugin contains only the VaccineDataManager which tracks by PersonId which people have been vaccinated. Like the FamilyDataManager, it too subscribes to PersonRemovalEvents and adjusts its data accordingly.

***Figure 2.6.7***



### Model Plugin

The model plugin contains a single actor, the ModelActor, that serves to:

* Add people to the simulation
* Group them into families
* Vaccinate some people
* Demonstrate that events cascade

### Connecting the Plugins

Both the family and vaccine plugins depend on the concept of a person as implemented by the PersonId class. They also need to respond when a person is removed from the simulation and do so by handling the corresponding PersonRemovalEvent generated by the person plugin. We build these dependencies via the Plugin.Builder class in the example code below.

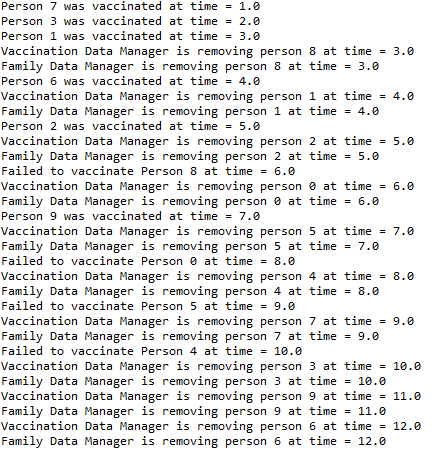
***Figure 2.6.8***



Note the addition of the dependency on the people plugin via its id when adding both the vaccine and family plugins. The order of addition of the plugins to the simulation is relatively unimportant as is ordering in general in any of the builder patterns used in GCM.

The resulting output:

***Figure 2.6.9***

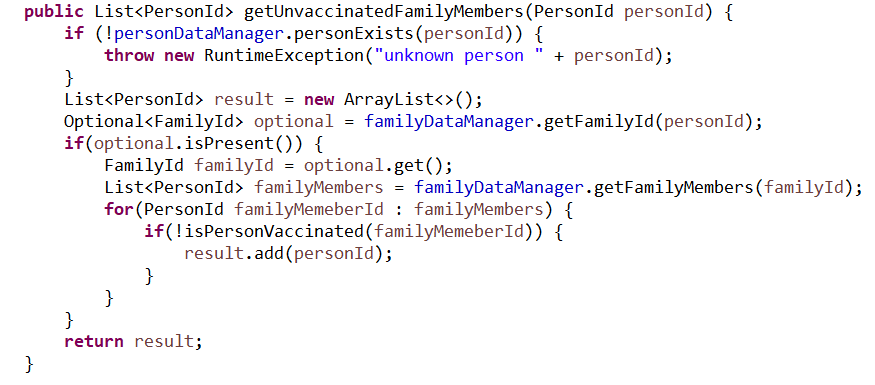


## Lesson 7: Plugin Dependency Graph

We extend the previous lesson by adding an additional dependency of the vaccine plugin on the family plugin. This will allow the VaccineDataManager to answer queries about which members of a family have yet to be vaccinated.

From the VaccineDataManager:

***Figure 2.7.1***



The plugins in this example form a dependency pattern:

***Figure 2.7.2***

people

family

vaccine

All plugin dependencies in GCM form similar directed, acyclic graphs (DAGs). There can be no loops in the dependency graph, but the graph does not have to be fully connected. The dependencies reflect the requirements of the data managers within a plugin to access data managers in other plugins. This pattern drives the order in which events are presented to data managers.

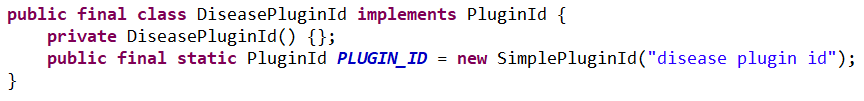
In this lesson, the VaccineDataManager and the FamilyDataManager have both subscribed to the PersonRemovalEvent generated by the PersonDataManager. Since the VaccineDataManager also has a dependency on the FamilyDataManager, the VaccineDataManager should receive the event after the FamilyDataManager. Events cascade through the subscribed data managers in an order that is consistent with the plugin dependency DAG.

## Lesson 8: Plugin Data

The Example code in the last lesson was a bit verbose and can be improved. Identifying and generating the plugins can be included in the plugin packages by introducing classes for each id and classes for each plugin’s contents.

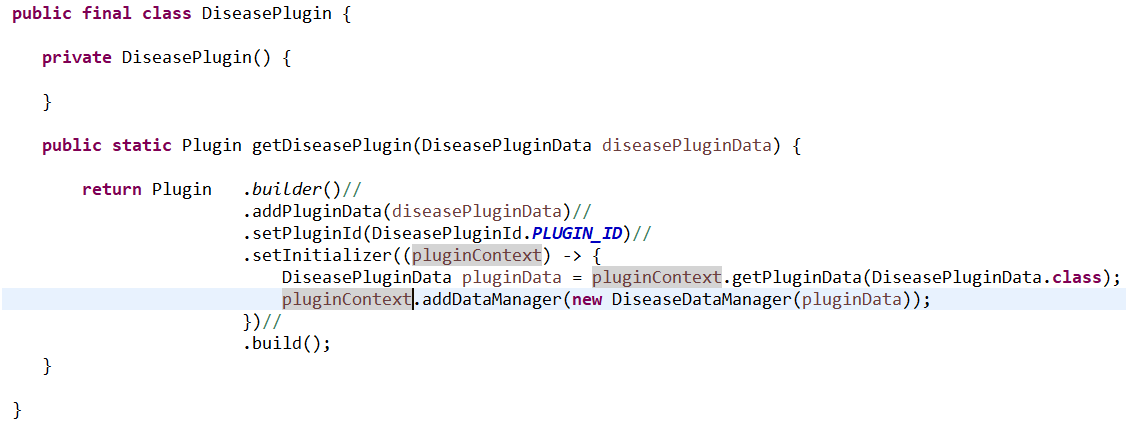
In the disease package we add a unique plugin identifier with a final static id field:

***Figure 2.8.1***



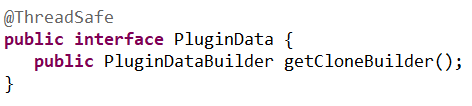
We also add a static class (DiseasePlugin) that implements the construction of the plugin from the required plugin data.

***Figure 2.8.2***



The plugin is initialized with a DiseasePluginData object that contains the initial values for r0, asymptomatic days and symptomatic days. Most plugins will have a single plugin data object, but some may not need any and some may be designed with multiple such classes. All such classes must implement the PluginData interface:

***Figure 2.8.3***

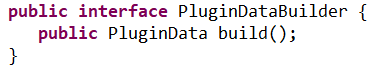


Plugin data classes must be threadsafe since they will be shared between multiple simulations running on separate threads. This stands in contrast to the actors and data managers which are created and managed in the thread of a single simulation. The best practice is to make plugin data classes immutable since immutable classes in Java are guaranteed to be threadsafe. For a class to be immutable in Java it must meet three conditions:

1. It cannot be mutated, i.e. it has no setters and no public fields.
2. All its fields are marked final.
3. Its constructor(s) do not pass reference to self. No reference to the newly created object leaks out before construction is complete.

Besides carrying whatever data is needed by the plugin, the PluginData implementor must provide a PluginDataBuilder:

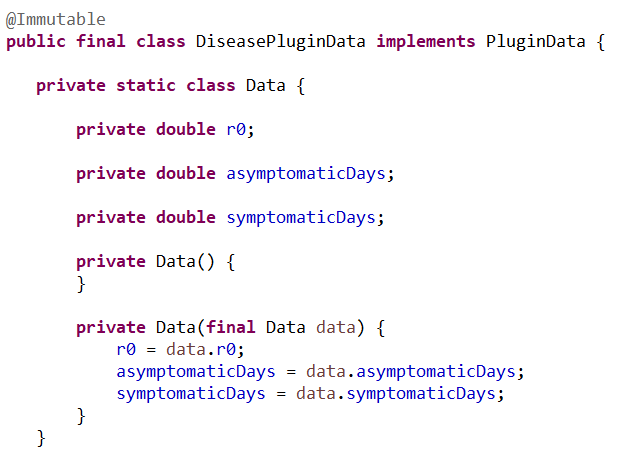
***Figure 2.8.4***



The role of the plugin data builder will be explored in the next lesson where it will be used to make alterable copies of plugin data to drive the experiment. For now, let’s examine the DiseasePluginData class. It is composed several sections:

* A data class
* A static builder class
* A single data field and private constructor
* Getter methods for the data
* A clone builder method

***Figure 2.8.5***



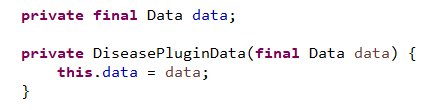
The Data class is private and just contains the fields needed by the plugin. Note that it is a mutable class and that its fields are not final. It will be used by the builder class later to store values. Its constructors are private and allow one Data object to be copied from another.

***Figure 2.8.6***



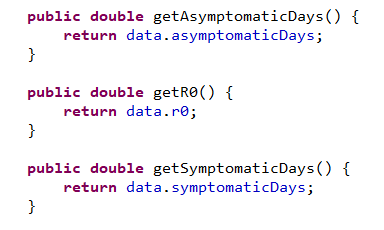
The static builder class is used instead of a constructor. The use of builder classes for plugin data objects is key to the creation of experiments covered in the next lesson. For now, let’s concentrate on what the builder does. First, it has setter methods for each of the data fields and each such method returns the builder instance to support method chaining. Next, the build() method returns the DiseasePluginData via a try finally block so that if anything goes wrong, the builder instance will uncorrupted and ready for reuse. Note that after invoking the build method, the data object gets replaced with a fresh version. Finally, the builder’s own constructor is private and is accessed via a static method. This is done to grant a syntax that is more compatible with the method chaining. We will defer discussion of the private dataIsMutable field until the end of this section.

***Figure 2.8.7***



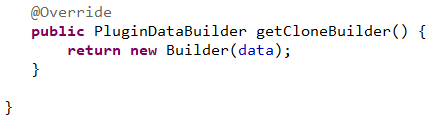
After the builder collects the data, it passes that data to the instance of the DiseasePluginData which is stored as a final field. Recall that the field must be final in an immutable class.

***Figure 2.8.8***



The getter methods for each field value in the data are added. There are no corresponding setter methods.

***Figure 2.8.9***



We end the class with the getCloneBuilder method.

*Our use of the term clone is intuitive but may cause some confusion. What we are doing is copying the data in the DiseasePluginData and placing into a builder so that it can be further mutated later in the experiment. Java formally defines the term clone as a part of the Object class definition and implements it with a protected method clone(). Use of the Object.clone() method has generally fallen out of favor in Java but still has some proponents/use cases.*

The method returns a new Builder that has reference to the current data object. The builder starts out with the private field **dataIsMutable** set to false. This is to ensure that we do not pay the cost of copying the data object if we never invoke any of the setter methods before invoking **build()**; While such considerations seem trivial with only three values, in most models there will be plugin data classes that contain millions of data values.

The resulting example class is easier to read and more succinct:

***Figure 2.8.10***



## Lesson 9: Experiments

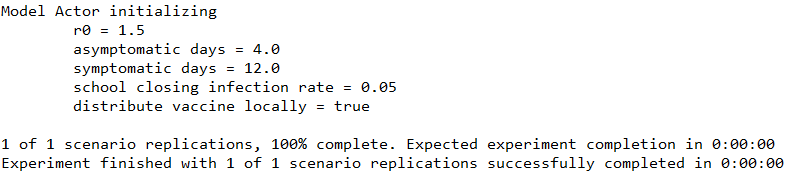
So far we have mentioned that the plugin data classes play a role in executing an experiment via the **getCloneBuilder** method. Let’s start with the simple experiment. We will update the last example class by replacing the Simulation execution with an Experiment execution:

***Figure 2.9.1***



The experiment class has a very similar builder to the Simulation class so we only have to swap out the Simulation reference for an Experiment reference. The resulting execution created an experiment containing exactly one simulation that runs in the main thread. However, the output contains information about the status of the experiment.

***Figure 2.9.2***

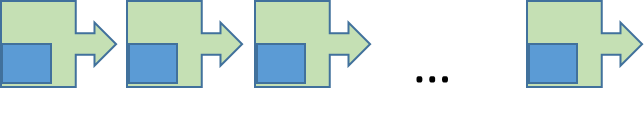


### What happens when the experiment executes?

You have contributed several plugins to the experiment and on execution the experiment generates multiple simulation runs on multiple threads. Let’s examine how this is accomplished as a way to motivate this lesson’s code examples.

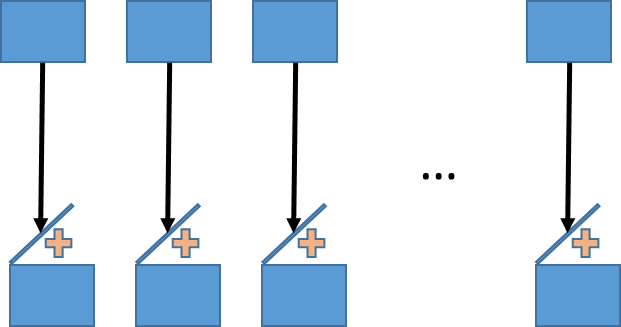
The experiment is composed of several plugins, each with zero to many plugin data objects. For purposes of the diagrams we will assume that each plugin has a single plugin data object.

***Figure 2.9.3***

**plugins**

The experiment gathers the plugin data objects and gets the plugin data builder for each. These plugin data builders will come pre-filled with the data from the original data objects.

***Figure 2.9.4***



By altering the data in these builders, we generate new scenarios for the simulations to execute. GCM manages the instructions to alter the plugin data via Dimensions. Each dimension contains one to many levels.

***Figure 2.9.5***



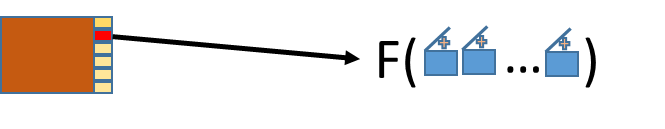
For example, we may have a dimension that alters the value of **alpha** from **plugin data A** and the value of **beta** from **plugin data B**. Each level in the dimension will set specific values for alpha and beta via the builders.

***Figure 2.9.6***

|  |  |  |
| --- | --- | --- |
| level | alpha | beta |
| 0 | 2.3 | FALSE |
| 1 | 3.6 | TRUE |
| 2 | 4.8 | FALSE |

Each level in a dimension is actually a function that takes in the builders and manipulates the content of each plugin as needed.

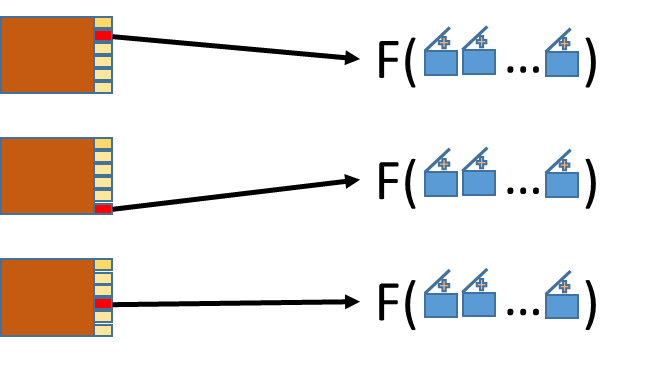
***Figure 2.9.7***



Consider an experiment with two dimensions having 3 and 5 levels respectively. The number of level permutations is 3x5 = 15. Each such permutation is referred to as a scenario and the scenarios are numbered from 0 to 14.

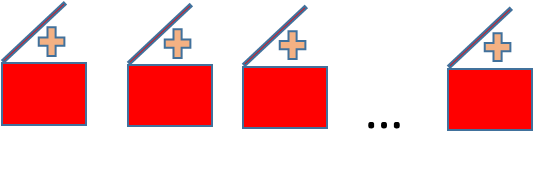
As the experiment executes, it works with each scenario id and determines for that id which levels are active for each dimension.

***Figure 2.9.8***



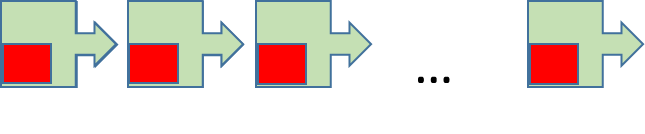
Each level (via its function) alters the contents of the builders in turn, resulting in a unique set of content for that scenario.

***Figure 2.9.9***



The builders are then instructed by the experiment to build the plugin data objects. The resulting data objects are inserted into copies of the original plugins to produce a unique set of altered plugins that are specific to the scenario id and executed via a single simulation instance.

***Figure 2.9.10***



You may have noticed that the initializer code above acquires the DiseasePluginData via the context rather than the instance passed to the getDiseasePlugin() method. This is a necessity due to experiment design and will be covered in the lessons that follow. **In general, the initializer code should always retrieve plugin data from the plugin context.**

We expand the example by adding a single dimension that set r0 to two values, generating two simulations.

***Figure 2.9.11***



In the dimension we see that there are two levels and the addition of some meta data in the addMetaDatum(“r0”) invocation. The meta data here represents the information that each level is altering in the experiment. The main purpose of each level is to alter the state of a builder(s) but must also return meta data values to match the meta data for the dimension. The meta data of the dimension acts as a header to a table while the meta data for each level are the values in that table.

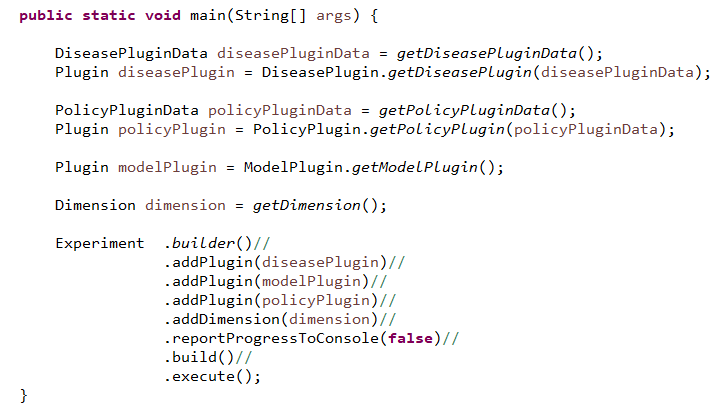
The building of the dimension can be streamlined without typing out each level:

***Figure 2.9.12***



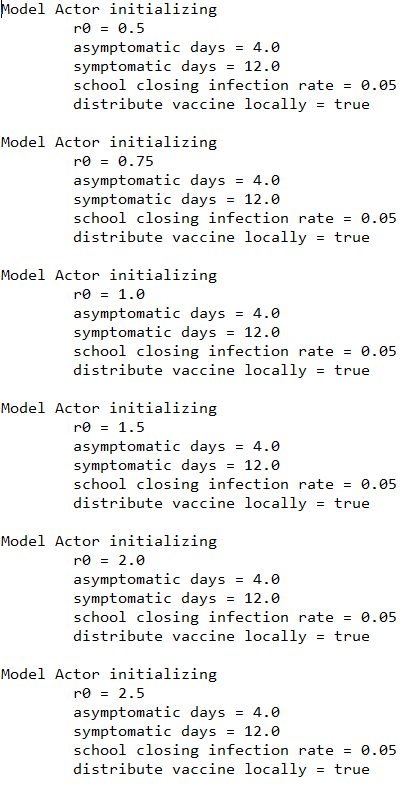
The resulting experiment execution is more streamlined:

***Figure 2.9.13***



We have turned off the experiment report progress to console in the code above. We have chosen six values for r0 in our dimension and thus we have 6 simulation executions, each having the model actor print out the contents of the DiseaseDataManager:

***Figure 2.9.14***



We are extending the example again, reducing the r0 dimension to just three levels and introducing a dimension over the policy data. This new dimension has four levels controlling local vaccine distribution and school closing infection rates:

***Figure 2.9.15***



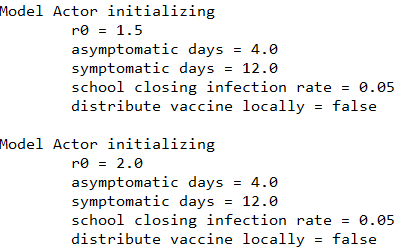
We add the new dimension to the experiment:

***Figure 2.9.16***



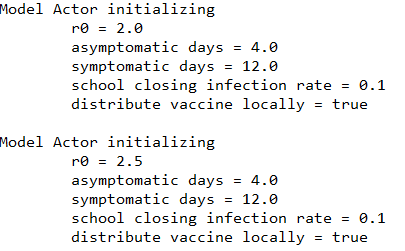
The result is now 12 executed scenarios:

***Figure 2.9.17***



…

***Figure 2.9.18***



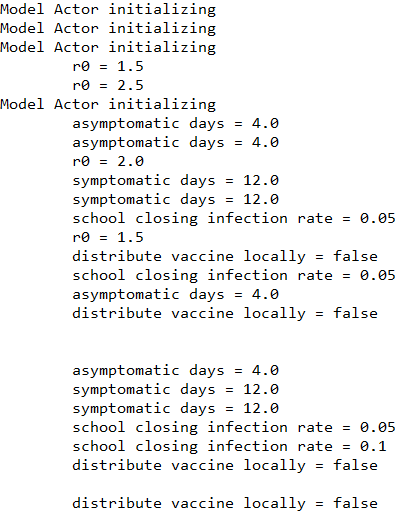
So far, the experiment has run in a single thread. We now run it in four threads by adding a single line to the experiment giving is a thread count:

***Figure 2.9.19***



The experiment runs in the main thread and the scenarios now run the four additional threads. The resulting console output a bit jumbled since the writes to the console are now coming from four simultaneous simulation runs:

***Figure 2.9.20***

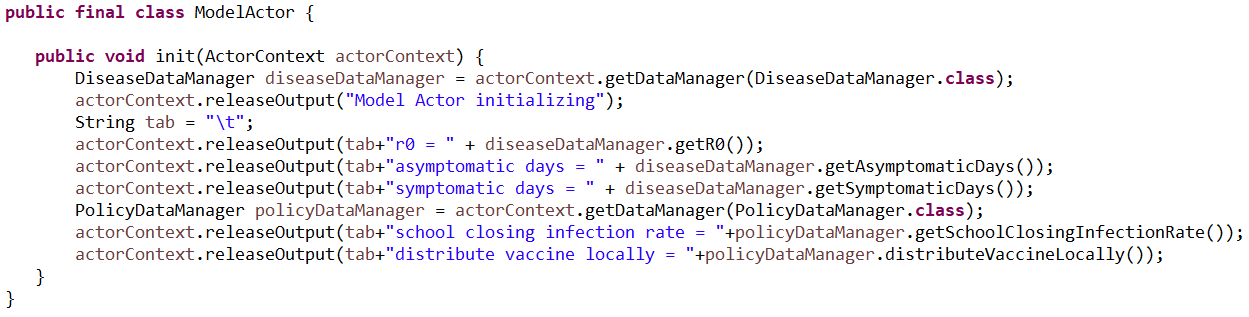


We will alleviate this problem as we explore how the simulation and experiment manage output.

## Lesson 10: output

So far we have only produced output by writing directly to the console in the various actors and data managers. The simulation contexts (ActorContext/DataManagerContext) provide for the release of output objects to an external handler (outside the simulation). In this lesson, the ModelActor class has been altered to use this mechanism:

***Figure 2.10.1***



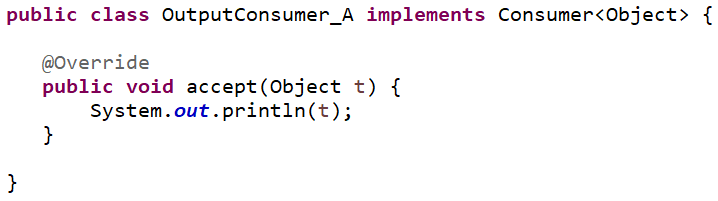
Data managers can release output in a completely similar way. The output objects are handled by an external handler presented during the build of the simulation:

***Figure 2.10.2***



Released output objects are sent to the output consumer. In the current example, that consumer is an instance of the class OutputConsumer\_A and it simply prints the object to the console:

***Figure 2.10.3***



At first glance this mechanism seems simple and not particularly useful. In practice, one rarely uses the simulation directly and instead favors the experiment which has a somewhat more sophisticated handling of output. With experiments, GCM is potentially using multiple threads to execute each simulation, so output handling must be threadsafe.

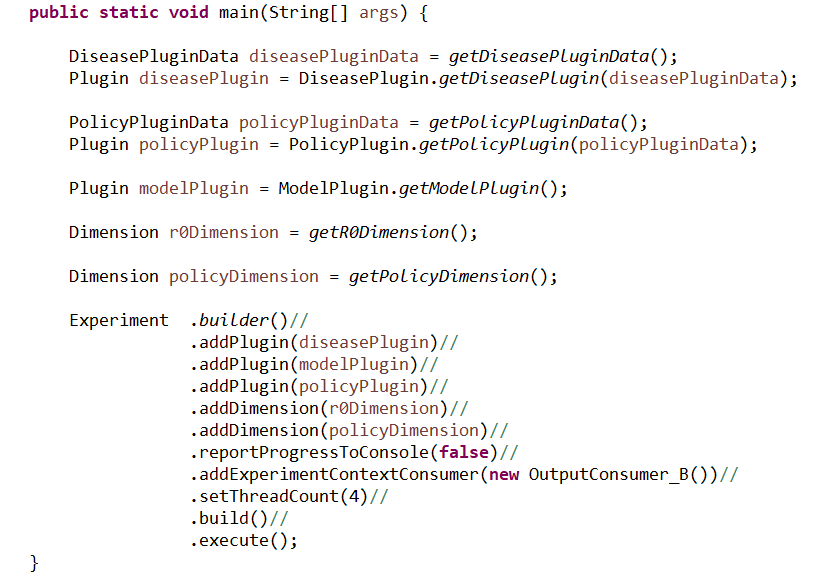
### Experiment Context

Just as the simulation supplies contexts, the experiment uses the ExperimentContext to give output consumers a view into the ongoing experiment. It gives each output consumer several capabilities:

* Subscription to output by output class type
* Subscription to the opening and closing of the experiment
* Subscription to the opening and closing of each simulation
* Scenario status information
* Experiment and Scenario meta data

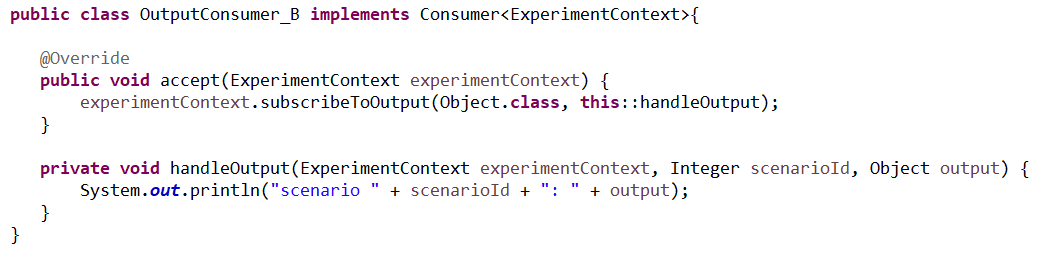
In Example\_10\_B, we bring back the dimensions from previous lessons and will excerpt just the main method:

***Figure 2.10.4***



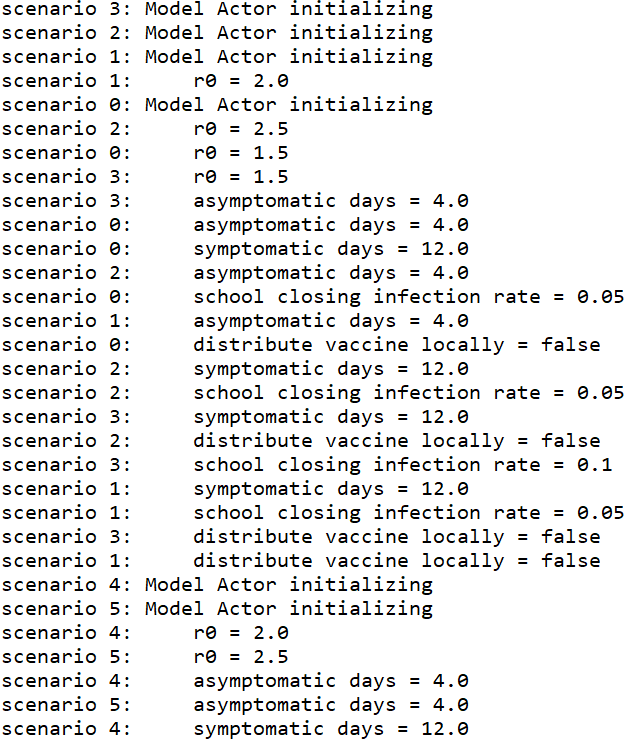
Like the simulation, the experiment is adding a consumer for output, but this time that consumer is “consuming” an experiment context. Once the consumer receives that context, it will use it to further subscribe to output and various experiment level events.

***Figure 2.10.5***



The experiment can have any number of ExperimentContext consumers and initializes each at the beginning of its execution via the accept() method. In OuputConsumer\_B, the only action the consumer takes is to subscribe to all output and have that output handled by the handleOutput() method. The resulting output shows the scenario id for each line:

***Figure 2.10.6***



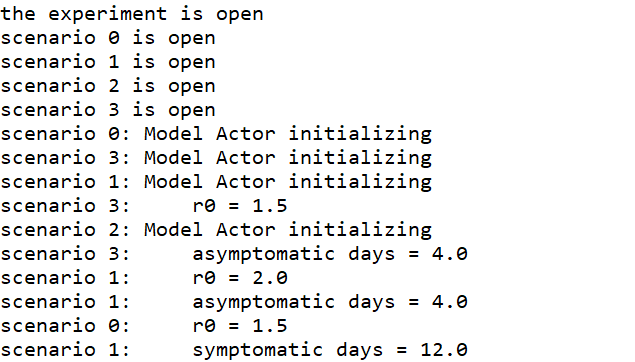
Example\_10\_C switches the experiment context consumer to an instance of OuputConsumer\_C which subscribes to all output types as well as the opening and closing of the experiment and all simulations (scenarios):

***Figure 2.10.7***



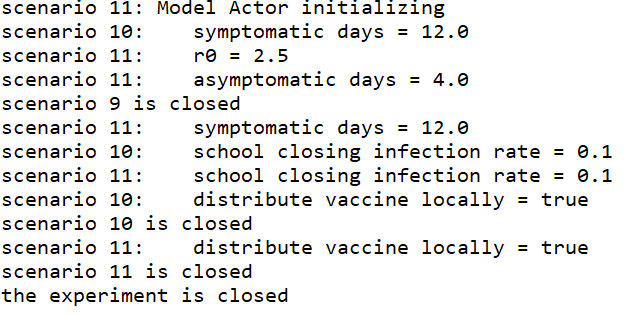
The resulting output shows the usual released output along with the opening and closing of each simulation:

***Figure 2.10.8***



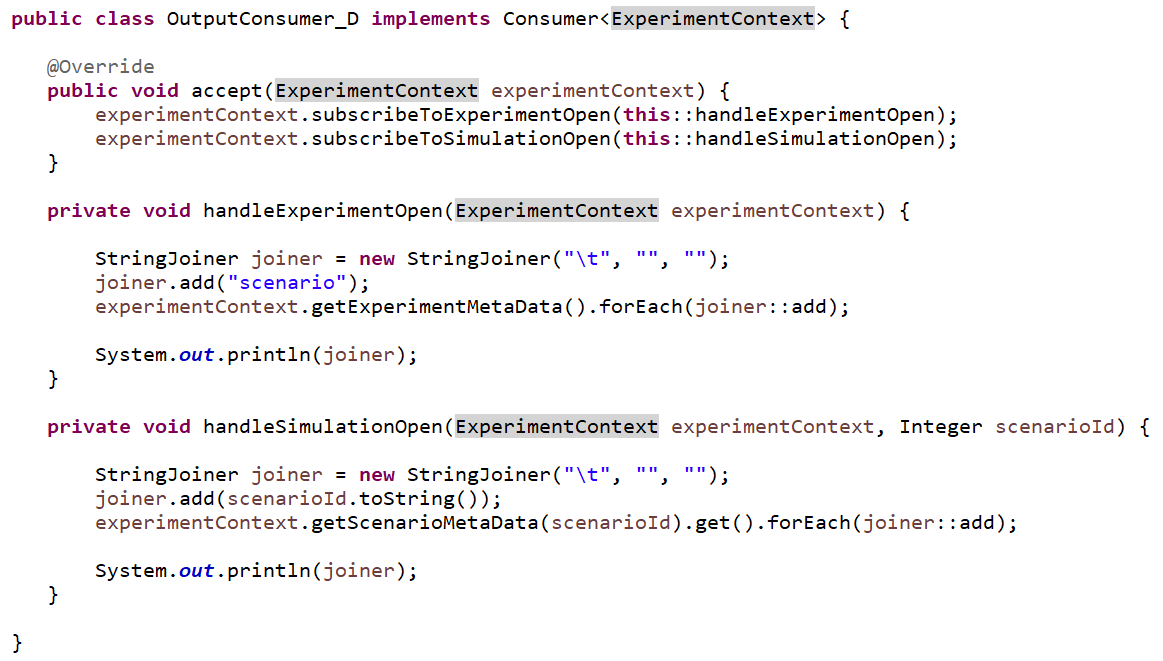
…

***Figure 2.10.9***



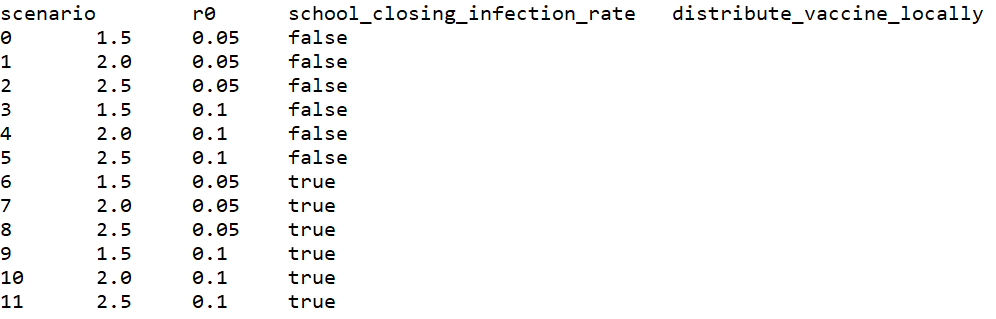
In the final example, OuputConsumer\_D, we drop the output handling and demonstrate that the meta data used to build the dimensions of the experiment can be retrieved from the experiment context and used for reporting:

***Figure 2.10.10***



The resulting output shows for each scenario the meta-data that defines that scenario:

***Figure 2.10.11***



Recall that as the experiment executes, it utilizes multiple threads to execute the individual scenarios. Thus every experiment context consumer must be threadsafe. We have accomplished this by making each such consumer stateless. In practice, it is often necessary for experiment context consumers to be stateful and this can involve careful consideration of the use of synchronization and other concurrency issues. Fortunately, GCM provides a reporting plugin that deals with these issues and provides a general method for producing tabular reports.

# Chapter 3: Stochastics Plugin

The stochastics plugin provides for the management of random number generators. It contains a default random number generator (RNG) as well as any number of RNGs associated with identifiers.

### Plugin Data Initialization

The plugin is initialized using a StochasticsPluginData object that collects a starting seed value for the default RNG as well as any number of RNG identifiers. These identifiers are implemented via the RandomGeneratorId interface which only specifies that such an identifier have a non-null, non-empty and stable implementation of the Object.toString() method.

### Plugin Behavior

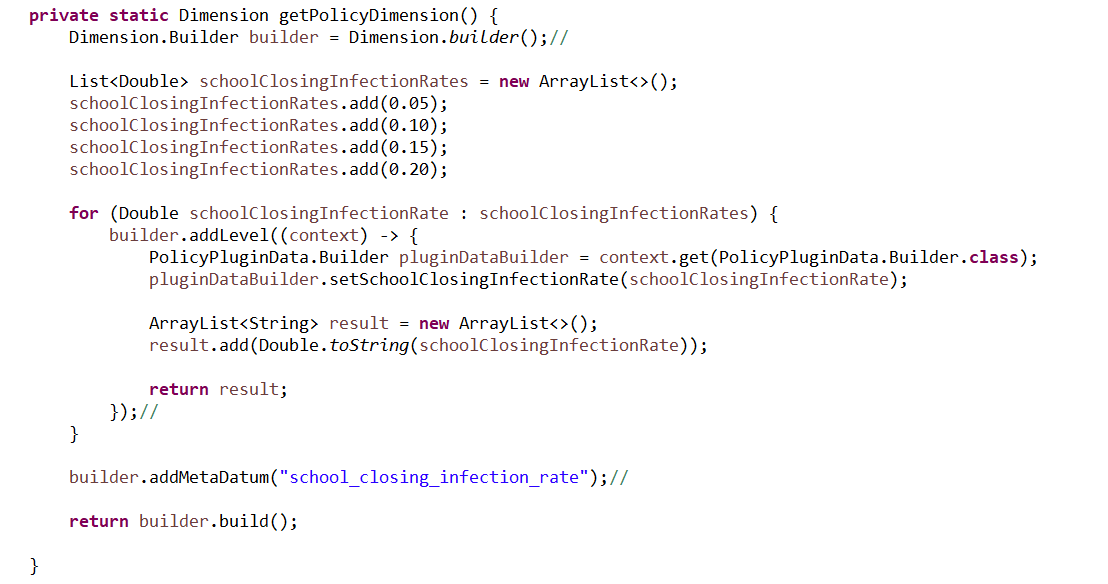
The plugin adds a single data manager to the simulation as an instance of the StochasticsDataManager that is initialized with the StochasticsPluginData.

### Data Manager

The data manager provides access to its RNGs via various getter methods. The RNGs are implemented as org.apache.commons.math3.random.Well44497b instances and are subject to seed manipulation. RNG seed management should be left to the data manager and never directly managed by any client. The default RNG is seeded with the plugin data’s seed. All other RNG’s have their seeds set to a function of their associated ids: rngSeed = id.toString().hashCode()+ pluginData.seed().

Our first example lesson uses the disease, model and policy plugins again. This time we will have the single ModelActor schedule three random times to set the R0 value to a random number between 1 and 2. Four scenarios will result from a policy based dimension that alters the school closing infection rates, which will not influence the ModelActor.

***Figure 3.1.1***

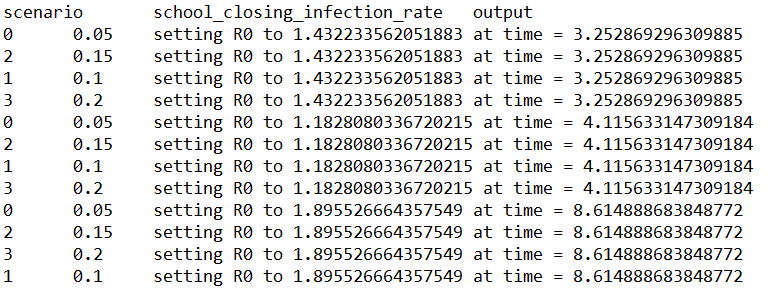


***Figure 3.1.2***



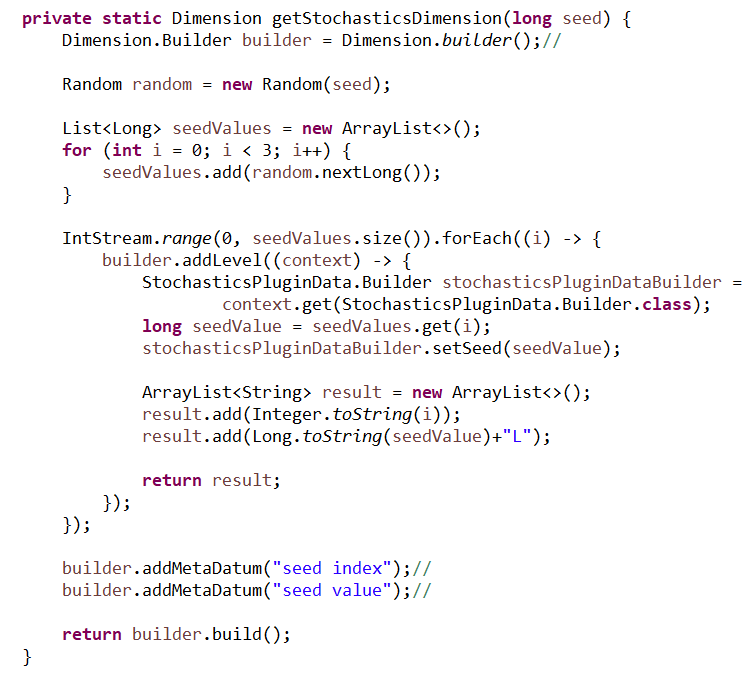
The stochastics plugin is initialized with a seed value of zero and that seed will be used in each scenario as the initial seeding for the default random generator. Thus we expect that each scenario will have identical output.

***Figure 3.1.3***



Our next example lesson adds a dimension used to alter the initial seed value of the stochastics plugin data to one of three values. Combined with the policy dimension, this will result in 12 scenarios.

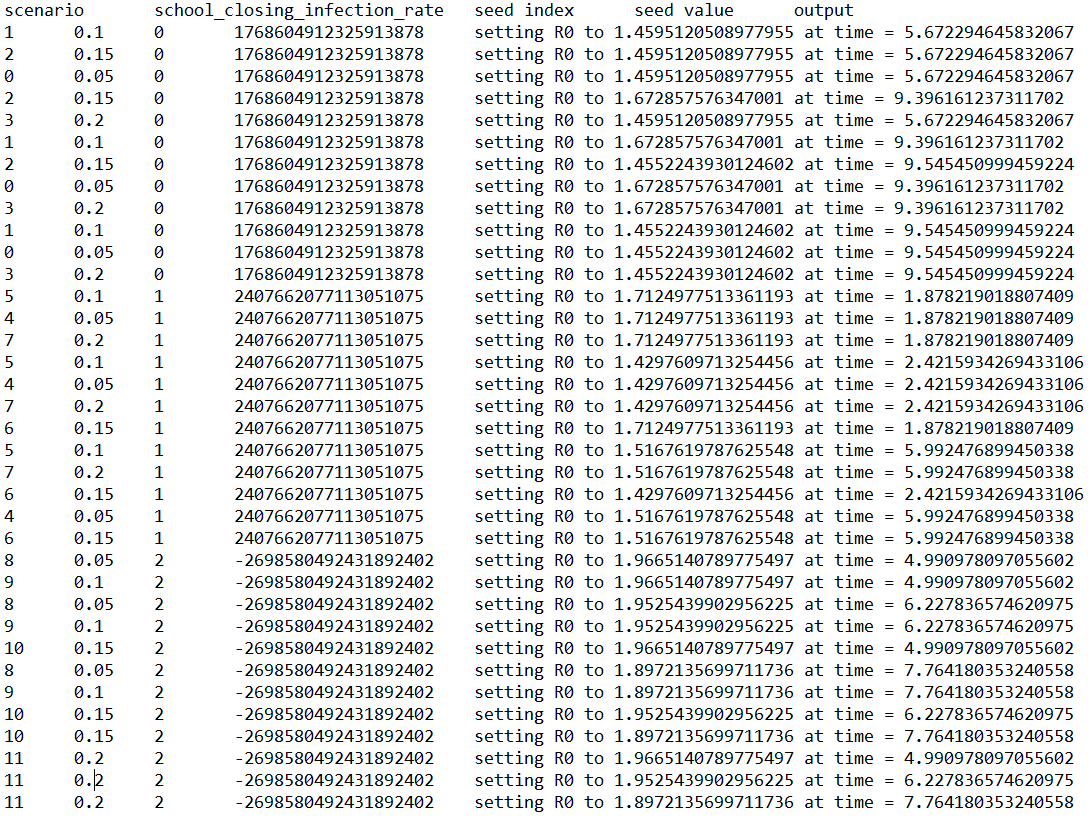
***Figure 3.1.4***



***Figure 3.1.5***



The resulting output shows the varying random number generation:

***Figure 3.1.6***

# Chapter 4: Reports Plugin

The reports plugin implements an experiment context consumer that records output into tab-delimited text files via the java.nio library using blocking file writes. Three new concepts form the core of the reports:

1. Report Id – a unique identifier for each report
2. Report Header – the header content for the report
3. Report Item – the data content(strings) for each line in the report

The report id is a unique identifier used to mark every report item that the plugin processes and helps associate each item to the specific file where it will be recorded. Report items are a flexible list of values that have an associated report id as well as a report header used to build the header of the file. A report file is built from the report items that are associated with a specific report id in the order received. The first report item is used to build the header of the report file. All other lines of the file are an ordered, tab-delimited listing of the string values contained in each report item. No attempt is made to ensure that the header matches the report lines or that all report lines have equal field lengths.

While any data manager or actor can release report items, in practice most reports are managed solely by special purpose, passive actors that simply observe events and do not act on any data manager. These report actors often subscribe to multiple event types and aggregate several events into a single report item. This reduces the amount of information recorded in output files and thus output files are often more complex than simple event trace files.

### Plugin Data Initialization

The plugin initialization data contains the initialization logic of report actors. Recall that plugin initialization data is shared across multiple scenarios in an experiment and that experiments can execute in multiple threads. This requires that this initialization logic be threadsafe. We achieve thread safety by contributing suppliers of consumers of ActorContext. Formally, these are Supplier<Consumer<ActorContext>>. When a particular scenario is executed in a simulation’s thread, each such supplier is invoked to produce the new consumer of ActorContext. Generally, this will be accomplished by having the supplier generate a new instance of the relevant report actor class with thread-safe content.

### Plugin Behavior

The Plugin processes the reports plugin data at the beginning of each simulation and adds an actor for each supplier of consumer of actor context.

### Experiment Context Consumer

So far we have seen that reports tend to be produced by specialized actors and that those actors can be added to the simulation via the plugin initialization data. This covers the production and release of the report items from each simulation instance but not what happens to the report items afterward. The output files that receive the report items must work with multiple threads. We manage this with a threadsafe experiment context consumer, the NIOReportItemHandler, that is added to the experiment. The NIOReportItemHandler is created via a builder pattern that allows the modeler to associate report ids to file paths.

### Example Reports

We reach back to the previous lessons where we introduced plugins for people, families and vaccines for a demonstration of reports. The reports will center on the vaccination of families in various forms and are implemented by three dedicated actor classes in the vaccine plugin:

* FamilyVaccineReport – Immediate reporting based on observed events
* HourlyVaccineReport – Hourly reporting based on observed events
* StatelessVaccineReport – Hourly reports based on inspection of current state

### General Setup

This example uses the following plugins

* Person Plugin – provides containment for person identifiers
* Stochastics Plugin – (GCM plugin) provides random number generation
* Reports Plugin – (GCM plugin) provides reporting mechanisms
* Family Plugin – defines families and associates people with families
* Vaccine Plugin – maintains vaccine assignments with people and families and defines the three reports
* Model Plugin – provides an actor for loading the initial population and an actor for scheduling vaccinations

The general flow of action in the simulation is that the PopulationLoader actor will add people and families to the simulation based on the initial plugin data provided in the family plugin. The VaccineScheduler actor will then schedule people at random times to be vaccinated. As people and families are created, people join families and people are vaccinated, the various data mangers will generate the relevant events for observation by the three report actors. The report actors will observe these events and correspondingly generate report items that will flow out of the simulation into the experiment level report mechanisms that will result in report files being written.

Let’s examine Example\_12. In figure 4.1 we see that the plugins are generated with the person, vaccine and model plugins requiring no input data. The stochastics plugin is generated with a fixed seed value. Next, the family plugin is created with initial data specifying that 30 families will be created and that each family will have a random number of members up to 5 people.

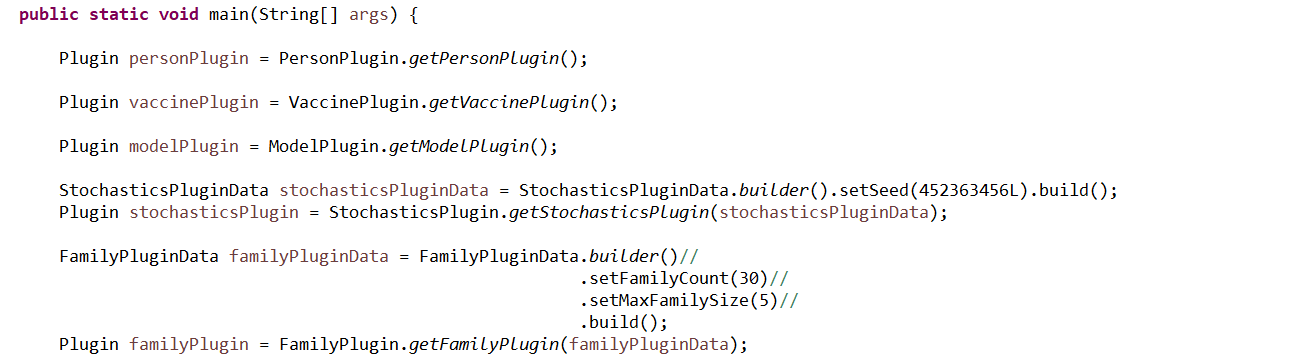
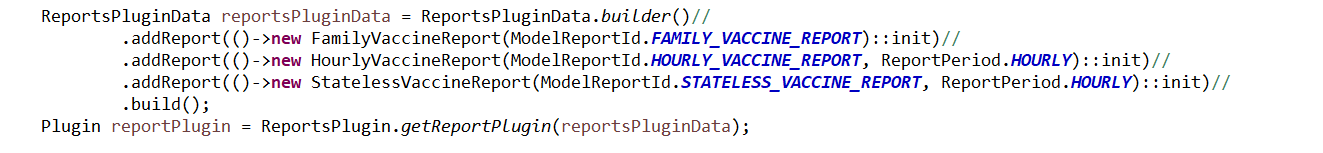
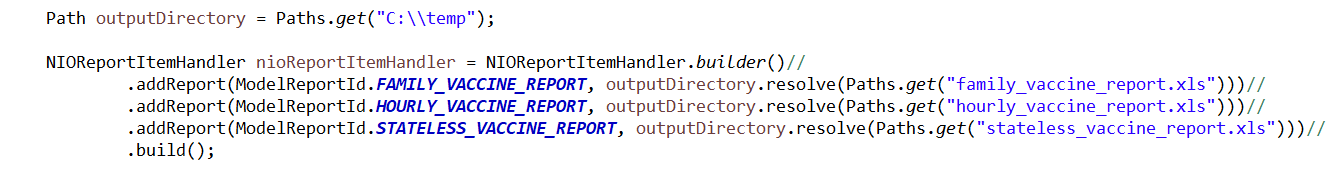
***Figure 4.1***

Figure 4.2 continues with the addition of the three report actors. These actors are defined in the vaccine plugin, but that plugin does not add them to the simulation since report actors are created at the discretion of the modeler based on their analytic needs. Thus the control of report actor creation is generally left to the report plugin. Since the report plugin was written without knowledge of how to implement these actors, we must provide the construction mechanisms. Further, each simulation instance will have to create these actors in its own thread, so a threadsafe mechanism must be employed. A new instance of the report is created with each invocation of addReport. Since the report id is an enumeration (and thus immutable) the total construction is threadsafe.

***Figure 4.2***

So far we have given the reports plugin three report actor classes and each simulation instance will create those report actors. The report actors will in turn generate report items that will be released to the experiment level. In figure 4.3 we now indicate to the experiment how to distribute those report items to files.

***Figure 4.3***



Each report id is now associated with a particular file path. Although each file is a tab-delimited text file, we use the .xls file extension so that they can be automatically opened as a spreadsheet. Had we skipped adding these last specifications, the report items would flow out of the simulation and into the experiment but would not find an associated file and would be ignored.

Finally, in Figures 4.4 and 4.5, we create a single experiment dimension that will override the maximum family size with four values and thus create four scenarios for the experiment.

***Figure 4.4***

***Figure 4.5***



### The Family Vaccine Report

The first report documents the changes in the number of families that are vaccinated over time as individual people receive the vaccine. The field headers for the report are:

* scenario – the id of the scenario
* max\_family\_size – the maximum family size dictated by the scenario
* time – the time in days for each item in the repot
* unvacinated\_families – the number of families that have no members vaccinated
* partially\_vaccinated\_families – the number of families that have at least one, but not all members vaccinated
* fully\_vaccinated\_families – the number of families that have all members vaccinated
* unvaccinated\_individuals – the number of people who are unvaccinated and have no family assignment
* vaccinated\_individuals – the number of people who are vaccinated and have no family assignment

The experiment report mechanisms are responsible for reporting the scenario and the max\_family\_size fields since they are part of the experiment design. The remaining fields are contributed by the report actor. Note that family membership is not guaranteed and that some people may not be associated with any family id. The report accounts for these people in the last two fields.

There are four events that drive the report:

1. the addition of a person to the simulation
2. the addition of a family to the simulation
3. the assignment of a person to a family
4. the vaccination of a person

Note that the model logic does not allow for the removal of a person from the simulation, the removal of person from a family or loss of vaccination coverage for a person. In a more nuanced model, there would likely be more events that would influence the report actor.

The FamilyVaccineReport has several private fields and class for maintaining the five counts of the reports. In figure 4.6 we have the list of two convenience enumerations for families and individuals that help with the creation of the report header and with maintaining counts.

***Figure 4.6***

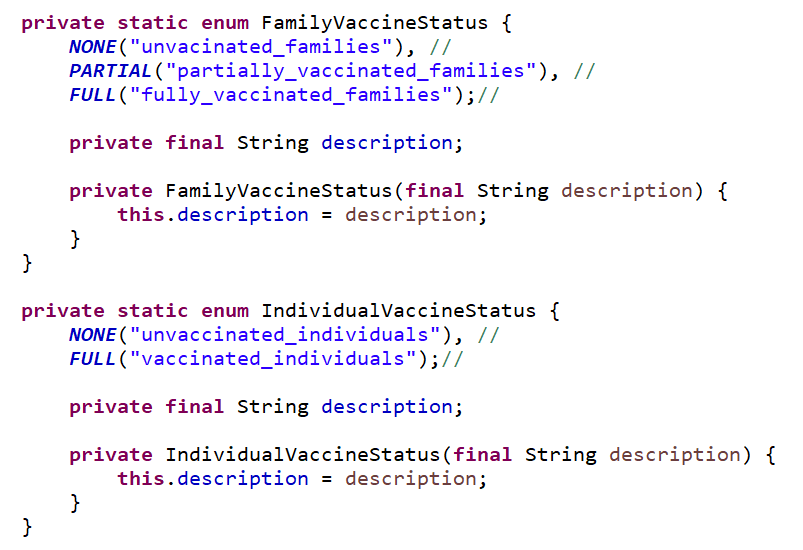
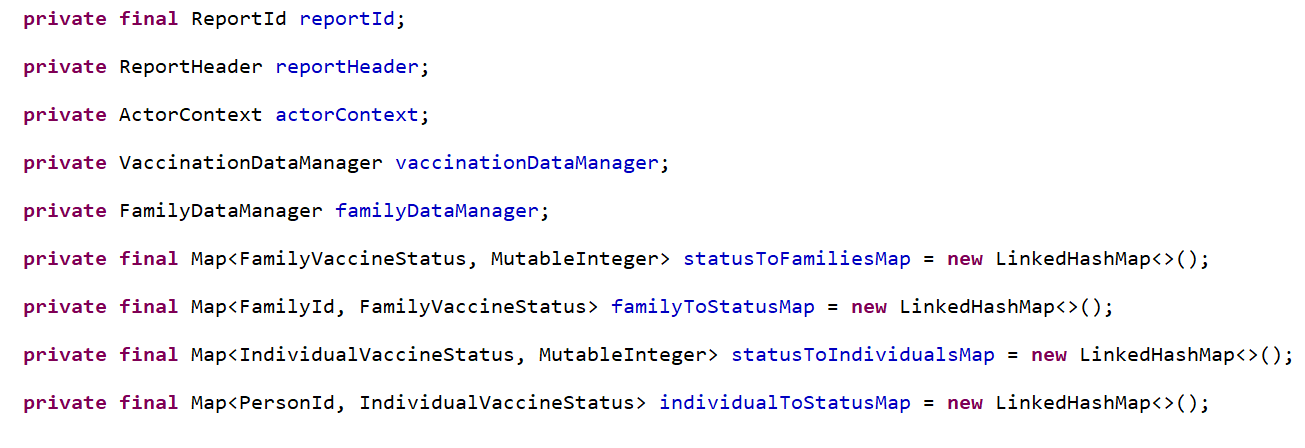
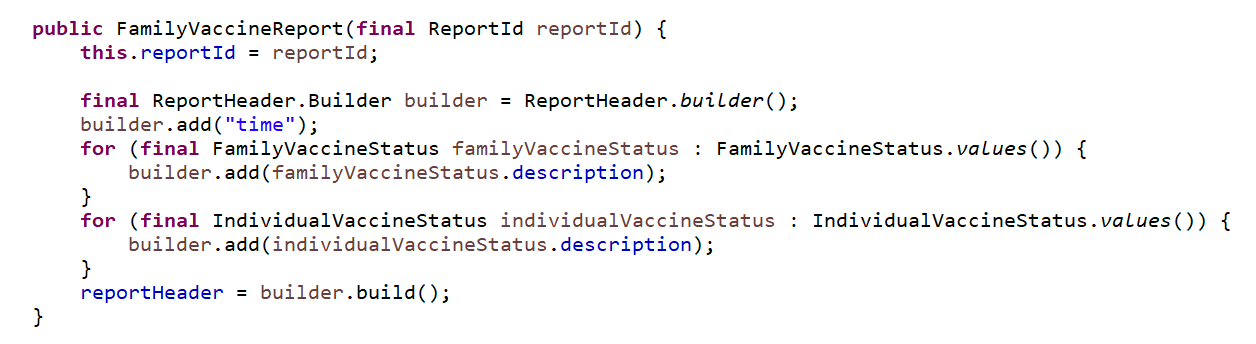


Figure 4.7 show the remaining private fields.

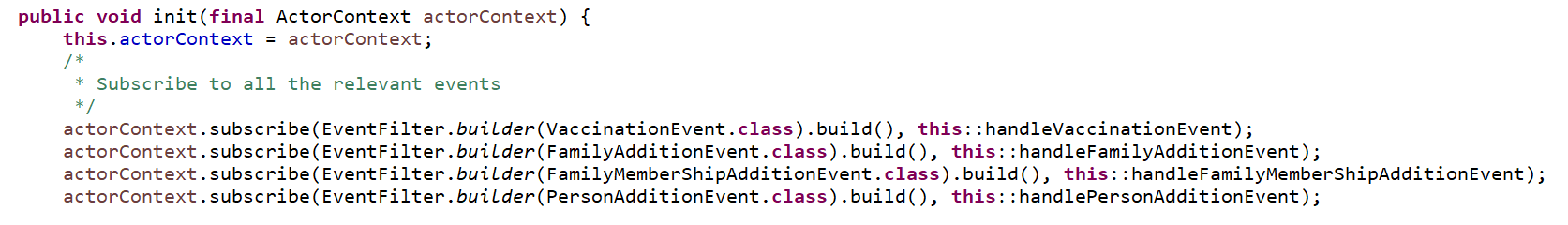
* report id – remains fixed from construction and is used to mark every report item
* reportHeader – is constructed once and used in the construction of every report item
* actorContext – a convenience reference kept by the actor to retrieve the simulation time
* vaccinationDataManager – a convenience reference to retrieve the vaccination status of each person
* familyDataManager – a convenience reference to retrieve the family members associated with a given person who has just been vaccinated
* statusToFamiliesMap – a map from family vaccine status to a mutable counter
* familyToStatusMap – a map for recording the current family vaccine status for each family
* statusToIndividualMap – a map from individual vaccine status to a mutable counter
* individualToStatusMap – a map for recording the current individual vaccine status for each person not assigned to a family

***Figure 4.7***

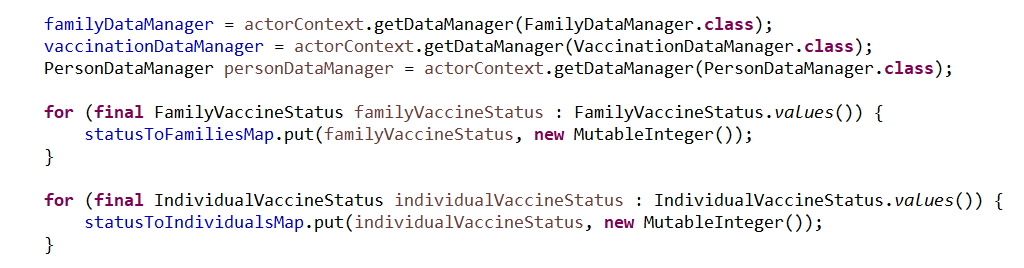
The report actor’s methods start with its constructor in figure 4.8. The report id is recorded and the report header field is built from the support enumerations.

***Figure 4.8***

Next is the initialization method that was used as the Consumer<ActorContext> that was passed to the simulation. This is invoked by the simulation just once at time zero and gives the report a chance to register for events and to initialize the private fields from figure 4.7. Figure 4.9 shows the report recording the actor context and subscribing to the four events of interest. These subscriptions reference local private method that will be described later.

***Figure 4.9***

In figure 4.10 we continue with the retrieval of the person, family and vaccination data managers. The maps containing the counts are initialized to zero.

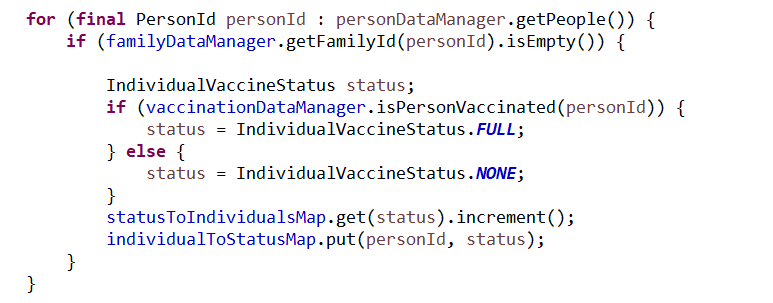
***Figure 4.10***

Figures 4.11 and 4.12 use the data managers to fill the count structures with the current state of the population.

***Figure 4.11***



***Figure 4.12***



Initialization finishes with the release of a single report item that summarizes the state of family vaccination at time zero.

***Figure 4.13***



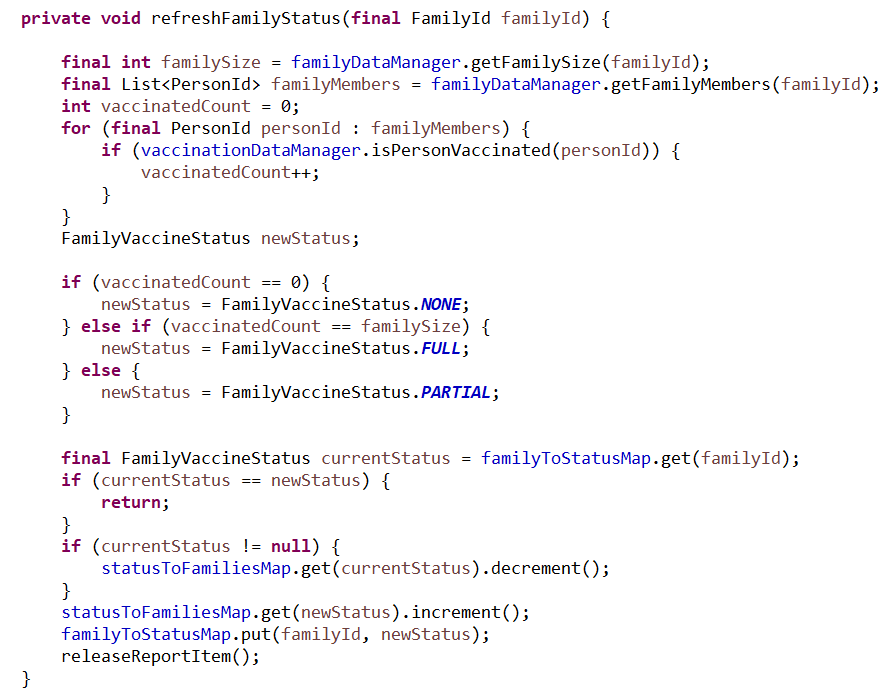
The methods for handling each event are shown in figure 4.14. All four methods select some relevant family id or person id and process changes to the counting data structure using the refreshFamilyStatus() and refreshInidividual() methods. The accounting for reports that are synthesizing multiple events can be somewhat tricky. No assumptions are made as to how and people are created, vaccinated and added to families so that changes to those processes in future versions of the model do not cause errors in the report.

***Figure 4.14***

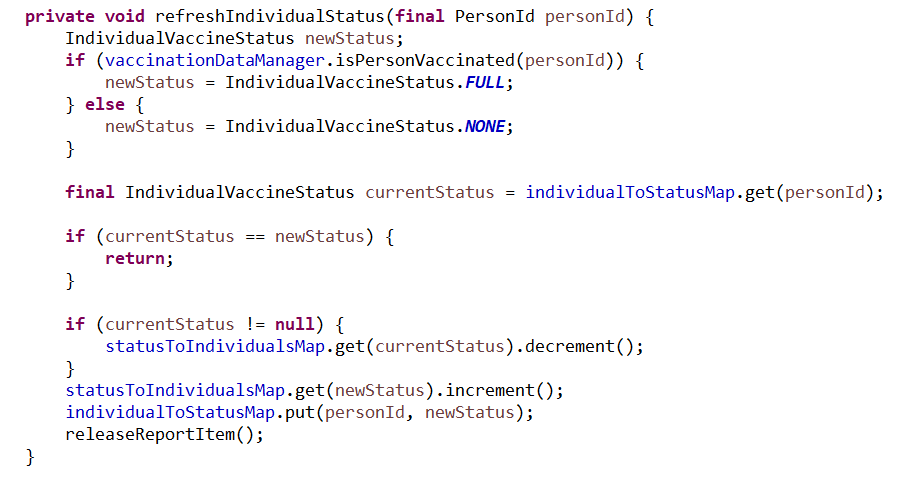


The refresh methods in figures 4.15 and 4.16 compare the current vaccination state of the families and individuals against the corresponding states tracked in the counting maps. If a change in the counts has occurred the counts are corrected and a new report item is released.

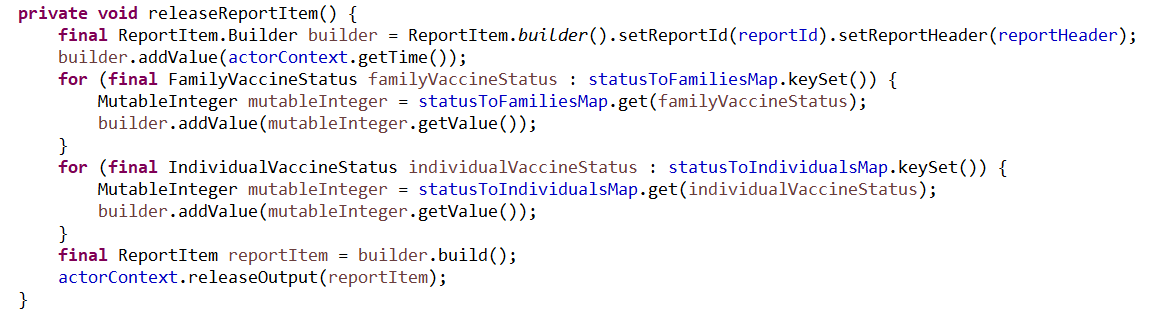
***Figure 4.15***



***Figure 4.16***

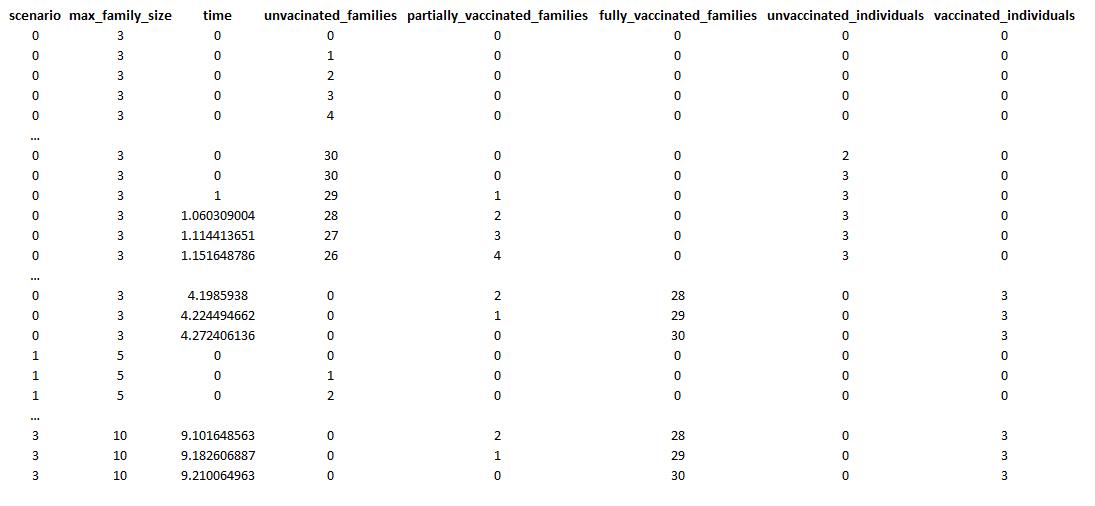


Releasing the report items that summarizes the family vaccination counts requires building a new report item with the fixed report id and report header values determined in the constructor. We then go on to add the time and count values in the order dictated by the helper enumerations so that they follow the header values established in the report header. Once the report item is complete it is released as output via the actor context. The simulation will in turn release the report item to the experiment where it will be distributed to the NIOReportItemHandler and then on the specific file manager(s) that record the items.

***Figure 4.17***

The resulting output in figure 4.18 contains the four scenarios showing the buildup of the population with all families and individuals being unvaccinated. Over time the number of vaccinated families increase and each simulation ends when all people have been vaccinated. The increase of max family size over the experiment causes there to be more people and thus the number of days to reach full vaccination also increases as expected.

***Figure 4.18***



## Periodic Reports

Producing a new report item each time a relevant event changes the internal tracking variable of a report actor will often produce too much output. An alternative is to periodically release one or more report items, usually on an hourly or daily basis. The reports plugin defines an abstract report actor class, the PeriodicReport, that manages the periodic flushing of the state of the report actor. This allows the descendant report actor class to concentrate on responding to the events while leaving the periodic production of report items to the base class.

The PeriodicReport provides several protected methods:

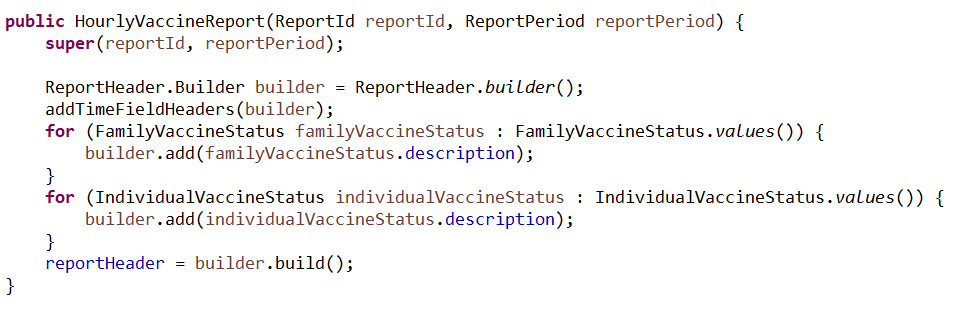
* two methods for filling in the time-based fields for the report header and report items
* three methods for subscribing to events that mirror the subscription methods of the actor context
* a method for retrieving the report id
* an abstract method for flushing the content of the report actor that must be implemented by the descendant report actor class

In addition, the PeriodicReport introduces a constructor that requires both a report id and a reporting period. If the constructor is overridden, the super() constructor must be invoked. The init() method is a consumer of actor context and if overridden, must also have the super() method invoked. The key takeaway is that by using the subscription methods supplied instead of the those on the actor context, the PeriodicReport is able to detect when time has reached the next reporting period and can force a flush of the current state of the descendant report **before** the next event is processed.

Our next example report actor class is the HourlyVaccineReport. It produces the same output as the FamilyVaccineReport, but does so on an hourly basis. This will cause the output of a report item every hour whether there are no stimulating events or many. The implementation of this report is nearly identical to the previous report and we will concentrate on highlighting the differences between the two approaches.

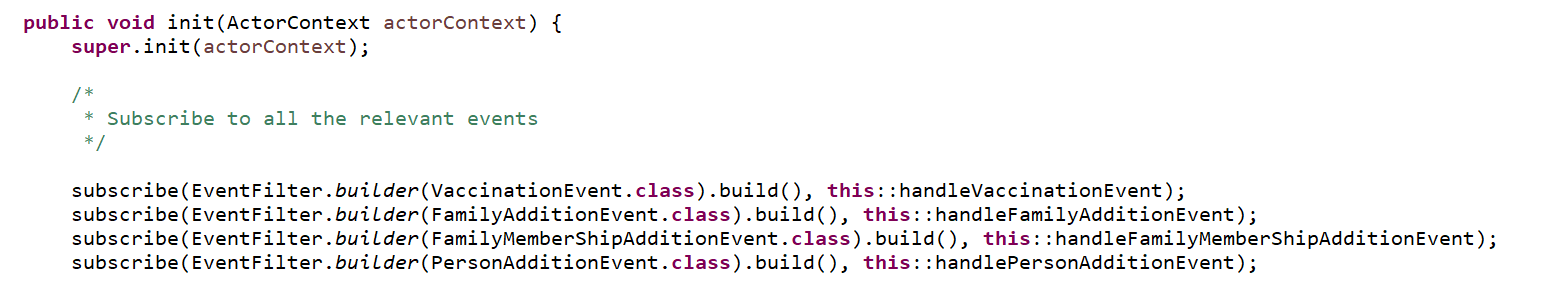
In figure 4.19 we see that the constructor invokes the super constructor. The construction of the report header is aided by the protected method addTimeFieldHeaders() which should be invoked as the first inputs to the report header builder. Note as well that we do not store the report id locally.

***Figure 4.19***



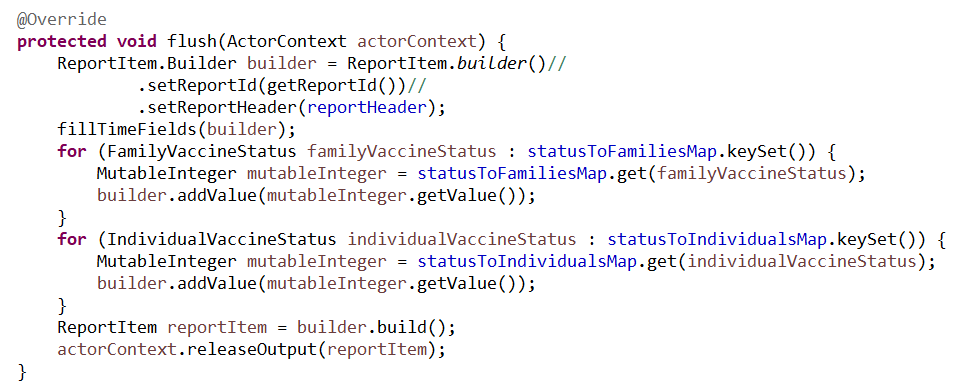
The init() method is nearly identical to the previous report. The only differences are:

* the invocation of the super constructor
* the subscription to events through the parent class rather than the actor context that allows the report to force report item flushing in the proper order

***Figure 4.20*** 

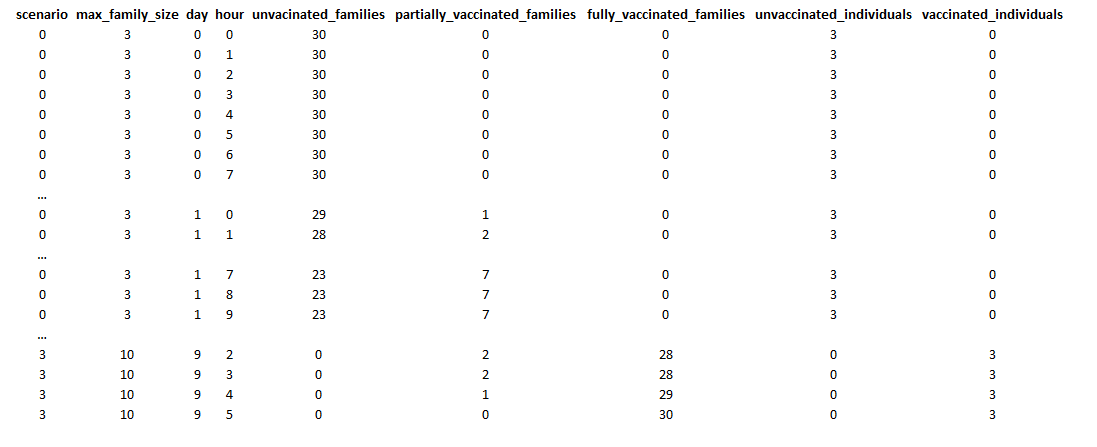
The releaseReportItem() method is now replaced by the flush() method override in figure 4.21.

***Figure 4.21***



The corresponding invocations of the releaseReportItem() that would have generated a new report item each time an event changed the internal counting variables are dropped. The flush() method will be invoked each time the parent report class determines that the planned next period has occurred. Note also that the time fields of the report item are filled by invoking the fillTimeFields() method which will add the correct time value for the period being reported rather than the current time. Otherwise, the implementations are identical.

The resulting output in figure 4.22 contains the four scenarios showing the buildup of the population with all families and individuals being unvaccinated. It shows the same overall pattern as the previous report, but treats the reporting of time in integer days and hours. Note that some of the output values repeat over the days and hours since there were no vaccinations during those periods.

***Figure 4.22***

Our final example, the StatelessVaccineReport in figure 4.23, continues from the HourlyVaccineReport but eschews the stateful counting mechanisms. Like the previous report, it is a periodic report actor but it does not store any state and does not subscribe to any events. Instead, it simply derives the report item on each flush() invocation.

***Figure 4.23***

This approach may seem wasteful since there is the potential for a great deal of recalculation, but since this is done on a daily basis, it may be well worth the reduction in memory if the model was actively tracking millions of families.

# Chapter 5: Properties

Modelers often need to associate properties with concepts found in plugins. For example, you may want to associate an integer number of times a person has been given a vaccine booster or have a double valued disease detection threshold that can be defined regionally. Most plugins will have a flexible, modeler-defined, set of properties that can be associated with people, groups or any other concept defined by the plugin. The core plugins included with GCM use a common property utility that introduces property Ids, property definitions and property values. It also provides several property value container classes to aid in efficient storage and retrieval.

### Property Identifiers

Property Ids are generally marker interfaces used to force unambiguous types in method signatures dealing with property related concepts. Each plugin that uses properties will introduce its own marker interface(s) and instances of the identifier are left to client (other plugins) to implement. This is often accomplished with enumerations.

### Property Definitions

Property definitions supply each plugin with:

* A class reference that defines the type of the property values
* A Boolean value indicating if property values are mutable
* A time tracking policy indicating whether a time value is stored each time a property value is assigned
* An optional default property value

The class reference dictates that type of all property values associated with the definition.

The mutability indicator controls whether property values can be set after the initial value is established. For example, consider the integer property “age” that is defined for people. Each person has a distinct integer age upon initial value assignment. If the property definition asserts that the property is not mutable, then the age value cannot be changed during the simulation’s execution. This is often used to fix global property values so that there is no chance that they can be reset by mistake.

It is often useful to know when a property was last assigned. The time tracking policy allow the definition to specify whether these time values are recorded. Some plugins will use this policy to avoid recording such time values where there would be tens of millions of entries and no use of these values by the modeler.

Default property values are used to spare the modeler from having to set property values when introducing new items to the simulation. For example, when adding a person to the simulation it might be useful to have a default of false for the property of “vaccinated”. However, for some properties there may be no meaningful default value. For example, consider the “age” property for a person. What would constitute a good default value? For this reason, supplying a default value as part of the property definition is optional.

### Concurrency Requirements

Property ids, property definitions and property values must be thread safe since they are shared across multiple scenarios (different simulation instances). It is usually best practice if they are implemented as immutable classes.

* Property ids are usually marker interfaces and are often implemented by static enumerations and are thus generally threadsafe
* The PropertyDefinition class is provided by the utility and is threadsafe subject to the thread safety of its default value
* Property values are often boxed primitives and are generally threadsafe. In general, mutation of a property value in GCM does not mean that the property value is mutated. Rather, it usually means that a new immutable value is now associated with the property id.

### Immutability

For a class to be immutable in Java it must meet three requirements

1. Its internal fields must not be mutated. There can be no setter methods or any other mechanism that changes an assignment post construction
2. All fields are declared final
3. No reference to the immutable object may be passed during its construction

### Expected Behaviors of Plugins using properties

All implementations of property mechanisms in GCM are expected to meet the following requirements:

* Property values are never null
* Property definitions that do not supply a default value must be supported by other mechanisms that ensure that property values are never null
* Property instance values must always assignment compatible to their corresponding property definition’s property type reference

# Chapter 6: Global Properties Plugin

The global property plugin implements a flexible property system for properties that have global scope. Specifically, global properties have no association with a specific person, place or other instance-based concept.

### Plugin Data Initialization

The plugin is initialized using a GlobalPropertiesPluginData object that collects global property definitions and global property values. Even though the property definitions can contain default property values, the ability to set property values is included to add some flexibility to the collection process since the client model may separate definitions from values in its input files.

### Plugin Behavior

The plugin adds a single data manager to the simulation as an instance of the GlobalPropertiesDataManager that is initialized with the GlobalPropertiesPluginData.

### Data Manager

The data manager provides access to the global properties and provides the ability to:

* Define new global properties (not contained in the initial data)
* Retrieve global property definitions
* Retrieve global property ids
* Retrieve global property values and the times when they were set
* Set global property values

The data manager also produces observable events when a new global property is defined or when a global property value is assigned. The plugin provides the GlobalPropertyReport that subscribes to these events and produces a trace report of property value assignments.

### Example Code

Example\_13.java shows a simple usage of the global properties plugin. In it we will add three double valued properties: ALPHA, BETA, and GAMMA. ALPHA and BETA will be used to vary the scenarios in the experiment and DELTA will be set to a simple function of ALPHA and BETA that will change over time in the simulation. This will culminate in a report that shows each time the global variables are defined or their values are set.

The example includes three plugins:

* Global properties plugin – (GCM core plugin) used to manage the properties
* Reports plugin – (GCM core plugin) used to manage the report
* Model plugin – (local plugin) used to introduce a single actor that will alter the value of DELTA over time

The example’s main method in figure 6.1 adds the three plugins:

* Global properties plugin
  + initialized with the three global properties
* Reports plugin
  + adds the GlobalPropertyReport (defined by the Global properties plugin)
  + uses the NIOReportItemHandler to associate an output file with the report
* Model plugin
  + Uses no inputs, but will add a single instance of the GammaActor class

The main method then forms a dimension for the experiment from variant values of ALPHA and BETA. Finally, it executes the experiment.

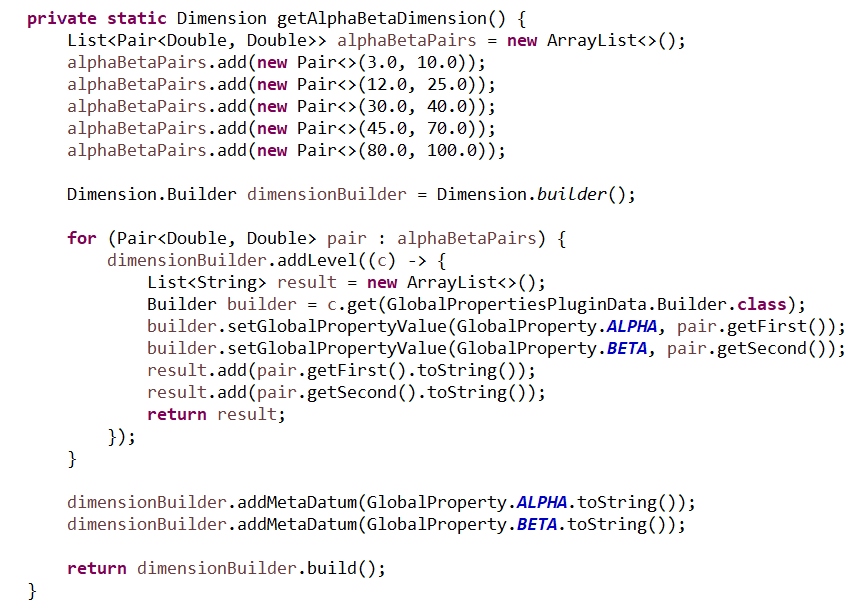
***Figure 6.1***

Initialization of the global properties is shown in figure 6.2

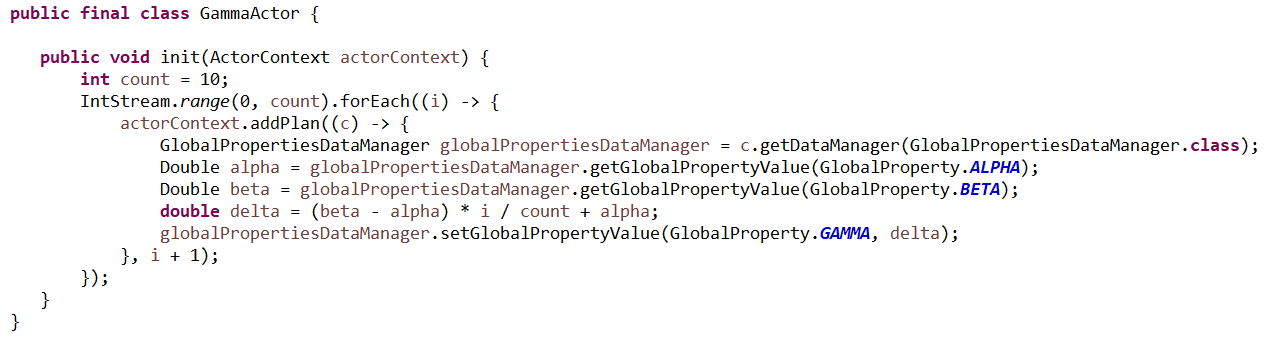
***Figure 6.2***

Figure 6.3 shows the construction of the experiment’s single dimension.

***Figure 6.3***

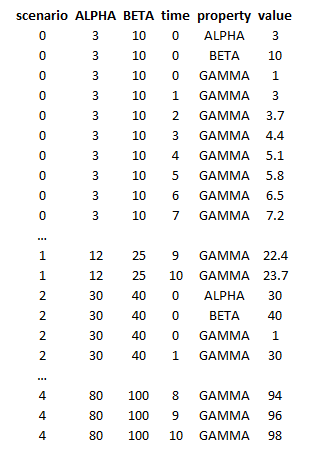


The GammaActor class in figure 6.4 schedules 10 plans, set one day apart, to change the DELTA value as a successive interpolation between the ALPHA and BETA values that are in turn controlled by the experiment.

***Figure 6.4***

The resultant global properties report shows the correct interpolated values for the five scenarios in figure 6.5.

***Figure 6.5***



# Chapter 7: People Plugin

Asdf awfsdf

# Chapter 8: Person Properties Plugin

asdf

# Chapter 9: Groups Plugin

asdf

# Chapter 10: Regions Plugin

asdf

# Chapter 11: Resources Plugin

asdf

# Chapter 12: Materials Plugin

asdf

# Chapter 13: Partitions Plugin

Asdf

