GCM 3.0 Modeling Guide

# Chapter 1: Overview

The General Compartmental Model (GCM) is a Java based simulation framework for building disease progression models. Compartments represent the disease progression state for a population. A modeler/developer creates compartment logic which defines the behavior of the simulation while GCM provides an event engine, reporting and other services that are common across models.

**Recommended Pre-requisites**

* Familiarity with Java and Object Oriented Programming
  + Immutability
  + Interface-based design
* Familiarity with modeling
  + Event based systems
  + Agent based systems
* Access to the GCM jar artifacts and knowledge of how to use them
* Access to an IDE

**Model Features**

* Regions – Different physical regions can have unique behavior
* People – Typically handles millions of people
* Compartments – Provided by the modeler and executes the progression of people through disease state.
* Resources – Data driven and customizable representation of medical and other resources used in patient treatment
* Materials -- Data driven and customizable representation of the production process for the creation of vaccines and other resources
* Groups – Convenient grouping of the population into families, schools and other social organizations
* Population Indexing – Rapid selection of people on the basis of complex person property logic
* Experiment – Scenarios and replications are easily managed and reported
* Multi-threaded – Management of concurrency issues and full use of machine resources
* Reports – Wide range of pre-defined reports that cover all simulation activities
* Custom Output – Customizable reports and other output
* State management – All data state of people, groups, indices, compartments and other simulation entities are fully managed
* Planning – Future event system with cancellation
* Meta-Analysis tools – Give insight into design considerations for managing memory and runtime trade space
* Memory Management – Utilizes various memory improvement techniques that are transparent to the modeler

# Chapter 2: Getting Started with GCM

## **Problem**

In this example we will build a simple model that moves people selectively over time from one compartment to another. People will have two properties:

1. IMMUNE: A Boolean value. People who are immune won’t get sick.
2. SICK: A Boolean value that starts out FALSE for everyone and switches to TRUE for 80% of those who are not immune.

The two compartments will be:

1. UNTREATED: People will be initialized at the start of the model as UNTREATED. Once they SICK a person will move to the TREATED compartment at random using a uniform distribution ranging from 1 to 11 days.
2. TREATED: When SICK people arrive in the TREATED compartment, they will receive treatment based on how long they have been ill.

A resource name MEDICATION will be added to treat the people in the TREATED compartment

## **Implementation**

Writing a GCM model can take a small amount of Java code. In this example we will show the three main activities when creating a model:

* Building the experiment
* Executing the experiment
* Designing the components used in the experiment

In this example we will be working mostly in a simple class with a main method.

**public** **class** GettingStartedRunner {

**public** **static** **void** main(String[] args) {}

}

We will also show some basic Components where most of the modeler’s Java code is developed.

**public** **class** TreatmentCompartment **extends** AbstractComponent {

@Override

**public** **void** init(Environment environment) {

}

}

## **Building the Experiment**

1. Create an experiment builder – We add items to this builder and demonstrate how it builds the experiment for us.
2. Define the IMMUNE and SICK properties of people
3. Add a single region to contain the people
4. Add the TREATED and UNTREATED compartments
5. Add the MEDICATION resource and supply the single region with 300 doses
6. Add 100 people to the experiment
7. Build the experiment

**public** **class** GettingStartedRunner {

**public** **static** **void** main(String[] args) {

// Create the experiment builder

ExperimentBuilder experimentBuilder = **new** ExperimentBuilder();

// Define person property - immune

PropertyDefinition propertyDefinition = **new** PropertyDefinition(Boolean.**class**, **false**, **true**);

experimentBuilder.definePersonProperty(PersonProperty.***IMMUNE***, propertyDefinition);

// Define person property - sick

propertyDefinition = **new** PropertyDefinition(Boolean.**class**, **false**, **true**, TimeTrackingPolicy.***TRACK\_TIME***);

experimentBuilder.definePersonProperty(PersonProperty.***SICK***, propertyDefinition);

// Add a single region

experimentBuilder.addRegionId(Region.***REGION\_1***, SimpleRegion.**class**);

/\*

\* Add the exposed, treatment, recovered and terminal compartments. Note

\* that we identify the compartment with an ID and then pass only a

\* class reference, not an instance of the compartment class. More on

\* that in later lessons.

\*/

experimentBuilder.addCompartmentId(Compartment.***TREATMENT***, TreatmentCompartment.**class**);

experimentBuilder.addCompartmentId(Compartment.***EXPOSED***, ExposedCompartment.**class**);

experimentBuilder.addCompartmentId(Compartment.***TERMINAL***, TerminalCompartment.**class**);

experimentBuilder.addCompartmentId(Compartment.***RECOVERED***, RecoveredCompartment.**class**);

// Add a resource to the model and give REGION\_1 300 doses of medication

experimentBuilder.addResource(Resource.***MEDICATION***);

experimentBuilder.addRegionResourceLevel(Region.***REGION\_1***, Resource.***MEDICATION***, 300);

// Add a few people

**for** (**int** i = 0; i < 100; i++) {

PersonId personId = **new** PersonId(i);

experimentBuilder.addPerson(personId, Region.***REGION\_1***, Compartment.***UNTREATED***);

}

// Build the experiment

Experiment experiment = experimentBuilder.buildAll();

}

}

## **Executing the Experiment**

We continue by creating an experiment executor that will manage our experiment and execute the various scenarios and replications needed. In our current example we have created an experiment that has exactly one scenario because we have not added alternate values for any of the properties we have defined.

The experiment executor can accept many settings, but we will do the minimum to get our experiment to run and get some output.

**public** **class** GettingStartedRunner {

**public** **static** **void** main(String[] args) {

// Continued from Code from above…

// Prepare the experiment executor

ExperimentExecutor experimentExecutor = **new** ExperimentExecutor();

experimentExecutor.setExperiment(experiment);

// Add reports that will show people moving from one compartment to another and show the distribution of MEDICATION

experimentExecutor.addCompartmentTransferReport(somePath, ReportPeriod.***DAILY***);

experimentExecutor.addPersonResourceReport(somePath, ReportPeriod.***DAILY***, **false**, **false**, Resource.***MEDICATION***);

experimentExecutor.addResourceReport(somePath, ReportPeriod.***DAILY***, Resource.***MEDICATION***);

// Execute the experiment

experimentExecutor.execute();

}

}

## **Designing the Components**

People start in the EXPOSED compartment. The EXPOSED compartment uses its init() method to initialize its activities. In this case, 20% of the people are declared immune and 80% of the remaining people get sick immediately. Those who are sick are sent on to the TREATMENT compartment at random times over the course of 1 to 10 days to seek treatment. People enter the TREATMENT compartment over time. The compartment uses its init() method to tell the simulation that it wants to be alerted when people arrive. It attempts to treat the sick people by giving them a medication dosage equal to the number of days that each person has been sick. However, it is constrained by a limited supply of the medication. When there is no medication left, the person is sent to the TERMINAL compartment.

The Seek Treatment Plan class will be used by the EXPOSED compartment to schedule a person to seek treatment in the future.

**public** **class** SeekTreatmentPlan **implements** Plan {

**private** **final** PersonId personId;

**public** SeekTreatmentPlan(**final** PersonId personId) {

**this**.personId = personId;

}

**public** PersonId getPersonId() {

**return** personId;

}

}

The EXPOSED compartment inspects each person and determines their immune status and whether they are sick. Sick people are scheduled to seek treatment in 1 to 10 days.

**public** **class** ExposedCompartment **extends** AbstractComponent {

@Override

**public** **void** init(Environment environment) {

// Get the random generator that is managed by the simulation

RandomGenerator randomGenerator = environment.getRandomGenerator();

/\*

\* Retrieve a list of people that are in this compartment. We assume

\* that this is the entire population of the model.

\*/

List<PersonId> people = environment.getPeopleInCompartment(Compartment.***EXPOSED***);

/\*

\* Checking our assumption

\*/

**if** (environment.getPopulationCount() != people.size()) {

**throw** **new** RuntimeException("The entire population should start in the EXPOSED compartment");

}

/\* Determine immunity status for every person. For each person, there is

\* a 20% chance to be immune. Recall that immunity was defaulted to

\* false, so we only set the true values.

\*/

**for** (PersonId personId : people) {

**boolean** immune = randomGenerator.nextDouble() < 0.2;

**if** (immune) {

environment.setPersonPropertyValue(personId, PersonProperty.***IMMUNE***, immune);

}

}

// Retrieve a list of people who are not immune

List<PersonId> nonImmunePeople = environment.getPeopleWithPropertyValue(PersonProperty.***IMMUNE***, **false**);

/\*

\* 80% of those who are not immune will get sick. Again the default was

\* false so we only set the true values.

\*/

**for** (PersonId personId : nonImmunePeople) {

**boolean** sick = randomGenerator.nextDouble() < 0.8;

**if** (sick) {

environment.setPersonPropertyValue(personId, PersonProperty.***SICK***, sick);

}

}

// Retrieve the list of the people who are sick

List<PersonId> sickPeople = environment.getPeopleWithPropertyValue(PersonProperty.***SICK***, **true**);

/\*

\* Each person who is sick will eventually seek treatment (1-10) days

\* from now. Make a plan for each person to seek treatment in the

\* future.

\*/

**for** (PersonId personId : sickPeople) {

// Make a future plan to move the person to TREATMENT compartment

SeekTreatmentPlan seekTreatmentPlan = **new** SeekTreatmentPlan(personId);

// Determine when the person should move

**double** treatmentDelayDays = 9 \* randomGenerator.nextDouble() + 1;

**double** treatmentEntryTime = environment.getTime() + treatmentDelayDays;

environment.addPlan(seekTreatmentPlan, treatmentEntryTime);

}

}

@Override

**public** **void** executePlan(**final** Environment environment, **final** Plan plan) {

// The plan made earlier has come back from the environment at the

// scheduled time. We now act on that plan by

// moving the person to the TREATMENT compartment

SeekTreatmentPlan seekTreatmentPlan = (SeekTreatmentPlan) plan;

environment.setPersonCompartment(seekTreatmentPlan.getPersonId(),

Compartment.***TREATMENT***);

}

}

The treatment compartment receives sick people who are seeking treatment. Treatment consists of administering a dose of MEDICATION sized to the number of days each person has been sick. The MEDICATION is however limited to the supply initialized in the person’s region. Treated people are sent to the RECOVERED compartment and untreated (or insufficiently treated) people are sent to the TERMINAL compartment.

**public** **class** TreatmentCompartment **extends** AbstractComponent {

@Override

**public** **void** init(Environment environment) {

// tell the simulation to alert us when a person arrives in this

// compartment

environment.observeCompartmentPersonArrival(**true**, Compartment.***TREATMENT***);

}

@Override

**public** **void** observeCompartmentPersonArrival(**final** Environment environment, **final** PersonId personId) {

/\*

\* Every person who shows up for treatment should be sick, but let’s

\* check to make sure nothing went wrong.

\*/

Boolean personIsSick = environment.getPersonPropertyValue(personId, PersonProperty.***SICK***);

**if** (!personIsSick) {

**throw** **new** RuntimeException("Unexpected non-sick person " + personId);

}

/\*

\* Determine how long the person has been sick

\*/

**double** timeWhenSick = environment.getPersonPropertyTime(personId, PersonProperty.***SICK***);

**double** currentTime = environment.getTime();

**double** sicknessDuration = currentTime - timeWhenSick;

/\*

\* Determine the dosage needed -- one dose per day sick

\*/

**long** requiredDosage = (**long**) sicknessDuration + 1;

/\*

\* Determine how many doses of MEDICATION are still on hand in the

\* person's region

\*/

RegionId regionId = environment.getPersonRegion(personId);

**long** availableDosage = environment.getRegionResourceLevel(regionId, Resource.***MEDICATION***);

/\*

\* Determine the dosage that will be given to the person -- don't exceed

\* the amount available

\*/

**long** distributedDosage = Math.*min*(requiredDosage, availableDosage);

**if** (distributedDosage >= requiredDosage) {

/\*

\* Give the person the treatment -- this will automatically move \* the amount from the region's supply to the person. Then send

\* the person to the RECOVERED compartment

\*/

environment.transferResourceToPerson(Resource.***MEDICATION***, personId, distributedDosage);

environment.setPersonPropertyValue(personId, PersonProperty.***SICK***, **false**);

environment.setPersonCompartment(personId, Compartment.***RECOVERED***);

} **else** {

/\*

\* We have run out of MEDICATION and the person is sent to the

\* TERMINAL compartment

\*/

environment.setPersonCompartment(personId, Compartment.***TERMINAL***);

}

}

}

The remaining compartments act only as the final compartments for sick people. They take no action.

**public** **class** RecoveredCompartment **extends** AbstractComponent {

@Override

**public** **void** init(Environment environment) {

//no action required

}

}

**public** **class** TerminalCompartment **extends** AbstractComponent {

@Override

**public** **void** init(Environment environment) {

//no action required

}

}

## **Examining the Results**

The compartment transfer report shows people moving from UNTREATED to TREATED over days 0 through 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scenario | Replication | Day | Region | Source Compartment | Destination Compartment | Transfers |
| 1 | 1 | 0 | REGION\_1 | EXPOSED | EXPOSED | 100 |
| 1 | 1 | 1 | REGION\_1 | TREATMENT | RECOVERED | 4 |
| 1 | 1 | 1 | REGION\_1 | EXPOSED | TREATMENT | 4 |
| 1 | 1 | 2 | REGION\_1 | TREATMENT | RECOVERED | 9 |
| 1 | 1 | 2 | REGION\_1 | EXPOSED | TREATMENT | 9 |
| 1 | 1 | 3 | REGION\_1 | TREATMENT | RECOVERED | 7 |
| 1 | 1 | 3 | REGION\_1 | EXPOSED | TREATMENT | 7 |
| 1 | 1 | 4 | REGION\_1 | TREATMENT | RECOVERED | 12 |
| 1 | 1 | 4 | REGION\_1 | EXPOSED | TREATMENT | 12 |
| 1 | 1 | 5 | REGION\_1 | TREATMENT | RECOVERED | 5 |
| 1 | 1 | 5 | REGION\_1 | EXPOSED | TREATMENT | 5 |
| 1 | 1 | 6 | REGION\_1 | TREATMENT | RECOVERED | 8 |
| 1 | 1 | 6 | REGION\_1 | EXPOSED | TREATMENT | 8 |
| 1 | 1 | 7 | REGION\_1 | TREATMENT | RECOVERED | 9 |
| 1 | 1 | 7 | REGION\_1 | EXPOSED | TREATMENT | 9 |
| 1 | 1 | 8 | REGION\_1 | TREATMENT | TERMINAL | 7 |
| 1 | 1 | 8 | REGION\_1 | TREATMENT | RECOVERED | 2 |
| 1 | 1 | 8 | REGION\_1 | EXPOSED | TREATMENT | 9 |
| 1 | 1 | 9 | REGION\_1 | TREATMENT | TERMINAL | 6 |
| 1 | 1 | 9 | REGION\_1 | EXPOSED | TREATMENT | 6 |

The person resource report shows how many people have cumulatively received medication. Note that days 8 and 9 indicate that the supply of MEDICATION had run out.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scenario | Replication | Day | Region | Compartment | Resource | People With Resource |
| 1 | 1 | 1 | REGION\_1 | RECOVERED | MEDICATION | 4 |
| 1 | 1 | 2 | REGION\_1 | RECOVERED | MEDICATION | 13 |
| 1 | 1 | 3 | REGION\_1 | RECOVERED | MEDICATION | 20 |
| 1 | 1 | 4 | REGION\_1 | RECOVERED | MEDICATION | 32 |
| 1 | 1 | 5 | REGION\_1 | RECOVERED | MEDICATION | 37 |
| 1 | 1 | 6 | REGION\_1 | RECOVERED | MEDICATION | 45 |
| 1 | 1 | 7 | REGION\_1 | RECOVERED | MEDICATION | 54 |
| 1 | 1 | 8 | REGION\_1 | RECOVERED | MEDICATION | 56 |
| 1 | 1 | 9 | REGION\_1 | RECOVERED | MEDICATION | 56 |

The resource report shows the MEDICATION being distributed initially to REGION\_1 and then to people over the course of 8 days.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Scenario | Replication | Day | Region | Compartment | Resource | Activity | Actions | Items |
| 1 | 1 | 0 | REGION\_1 |  | MEDICATION | Region Resource Addition | 1 | 300 |
| 1 | 1 | 1 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 4 | 8 |
| 1 | 1 | 1 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 4 | 8 |
| 1 | 1 | 1 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 4 | 8 |
| 1 | 1 | 2 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 9 | 27 |
| 1 | 1 | 2 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 9 | 27 |
| 1 | 1 | 2 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 9 | 27 |
| 1 | 1 | 3 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 7 | 28 |
| 1 | 1 | 3 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 7 | 28 |
| 1 | 1 | 3 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 7 | 28 |
| 1 | 1 | 4 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 12 | 60 |
| 1 | 1 | 4 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 12 | 60 |
| 1 | 1 | 4 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 12 | 60 |
| 1 | 1 | 5 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 5 | 30 |
| 1 | 1 | 5 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 5 | 30 |
| 1 | 1 | 5 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 5 | 30 |
| 1 | 1 | 6 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 8 | 56 |
| 1 | 1 | 6 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 8 | 56 |
| 1 | 1 | 6 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 8 | 56 |
| 1 | 1 | 7 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 9 | 72 |
| 1 | 1 | 7 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 9 | 72 |
| 1 | 1 | 7 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 9 | 72 |
| 1 | 1 | 8 | REGION\_1 | TREATMENT | MEDICATION | Person Compartment Departure | 2 | 18 |
| 1 | 1 | 8 | REGION\_1 | TREATMENT | MEDICATION | Resource Transfer To Person | 2 | 18 |
| 1 | 1 | 8 | REGION\_1 | RECOVERED | MEDICATION | Person Compartment Arrival | 2 | 18 |

# Chapter 3: Experiments

Designing an experiment is the primary means for constructing scenarios to run in GCM. Some definitions first:

***Experiment*** *– A hypercube that is formed from a set of Scenario parameterizations.*

***Scenario*** *- Describes a point in the Experiment hypercube and contains a specific parameter set. The scenario contains the sum of data describing the initial state of a GCM simulation, except for the initialization of random number generation.*

***Replication*** *– A scenario instance mapped to a specific random seed value to explore the stochastic nature of a given Scenario. Replications in GCM are derived from the random number generation seed value for a GCM experiment.*

The ExperimentBuilder class provides the means for collecting the data that will result in the construction of one to many scenarios. Most of the data collected will apply to every scenario that is created. Only property values and resource allocations may vary from one scenario to another. The ExperimentBuilder has dozens of data entry methods and is insensitive to the order of their invocation. For example, you can specify property values *before* defining the property.

**Invariant data**: All scenarios in an experiment share the following data:

* Components and their identifiers
* Property definitions
* People
* Group Types
* Groups
* Person membership in groups
* Resources
* Materials
* Batches
* Stages
* Batch membership in Stages
* Time tracking policies
* Map Options

**Variant data**: Scenarios can vary in an experiment over the following data:

* All property values
* All resource allocations

The methods that collect varying values for properties and resources naturally lead to a regular space. For example, suppose that 3 values were collected for a global property called ALPHA and 10 values were collected for a resource property called BETA. There are 3x10=30 possible (ALPHA, BETA) combinations for these values and thus there would be 30 scenarios defined by the data. By adding multiple values for various properties it is easy to create a large experiment space containing thousands of scenarios.

The ExperimentBuilder also allows the modeler to specify covariant properties. For example, suppose that there are 5 values added for ALPHA and 5 values added for BETA. Since the two properties have the same number of variants it is possible to link them dependently. So, instead of generating 5x5=25 scenarios, only 5 scenarios are generated. This pattern may be applied to any number of variables and in any number of combinations.

*Examples of specifying* ***invariant*** *data: each is defined/set at most once*

experimentBuilder.definePersonProperty(PersonProperty.***IMMUNE***, **new** PropertyDefinition(Boolean.**class**, **false**, **true**));

experimentBuilder.setCompartmentMapOption(MapOption.***ARRAY***);

experimentBuilder.defineGlobalProperty(GlobalProperty.R0, **new** PropertyDefinition(Double.**class**, 1.3) );

*Examples of specifying* ***variant*** *data: each may be set multiple times to different values*

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 0);

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 10);

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 20);

*Examples of specifying* ***covariant*** *data:*

We add three values for person 456’s level of ASPIRIN

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 0);

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 10);

experimentBuilder.addPersonResourceLevel(new PersonId(456), Resource.***ASPIRIN***, 20);

We add three values for Region 7’s level of BANDAGE

experimentBuilder.addRegionResourceLevel(Region.REGION\_7, Resource.***BANDAGE***, 0);

experimentBuilder.addRegionResourceLevel(Region.REGION\_7, Resource.***BANDAGE***, 15000);

experimentBuilder.addRegionResourceLevel(Region.REGION\_7, Resource.***BANDAGE***, 100000);

We use the string “key” – but it could be any object – as a tag marking each as be joined covariantly.

experimentBuilder.covaryPersonResource(new PersonId(456), Resource.***ASPIRIN***, “key”);

experimentBuilder.covaryRegionResource(Region.REGION\_7, Resource.***BANDAGE***, “key”);

Once all the data has been added to the ExperimentBuilder, we build the Experiment:

Experiment experiment = experimentBuilder.build();

The scenarios are contained in the derived experiment and will be passed to an ExperimentExecutor to execute the experiment. Replications are specified at the ExperimentExecutor. The number of simulation runs will be equal to the product of the number of scenarios and the number of replications chosen.

## **Executing an Experiment**

The Experiment Executor class accepts an Experiment (containing the scenarios) and executes the experiment over some number of replications. It takes in additional parameters to guide this execution:

* Replication control
* Concurrency control
* Console progress
* Standard reports
* Custom reports
* Custom output
* Profiling reports
* Run resumption

*Start by adding the experiment:*

ExperimentExecutor experimentExecutor = **new** ExperimentExecutor();

experimentExecutor.setExperiment(experiment);

*Replication control*

Set an initial seed value for random number generation and set a replication count. The given seed value will be used to generate a new random seed for each replication. The simulation instances will use the random seed in the replications to start their random number generation. The default seed is 0 and the default replication count is 1.

experimentExecutor.setSeed(123456789L); experimentExecutor.setReplicationCount(100);

*Concurrency control:*

Set the number of threads outside of the main thread that the experiment executor will use to run the simulation instances concurrently. While a large value is tolerated, best practice is to set it to the known number of available cores or perhaps a bit less. Setting the value to zero will cause the experiment executor to run all simulation instances in the main thread. The default thread count is zero.

experimentExecutor.setThreadCount(8);

*Console progress:*

Control experiment progress reporting to standard out. The report will show the number of scenario/replication pairs that have completed execution and give an estimate of how long the remaining runs will take. The default for producing console output is false.

experimentExecutor.setConsoleOutput(true);

*Standard and Custom reports:*

There are 17 standard reports provided by GCM. See [Output and Reports](#_Chapter_13:_Output) for the details of each report. Although the specific arguments for each report can vary they are all similar to these examples:

*Standard reports:*

experimentExecutor.addCompartmentTransferReport(somePath, ReportPeriod.***DAILY***);

experimentExecutor.addResourceReport(somePath, ReportPeriod.***DAILY,Resource.BANDAGE, Resource.ASPIRIN***);

*Custom report:*

experimentExecutor.addCustomReport(somePath, Custom.class, data1, data2, …);

*Custom output:*

See [Output and Reports](#_Chapter_13:_Output) details:

experimentExecutor.setOutputItemHandler(someOutputItemhandler);

*Profiling reports:*

See [Profiling Reports](#_Profile_Report) for the details:

experimentExecutor.setProfileReport(somePath);

experimentExecutor.setPlanningQueueReport(somePath, 10000);

experimentExecutor.setMemoryReport(somePath, 10);

*Run resumption:*

See [Halting and Resume an Experiment](#_Halting_and_Resuming) for the details:

experimentExecutor.setOutputItemHandler(someOutputItemhandler);

## **Halting and Resuming an Experiment**

GCM allows the user to halt and resume a long running experiment. You do this by setting an experiment progress log (EPL) with the single command:

Path progressLogPath = Paths.get(“c:\\....\\ progressLog.txt”);

experimentExecutor.setExperimentProgressLog(progressLogPath);

GCM uses the EPL file to record the completed progress of scenario/replication pairs as it executes the experiment. Terminating GCM in any way (powering down, killing the process, etc.) will leave the EPL in a recoverable state. On GCM ‘s next execution with that EPL it will account for the progress that was previously made and continue the remaining scenario/replication pairs. Progress for the currently running replications will be discarded. GCM will progress an experiment through repeated halts and resumes.

Note that changes to the experiment between runs are not taken into account when resuming an experiment and some care must be given to not producing inconsistent output data. You can get a fresh start on the experiment by either removing the command above or deleting the EPL file.

# Chapter 4: Identifiers

GCM uses distinct identifier classes for categorizing and keying every major element of the simulation. Each type of data has its own identifier class. GCM provides these identifier classes, but often as interfaces only, so that the modeler must provide the specific identifier instances. For example, although GCM defines a RegionId class to identify regions, the modeler must create the region id instances when adding regions to the experiment.

**Identifiers fall into three categories**

1. Integer based identifiers
2. Marker interface identifiers
3. Internal identifiers

## **Integer Based Identifiers**

These six identifiers are essentially ints wrapped in a class.

1. ScenarioId
2. ReplicationId
3. BatchId
4. StageId
5. GroupId
6. PersonId

ScenarioId and ReplicationId instances are created by the experiment builder and experiment executor and use contiguous values. The remaining four are BatchId, StageId, GroupId and PersonId. These can be created by the modeler when specifying experiment data to the experiment builder. When the simulation loads the scenario containing these values it will often re-number them so that the values create a contiguous range starting with zero. This is done to make the simulation more efficient and does not alter any relationships established in the experiment. As the simulation progresses, additional elements such as batches, stages, groups and people may be created dynamically as well. To maintain its efficiency, GCM will issue these dynamically allocated identifiers when the relevant item is added to the simulation.

## **Marker Interface Identifiers**

The remaining identifiers cover the rest of the simulation:

* BatchPropertyId
* CompartmentPropertyId
* ComponentId
  + CompartmentId
  + GlobalComponentId
  + MaterialsProducerId
  + RegionId
* GlobalPropertyId
* GroupPropertyId
* GroupTypeId
* MaterialId
* MaterialsProducerId
* MaterialsProducerPropertyId
* PersonPropertyId
* RegionId
* RegionPropertyId
* ResourceId
* ResourcePropertyId

Each identifier is a separate interface type that the modeler must implement as needed. These interfaces are marker interfaces that define no methods and are used solely to enforce explicit type identification. All instances of these identifiers exist when the experiment is built and no new instances can be added during the simulation run.

Example marker interface:

@ThreadSafe

**public** **interface** ResourceId {

}

## **Internal Identifiers**

GCM defines the class, InternalComponentId, which extends the ComponentId marker interface. It is used by GCM to add a synthetic component that drives the memory reporting mechanisms in an efficient manner. When the memory report is active, a component named “Memory Report Internal ComponentId” will appear in the memory and profile reports.

**Thread safety**

Identifiers will be accessed by numerous threads in the experiment and so must be thread safe. Typically, the best strategy is to use immutable objects. Immutable objects must:

1. Declare all fields final
2. allow any reference to the object be passed during its construction

**Equals contract**

All identifiers must comply with the equals contract defined by Object.java. Any identifier class that overrides the implementation of equals() must guarantee that:

1. Equality is reflexive – x.equals(x) is always true
2. Equality is symmetric – x.equals(y) and y.equals(x) return the same value
3. Equality is transitive – x.equals(y) and y.equals(z) implies that x.equals(z)
4. Equality is stable – x.equals(y) will return the same value so long as neither x nor y are mutated
5. No object is equal to null – x.equals(null) is always false
6. hashCode() is also overridden such that equal objects have equal hash codes – x.equals(y) implies x.hashCode() = y.hashCode()

The simplest way to comply with the equals contract is to not override the implementation provided by Object.

**Using enumerations for identifiers**

Most of the marker identifier types will have a relatively small number of instances. For example, a model may contain compartment identifiers for susceptible, exposed, infected and recovered. A typical way a modeler may create these identifiers is to use an enumeration.

**public** **enum** Compartment **implements** CompartmentId {

***SUSCEPTIBLE***, ***EXPOSED***, ***INFECTED***, ***RECOVERED***;

}

The enumeration’s elements will implement the marker interface and no other behavior is required. Additionally, enumerations are both thread safe and immutable when implemented as above.

**Property Identifiers and enumerations:**

Consider the following implementation of GlobalPropertyId identifiers.

**public** **enum** GlobalProperty **implements** GlobalPropertyId {

***TOTAL\_POPULATION***(**new** PropertyDefinition(Integer.**class**, 0)),

***INITIAL\_EXPOSED\_POPULATION***(**new** PropertyDefinition(Integer.**class**, 0));

**private** **final** PropertyDefinition propertyDefinition;

**private** GlobalProperty(PropertyDefinition propertyDefinition) {

**this**.propertyDefinition = propertyDefinition;

}

**public** PropertyDefinition getPropertyDefinition() {

**return** propertyDefinition;

}

}

Each identifier is:

* A GlobalPropertyId
* Immutable – note that the property definition field is declared final
* Thread safe

Furthermore, the enumeration guarantees that each property:

* Will have an accompanying property definition
* Is easy to add to the experiment via a simple loop
* Provides auto-completion and code clarity to the component code that uses the property

# Chapter 5: Components

Components are the core contribution of the modeler. They initiate all activity in the simulation and process people, resources and materials through a series of user defined rules and events. Components implement the Component interface and must have an empty constructor.

**Component life cycle**

1. The component is added as a class reference in the experiment
2. The simulation constructs an instance of the component
3. The simulation initializes the component by invoking its init() method
4. The component registers for observations, takes actions and makes plans
5. The simulation alerts the component with observations, executes actions and returns plans to the component when needed
6. The simulation invokes close() on the component at the end of the run

**Why the class reference and empty constructor?**

GCM is designed to run experiments in a multithreaded manner. Rather than burden the modeler with undue considerations about thread safety, GCM confines the component instances to the threads that are executing the simulation. The component class reference is stored in the experiment and the simulation creates the needed components using an empty constructor.

**Component interface**

The Component interface has four main parts (methods). Each represents a different event that the component needs to handle.

1. init() – Invoked exactly once at the beginning of the run and used to get things going. This is often where the component will register for observation events.
2. observe\*() – Multiple observation methods implemented by modelers. Each is invoked to allow the component to respond to some change in the simulation such as a person entering a compartment (e.g. observePersonArrival()) or a resource inventory change in a region (e.g. observeResourceConsumption()).
3. executePlan() – Invoked by the simulation when a plan that was created by the component has come to fruition. The component is free to take whatever action it wants with the Plan.
4. close() – Invoked exactly once at the end of the run.

## **Four Types of Components**

Components added by the modeler fall into one of four roles.

1. Compartments – Represent the various stages in disease progression and treatment. Compartments work directly on people.
2. Regions – Represent a physical region but do not have any geographic data. Regions possess resources that can be distributed to people by compartments.
3. Global Components – Usually manage global events such executing disease transmission or transitioning treatment policy.
4. Materials Producers – Represent the industrial transformation of materials into the resources used to treat people.

Each component role restricts the range of actions that a component can take. The role restrictions for each mutative action are documented in the Environment class.

Some examples:

* A person can only be deleted by the compartment that is currently associated with the person.
* Resources can only be added directly to a region by a global component or the region itself.
* Stages may only be created by materials producers

These role restrictions serve to ensure that each component exercises a reasonable range of activities and helps the modeler avoid unintuitive behaviors.

Component are added to the experiment aligned to their component type:

experimentBuilder.addRegionId(region, StandardRegion.**class**);

experimentBuilder.addCompartmentId(Compartment.***SUSCEPTIBLE***, SusceptibleCompartment.**class**);

experimentBuilder.addGlobalComponentId(GlobalComponent.***POPULATION\_LOADER***, PopulationLoader.**class**);

experimentBuilder.addMaterialsProducerId(MaterialsProducers.***PRODUCER\_1***, ResourceFactory.**class**);

## **Component Capabilities**

Components are able to take three types of action:

1. **Direct Action:** Take a direct action on a person, resource, material batch or staged material collection. Control a population index.

environment.setPersonPropertyValue(personId, PersonProperty.***SICK***, sick);

environment.addResourceToRegion(Resource.VACCINE, Region\_5, 13000);

environment.addPopulationIndex(filter, ***ASYMPTOMATIC\_INFECTED\_CHILDREN***);

1. **Observation:** Register to start or stop receiving observations.

environment.observeCompartmentPersonArrival(**true**, Compartment.***TREATED***);

environment.observeRegionResourceChange(**false**, Region.***REGION\_1***,Resource.***MEDICATION***);

1. **Planning:** Schedule a plan for future action. Plan is a marker interface – it has no methods. The modeler creates a plan implementation and schedules it with the environment for some future time. The plan is sent back to the component at the scheduled time and the component takes whatever action is appropriate.

TreatmentPlan treatmentPlan = **new** TreatmentPlan(personId);

**double** treatmentDelayDays = 10 \* randomGenerator.nextDouble() + 1;

**double** treatmentEntryTime = environment.getTime() + treatmentDelayDays;

environment.addPlan(treatmentPlan, treatmentEntryTime);

## **Abstract Component**

The AbstractComponent class provides some implementation of the Component interface as a convenience for the modeler:

1. init() – Not implemented, the modeler must write their own implementation
2. observe???() – Each is implemented but will throw a Runtime Exception if invoked. Any observation that is registered by the component must have a corresponding observation method implemented by the modeler.
3. executePlan() – Implemented but will throw a Runtime Exception if invoked. Any plan must be handled by modeler.
4. close() – Implemented but does nothing

## **Example Component**

ExampleComponent to extends AbstractComponent so that only the methods needed are implemented:

**public** **class** ExampleComponent **extends** AbstractComponent {

Next, the modeler implements the init() method. Typically, this is where initial actions involving people are executed, observations are registered and plans are initialized.

@Override

**public** **void** init(Environment environment) {

environment.observeCompartmentPersonArrival(**true**, Compartment.***TREATED***);

environment.observeRegionResourceChange(**true**, Region.***REGION\_1***, Resource.***MEDICATION***);

**for**(PersonId personId : environment.getPeople()){

...

environment.transferResourceToPerson(personId, Resource.***MEDICATION,***2);

}

**for**(PersonId personId : environment.getPeople()){

**...**

environment.addPlan(treatmentPlan, environment.getTime()+0.5);

}

}

Since the component has registered for observations, the modeler must implement the corresponding methods to handle those observations.

@Override

**public** **void** observeRegionResourceChange(**final** Environment environment,**final** RegionId regionId, **final** ResourceId resourceId) {

// take action based on the resource type

}

@Override

**public** **void** observeCompartmentPersonArrival(**final** Environment environment, **final** PersonId personId) {

// take some action or plan relative to this person

}

Similarly, the component has issued plans so the executePlan method must be implemented.

@Override

**public** **void** executePlan(**final** Environment environment, **final** Plan plan) {

// validate the plan to make sure that it is still appropriate

// take some action or re-plan

}

# Chapter 6: Observation

GCM utilizes a few strategies to reduce component complexity and bug-prone code.

* **No Messages:** Components cannot send messages to one another. The component designer should keep intra-component dependencies to a minimum.
* **Thread Confinement:** Components are created by each simulation instance and operate only in the thread of the simulation.
* **Statelessness:**
  + **Environment:** All component public methods are passed the current reference to the Environment. There is no need to retain this reference.
  + **Data Model:** GCM provides a complete data model. Whenever possible, state should be maintained by GCM and not the component.

In GCM’s data model (people, groups, resources, properties, etc.), nearly all data changes can be observed. Components are generally interested in data changes and not in the actions that caused those changes. For example, suppose a vaccination component is waiting for new doses of vaccine to be delivered to a region. These doses could 1) be created directly on the region, 2) be transferred from another region and 3) be transferred from a materials producer component. The vaccination component should only care that new vaccine doses are available and so it registers to observe changes in the vaccine dose inventory level of the region.

Some data changes are not observable since they are private to specific components.

* Plan additions and removals. Plans are not shared between components.
* Population index additions and removals. Each index is controlled by a single component even though all components can access any population index.
* Simulation halt() causes the simulation to stop and is thus not observable.
* Materials activities. Most materials related activities occur strictly within a materials producer. Only produced resources and offered stages are open to registered observation.

The Environment offers a wide array of observation methods to the components. The general pattern for these methods is:

*observe + Item + By + Constraints (boolean observe, constraint, constraint….){}*

For example, (From Environment class)

* **public** **void** observePersonResourceChange(final boolean observe, final PersonId personId, final ResourceId resourceId);
* **public** **void** observeGroupConstructionByType(final boolean observe, final GroupTypeId groupTypeId);
* **public** **void** observeStageOfferChangeByStageId(final boolean observe, final StageId stageId);

The Boolean observe argument serves to register/unregister the component for the observation. Once registered, the component must implement the corresponding public method that will handle the observation. The component will continue to receive all observations until it unregisters. The general pattern for these methods is:

*observe + Item + (Environment environment, item info, item info, … ){}*

For example, (From Component class)

* **public** **void** observePersonResourceChange(Environment environment, PersonId personId, ResourceId resourceId);
* **public** **void** observeGroupConstruction(Environment environment, GroupId groupId);
* **public** **void** observeStageOfferChange(Environment environment, StageId stageId);

Although many of the Environment methods used to register/unregister observations have a one to one relationship to the Component methods used to handle the observations, some methods offer multiple constraint patterns. For example, these four Environment methods:

* **public** **void** observeStageTransfer(final boolean observe);
* **public** **void** observeStageTransferByDestinationMaterialsProducerId(boolean observe, MaterialsProducerId destinationMaterialsProducerId);
* **public** **void** observeStageTransferBySourceMaterialsProducerId(boolean observe, MaterialsProducerId sourceMaterialsProducerId);
* **public** **void** observeStageTransferByStageId(final boolean observe, final StageId stageId);

all correspond to the Component method:

* **public** **void** observeStageTransfer(Environment environment, StageId stageId, MaterialsProducerId sourceMaterialsProducerId, MaterialsProducerId destinationMaterialsProducerId);

**Unregistering observations**

Unregistering an observation must be done via the same method used to register. For example, suppose that a component registers for observing a particular property change for a particular person:

environment.observePersonPropertyChange(**true**, personId, PersonProperty.***INFECTED***);

An invocation of:

environment.observeGlobalPersonPropertyChange(false,PersonProperty.***INFECTED***);

will not unregister the observation. Instead the component must invoke:

environment.observePersonPropertyChange(**false**, personId, PersonProperty.***INFECTED***);

# Chapter 7: People

People are represented as individuals in GCM. Each person is issued a unique identifier, the PersonId, which has an integer value. People are numbered from zero in the order of their creation. Typical use of GCM can support 10 million people. People are associated with various activities:

1. Can be created
2. Can be destroyed
3. Have a region association
4. Have a compartment association
5. Can have properties
6. Can be indexed by their property values
7. Be grouped together
8. Can have resources

## **Creating People**

People can be created in two ways:

1. They can be added to the experiment(uncommon)
2. They can be added during the simulation run

Adding a person to the experiment:

PersonId personId = new PersonId(7);

experimentBuilder.addPerson(personId, Region.***REGION\_1***, Compartment.***UNTREATED***);

}

When adding the person to the experiment, the client supplies the person id. It is important to the operation of the simulation that such ids be packed to a contiguous span of ids starting with zero. To accomplish this, the simulation will reissue contiguous person id values from the data collected in the experiment and transform any relationships associated with those ids accordingly. For example, suppose that person 7 and person 12,000,000 were the only people added to the experiment. The simulation would renumber them as person 0 and person 1 respectively and any data such as region assignments or property values would be properly associated with the new identifiers.

Adding a person during the simulation run:

PersonId personId = environment.addPerson(personId, Region.***REGION\_1***, Compartment.***UNTREATED***);

GCM issues the person id dynamically when a person is created by a component running in the simulation.

## **Destroying People**

Destroying (or removing) a person in GCM removes them and their associated data from the simulation. This should not be confused with a person dying. Death is not a GCM concept and is strictly a modeler contrivance. Dead people are still in the simulation.

environment.removePerson(personId);

A person may only be removed by their current compartment or any global component.

## **People and Regions**

A person always has an associated region. A person’s region is set when they are added to the simulation and the region may be updated by the person’s current region or any global component.

environment.setPersonRegion(personId, Region.***REGION\_5***);

## **People and Compartments**

A person always has an associated compartment. A person’s compartment is set when they are added to the simulation and the compartment may be updated by the person’s current compartment or any global component.

environment.setPersonCompartment(personId, Compartment.***RECOVERED***);

## **People and Properties**

All people share a group of property definitions added to the experiment by the modeler. Each person may have unique property values for each such definition. See *Working with Properties* for more details.

There are two ways to set a person’s property values:

1. In the experiment
2. During the simulation run

Setting a person property in the experiment:

experimentBuilder.addPersonPropertyValue(personId, PersonProperty.***VACCINATED,*** **true**);

Setting a person property during the simulation run: A person property may only be set by the person’s current region, current compartment or any global component.

environment.setPersonPropertyValue(personId, PersonProperty.***VACCINATED,*** **true**);

## **People and Indexes**

People with related property values can be organized by a population index. A population index is defined by an identifier (the name of the index) and a filter that is tested against the entire population. By using a population index, the modeler can significantly reduce the time required to find people that match a complex set of criteria. See *Working with Indexes* for more details.

**Grouping People**

People can also be grouped by close physical contact. These groups usually represent relatively small social organizations such as a family, a religious group or a school. See *Working with Groups* for more details.

**People and Resources**

People may be allocated resource units such as medication dosages and hospital beds. See *Working with Resources* for more details.

# Chapter 8: Groups

People form many small-scale social organizations that affect the transmission and treatment of disease. These groups can be families, religious organizations, schools, etc. The typical goal for representing these groups is to limit/control transmission between individuals in groups in ways that are specific to the type of group or that group’s own particular history and properties.

In GCM the first approach that a modeler might choose to represent groups is to add one or more person-based properties that indicate group membership. For example, consider a model that has family groups. To keep things simple, each person is a member of at most a single family and families are numbered 0, 1, 2, etc. A person property could be defined that accomplishes this:

PropertyDefinition propertyDefinition = **new** PropertyDefinition(Integer.**class**, -1, **true**);

experimentBuilder.definePersonProperty(PersonProperty.***FAMILY\_ID***, propertyDefinition);

The property is defined as an integer with a default value of -1 to represent that a person is defaulted to not having a family. The modeler might choose a single component to manage the creation of groups so that the id values are assigned in a rational fashion. It is likely, however, that multiple components will need to create families dynamically and so the modeler introduces a global property to aid in family id assignment:

PropertyDefinition propertyDefinition = **new** PropertyDefinition(Integer.**class**, 0, **true**);

experimentBuilder.defineGlobalProperty(PersonProperty.***MASTER\_FAMILY\_ID***, propertyDefinition);

The master family id would be incremented each time a family is created.

Integer familyId = environment.getGlobalPropertyValue(GlobalProperty. ***MASTER\_FAMILY\_ID***);

environment.setGlobalPropertyValue(GlobalProperty.***MASTER\_FAMILY\_ID***, familyId + 1);

Finally, to select people from the same family as a given person, the modeler would likely first have to create a population index per family to reduce the time needed to make such selections:

environment.addPopulationIndex(Filters.*property*(PersonProperty.***FAMILY\_ID***, Equality.***EQUAL***, familyId), familyId);

Selecting another family member might then be:

List<PersonId> familyMembers = environment.getIndexedPeople(familyId);

// continued selection logic…

The complexities only grow as we consider what happens if a person can belong to multiple families or multiple types of socials organizations are needed. Further, there is no convenient place to record group properties such as a general contact probability within a particular group.

The fundamental problem is that person properties represent facts about an individual and not relationships between individuals. GCM solves this problem by the Groups mechanism.

Groups:

1. Are associated with a modeler defined group type
2. Can be created
3. Can be destroyed
4. Can have people added
5. Can have people removed
6. Can have properties
7. Support weighted selection

## **Group Types**

Groups have a group type defined by the GroupTypeId marker interface. GroupTypes are defined in the experiment. GroupType.java, used here as an example, is an enumeration that implements GroupTypeId and provides a convenient structure to hold the group types:

**public** **enum** GroupType **implements** GroupTypeId {

***FAMILY***, ***SCHOOL***, ***WORK\_PLACE***;

}

experimentBuilder.addGroupType(GroupType.***FAMILY***);

## **Creating Groups**

Groups can be created in two ways:

1. They can be added to the experiment(uncommon)
2. They can be added during the simulation run

Adding a group to the experiment:

GroupId groupId = new GroupId(7);

experimentBuilder.addGroup(groupId, GroupType.***FAMILY***);

When adding the group to the experiment, the client supplies the group id. It is important to the operation of the simulation that such ids be packed to a contiguous span of ids starting with zero. To accomplish this, the simulation will reissue contiguous group id values from the data collected in the experiment and transform any relationships associated with those ids accordingly. For example, suppose that group 7 and group 12,000,000 were the only groups added to the experiment. The simulation would renumber them as group 0 and group 1 respectively and any data such as group membership assignments or property values would be properly associated with the new identifiers.

Adding a group during the simulation run:

GroupId groupId = environment.addGroup(GroupType.***FAMILY***);

GCM issues the group id dynamically when a group is created by a component running in the simulation.

## **Destroying Groups**

Destroying(removing) a group in GCM removes it from the simulation. Any people associated with the group are no longer associated with the group but are otherwise unaffected.

environment.removeGroup(groupId);

## **Adding People**

People may be added to any number of groups.

Groups can be created in two ways:

1. In the experiment
2. During the simulation run

In the experiment:

experimentBuilder.addPersonToGroup(personId, groupId);

During the simulation run:

environment.addPersonToGroup(personId, groupId\_2);

## **Removing People**

People may be removed from a group.

environment.removePersonFromGroup(personId, groupId);

## **Group Properties**

Groups can have property values. Each group type has zero to many property definitions and a group has property values associated with those definitions.

PropertyDefinition propertyDefinition = new PropertyDefinition(Boolean.**class**, **false**, **true**);

experimentBuilder.defineGroupProperty(GroupType.***FAMILY,*** GroupPropertyType.***SINGLE\_PARENT,*** propertyDefinition);

Groups properties can be assigned in two ways:

1. In the experiment
2. During the simulation run

In the experiment:

experimentBuilder.addGroupPropertyValue(groupId, GroupPropertyType.***SINGLE\_PARENT,*** true);

During the simulation run:

environment.setGroupProperty(groupId, GroupPropertyType.***SINGLE\_PARENT,*** **true**);

## **Selecting People**

**Weighted Selection**

People may be selected from groups via weighted and unweighted selection.

**Unweighted selection**: A person is randomly selected from the group.

environment.getNonWeightedGroupContact(groupId);

**Unweighted selection with exclusion**: A person is randomly selected from the people in the group other than the excluded person.

environment.getNonWeightedGroupContactWithExclusion(groupId, excludedPersonId);

**Weighted selection**: A person is randomly selected from the group based on the weight assigned to each person via a weighting function. The weighting function (shown below) returns for each person in the group a finite, non-negative weight value. The chance of selection of each in person in the group is proportional to the weight value returned by the function.

The functional interface that defines a MonoWeightingFunction:

**public** **interface** MonoWeightingFunction {

**public** **double** getWeight(ObservableEnvironment observableEnvironment, PersonId personId, GroupId groupId);

}

Using the contact methodology in a component: Since MonoWeightingFunction is a functional interface, we choose to use the locally defined method getWeight() as the monoWeightingFunction.

/\*

\* Age [0, 10] returns 50% of the time

\* Age (10, 16] returns 40% of the time

\* Age (16, INF) returns 10% of the time

\*/

**private** **double** getWeight(ObservableEnvironment observableEnvironment, PersonId personId, GroupId groupId) {

**int** age = observableEnvironment.getPersonPropertyValue(personId, PersonProperty.***AGE***);

**if** (age <= 10) {

**return** 1;

}

**if** (age <= 16) {

**return** 0.5;

}

**return** 0.1;

}

MonoWeightingFunction monoWeightingFunction = this::getWeight;

PersonId contactedPersonId = environment.getMonoWeightedGroupContact(groupId, monoWeightingFunction);

**Weighted selection with a source person**: A person is randomly selected from the group based on the weight assigned to each person via a weighting function. The weighting function (shown below) returns for each person in the group a finite, non-negative weight value that is based on that person’s interaction with the person who is the source of the contact. The chance of selection of each in person in the group is proportional to the weight value returned by the function.

The functional interface that defines a BiWeightingFunction:

**public** **interface** BiWeightingFunction {

**public** **double** getWeight(ObservableEnvironment observableEnvironment, PersonId sourcePersonId, PersonId targetPersonId, GroupId groupId);

}

Using the contact methodology in a component: A sick person may contact a person in one of its social groups. The person so contacted will not include the sick person. Since BiWeightingFunction is a functional interface, we choose to use the locally defined method getWeight() as the biWeightingFunction.

**private** **double** getWeight(ObservableEnvironment observableEnvironment, PersonId sourcePersonId, PersonId targetPersonId, GroupId groupId) {

**double** result = 0;

**boolean** immune = observableEnvironment.getPersonPropertyValue(targetPersonId, PersonProperty.***IMMUNE***);

**double** susceptibility = observableEnvironment.getPersonPropertyValue(targetPersonId, PersonProperty.***SUSCEPTIBILITY***);

**if** (!immune) {

result = susceptibility;

}

**return** result;

}

**boolean** excludeSourcePerson = **true**;

BiWeightingFunction biWeightingFunction = **this**::getWeight;

PersonId contactedPersonId = environment.getBiWeightedGroupContact(groupId, sourcePersonId, excludeSourcePerson, biWeightingFunction);

# Chapter 9: Properties

Properties are the primary means for storing data in GCM. Properties have three main features:

1. Definitions
2. Identifiers
3. Values

## **Property Definitions**

The modeler defines all properties. Property definitions are added to the experiment and are fixed during the execution of the experiment. They are composed of:

* Type
* Default Value
* Override policy
* Map Option
* Time Tracking Policy

Example:

**new** PropertyDefinition(Integer.**class**, 5, **false**, MapOption.***NONE***, TimeTrackingPolicy.***DO\_NOT\_TRACK\_TIME***));

Property Type

The type of a Property may be of any class. Typically, a property uses Integer.class, Double.class or other wrapped primitives but the modeler is free to use any type.

Default Value

The default value must not be null and must be consistent with the chosen type. For example, if the type is Number.class, then the default value could be an Integer, a Double or any other descendant type of Number.

Override Policy

The override policy is almost always set to true so that the modeler may assign values to properties during the execution of the simulation. Usually when the override is set to false it is for setting global constants that should not change over the course of the experiment.

Map Option

GCM allows the modeler to find all people who have a particular person property value. Rather than inspect each person (since there can be millions of people) GCM can maintain a reverse map from each value back to the people having that value. While this is more performant, it can be rather expensive in terms of memory overhead. The Map Option is an enumeration having values NONE, ARRAY, and HASH. NONE tells GCM that no reverse mapping should be maintained. The ARRAY option tells GCM to maintain the reverse mapping in a data structure that is almost as fast as a map, but requires far less memory. Finally, the HASH option tells GCM to use a standard Collections based map to store the reverse mapping and achieve the highest performance. If the modeler knows that the reverse mapping will not be needed or will be rarely used, they should choose NONE. Otherwise the best practice is the prefer ARRAY over HASH unless performance is at a premium.

Time Tracking Policy

The time tracking policy is an enumeration having values TRACK\_TIME and DO\_NOT\_TRACK\_TIME. When time tracking is turned on, GCM records the last model time each property value was assigned. GCM does not usually store past state and for some property values this can be very useful information and prevents the modeler from having to create more properties just to track assignment times. Maintaining the model times for each property value assignment of course has an impact on memory and should only be turned on when needed.

## **Property Identifiers**

Some background on Identifiers in GCM

Identifiers in GCM are used to clearly differentiate between content and prevent ambiguity and confusion. Rather than identifying a compartment, a region or property with Strings and other common identifier types, GCM uses prescribed classes to identify information by type. For properties these types are all marker interfaces – empty interfaces having no methods.

Property definitions are identified with one or sometimes more identifiers.

* Batch – uses both a MaterialId and BatchPropertyId so that different batches of materials can have properties that are consistent with the type of material in the batch
* Compartment – uses both a CompartmentId and CompartmentPropertyId so that different compartments can have properties that unique to that compartment
* Global – uses GlobalPropertyId
* Group – uses both a GroupTypeId and GroupPropertyId so that different types of groups can have properties that unique to those group types
* Materials Producer – uses MaterialsProducerPropertyId
* Person – uses PersonPropertyId
* Region – uses RegionPropertyId
* Resource – uses ResourcePropertyId

Example:

Note that the compartment property requires both a compartment identifier as well as a compartment property identifier since this will be a property that is unique to that compartment.

experimentBuilder.defineCompartmentProperty(Compartment.***INFECTED***,

CompartmentProperty.***WEIGHT\_THRESHOLD***,

**new** PropertyDefinition(Double.**class**, 0d, **true**));

## **Property Values**

Property values are unique to the instance of the thing having the property definition. For example, suppose a person property, VACCINATED, was defined by the modeler:

experimentBuilder.definePersonProperty(PersonProperty.VACCINATED,

**new** PropertyDefinition(Boolean.**class**, **false**, **true**));

The modeler here is using an enumeration called PersonProperty that implements the PersonPropertyId marker interface. Thus PersonProperty.VACCINATED is a PersonPropertyId. All people will start with a VACCINATED property value of false and that value can be changed in the future. Finally, the defaulted map option will be NONE and the time tracking policy will be DO\_NOT\_TRACK\_TIME.

It is the individual person who possesses the property value. This pattern of value association holds for

* Batches
* Compartments
* Groups
* Materials Producers
* People
* Regions

Global property values belong to the entire simulation and NOT to Global Components.

Resource property values belong to the resource type and NOT to individual allocations of a resource. For example, suppose that the resource, HOSPITAL\_BED, has the defined Boolean property REQUIRES\_STERILIZATION.

experimentBuilder.defineResourceProperty(Resource***.HOSPITAL\_BED***,

ResourceProperty.***REQUIRES\_STERILIZATION***,

**new** PropertyDefinition(Boolean.**class**, false, **true**));

All allocations of HOSPITAL\_BED will have the same value of REQUIRES\_STERILIZATION at any particular time.

## **Assigning Property Values**

Examples: There are property assignment methods in the Environment for each type of property.

environment.setPersonPropertyValue(personId, PersonProperty.***IMMUNE***, immune);

environment.setBatchPropertyValue(batchId, BatchProperty.***EXPRIATION\_TIME***, 34.5);

Property assignments are subject to a few constraints:

1. Values cannot be null
2. Values must be assignment-compatible with the property definition’s type
3. The property definition must allow value assignment
4. The target receiving the value must exist and the property must be defined
5. The component (Compartment, Region, Materials Producer or Global Component) that is setting the value must have permission to do so

Property assignment permissions are determined by the type of property:

1. Batch properties – Only the batch’s current materials producer
2. Compartment properties – Any global component and the compartment that owns the property value
3. Global properties – Any global component
4. Group properties – Any global component, region or compartment
5. Materials Producer properties – Any global component and the materials producer that owns the property value
6. Person properties – Any global component and the compartment or region where the person currently resides
7. Region properties – Any global component and the region that owns the property value
8. Resource properties – Any global component

## **Some Further Considerations**

*Use default values*

The default values contained in each property definition should be chosen so as to maximize the number of values that are at default. GCM uses default values to reduce memory allocations.

*Favor wrapped primitives*

In general, the modeler should favor the use of wrapped primitives since GCM has specific efficiencies when storing these values. The integer based wrapped primitives Long, Integer, Short and Byte are stored in GCM using the minimum number or bytes needed to represent the largest value given to GCM for that property. There is no need to try to save memory by defining a property to be of type Short instead of the more natural Integer since GCM will automatically internally scale values accordingly. The case for the floating point wrapped primitives Double and Float is more complicated. Floats require approximately half of the memory of Doubles and are often sufficient for storing floating point property values. Unfortunately, Floats and Doubles are not interchangeable and the modeler must decide which one to use.

*Strongly favor Immutability*

Every change to every property value is observable by components. Mutating a property value directly instead of setting a new value via the Environment prevents the Environment from alerting components of the change and thus thwarts the intention of GCM’s design.

For example, suppose that a modeler has introduced a person property definition:

experimentBuilder.definePersonProperty(PersonProperty.VACCINATIONS,

**new** PropertyDefinition(VaccineList.**class**,new VaccineList(), **true**));

where VaccineList is a mutable data structure created by the modeler:

public class VaccineList{

…

public List<String> getVaccines(){…}

public void addVaccine(String vaccineName){…}

}

The modeler continues with the following code snippet in a compartment:

VaccineList vaccineList = environment.getPersonPropertyValue(personId, PersonProperty.VACCINATIONS);

vaccinelist.addVaccine("MMR");

By directly altering the vaccine list, the modeler has prevented GCM from observing the change. A better alternative is to create an immutable class:

public **final** class VaccineList{

private final List<String> vaccines;

public VaccineList(List<String> vaccines){

this.vaccines = new ArrayList<>(vaccines);

}

public List<String> getVaccines(){

return new ArrayList<>(vaccines);

}

}

The compartment code would now be:

VaccineList vaccineList = environment.getPersonPropertyValue(personId, PersonProperty.VACCINATIONS);

List<String>vaccines = vaccineList.getVaccines();

vaccines.add("MMR");

VaccineList updatedVaccineList = **new** VaccineList(vaccines);

environment.setPersonPropertyValue(personId, PersonProperty.VACCINATIONS, updatedVaccineList);

Although GCM seeks to shelter the modeler from concurrency issues where possible, some consideration must be given to this issue. It is natural for property values to be used in reports and other output types. Outputs are generally integrated across multiple threads. Immutable classes are thread safe. It far simpler to reason about the state of properly constructed, immutable classes than to worry about thread safety.

*Requirements for an Immutable Class*

1. The class cannot be mutated
2. All fields are declared final
3. The class is properly constructed: It does not allow a reference to itself to leak outside of its constructor

# Chapter 10: Population Indexes

A GCM modeler will often need to select an individual at random from a population of millions of people. Often the selected individual will need to meet several criteria and the number of potential candidates could be quite small compared to the population. For example, consider a global component designed to vaccinate school aged children at random from a certain region. The modeler will have defined some supporting properties:

experimentBuilder.addRegionId(Region.***REGION\_1***, RegionClass.**class**);

PropertyDefinition propertyDefinition = **new** PropertyDefinition(Boolean.**class**, **false**, **true**);

experimentBuilder.definePersonProperty(PersonProperty.***IMMUNE***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Boolean.**class**, **false**, **true**);

experimentBuilder.definePersonProperty(PersonProperty. ***VACCINATED***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Integer.**class**, 0, **true**);

experimentBuilder.definePersonProperty(PersonProperty.***AGE***, propertyDefinition);

Each time a new vaccine dose becomes available, the vaccinator component could query the entire population and randomly select one of the available candidates:

//create a list to store vaccination candidates

List<PersonId> vaccinationCandidates = **new** ArrayList<>();

//iterate through all people in the model to find candidates

List<PersonId> people = environment.getPeople();

**for** (PersonId personId : people) {

RegionId regionId = environment.getPersonRegion(personId);

Integer age = environment.getPersonPropertyValue(personId,

PersonProperty.***AGE***);

Boolean immune = environment.getPersonPropertyValue(personId, PersonProperty.***IMMUNE***);

Boolean vaccinated = environment.getPersonPropertyValue(personId, PersonProperty.***VACCINATED***);

**boolean** isVaccinationCandidate = (regionId == Region.***REGION\_1***) && (age <= 18) && !immune && !vaccinated;

**if** (isVaccinationCandidate) {

vaccinationCandidates.add(personId);

}

}

//if any candidates were found, select one at random and vaccinate them

**if** (!vaccinationCandidates.isEmpty()) {

**int** selectedCandidateIndex = environment.getRandomGenerator().nextInt(vaccinationCandidates.size());

PersonId selectedPersonId = vaccinationCandidates.get(selectedCandidateIndex);

environment.setPersonPropertyValue(selectedPersonId, PersonProperty.***VACCINATED***, **true**);

}

This approach is exhaustive and inefficient. Various compromises include 1) vaccinating larger groups in bulk and 2) maintaining a permanent list of vaccination candidates. In the first case, the model would no longer reflect how the vaccinations are intended to be distributed. In the second case, the component becomes stateful, and would have to register to observe property and region changes on all people. Finally, GCM’s group concept will not help since the school aged children in Region 1 do not all participate in a single social group and would likely exceed the expected size of groups in GCM (up to a few hundred people).

GCM uses population indexes to significantly reduce the run time and complexity when selecting people with complex criteria. For example, by using population indexes, the global vaccinator component would create a population index on its initialization:

//static identifier for children who need vaccination

**private** **final** Object CHILD\_VACCINATION\_CANDIDATES = **new** Object();

//logic in the init() method

FilterBuilder filterBuilder = **new** FilterBuilder();

filterBuilder.openAnd();

filterBuilder.addRegion(Region.***REGION\_1***);

filterBuilder.addProperty(PersonProperty.***AGE***,Equality.***LESS\_THAN\_EQUAL***,18);

filterBuilder.addProperty(PersonProperty.***IMMUNE***,Equality.***EQUAL***,**false**);

filterBuilder.addProperty(PersonProperty.***VACCINATED***,Equality.***EQUAL***,**false**);

filterBuilder.closeLogical();//closes the AND

Filter filter = filterBuilder.build();

environment.addPopulationIndex(filter, CHILD\_VACCINATION\_CANDIDATES);

GCM will maintain the index as changes are made to person properties, region associations and other relevant data. Each time a new vaccine dose becomes available, the vaccinator component would now perform the simplified logic:

//select a single person from the static population index that has been continuously updated by GCM

Optional<PersonId> optional = environment.getRandomIndexedPerson(CHILD\_VACCINATION\_CANDIDATES);

//The returned optional may indicate that no person was available for //selection

**if**(optional.isPresent()) {

PersonId selectedPersonId = optional.get();

environment.setPersonPropertyValue(selectedPersonId, PersonProperty.***VACCINATED***, **true**);

}

**Population Indexes**

Population Indexes are collections of people maintained by GCM containing all the people who match some complex criteria. They are created by components through the submission of a Filter to the environment. As person properties, group membership, region assignment, compartment assignment or resource allocations are updated on individuals, the relevant population indexes are maintained. Using population indexes can significantly reduce runtime.

**Best practice when adding multiple people**

Often modelers choose to add all the people to the simulation via a global component that acts very early in the simulation run and has only that responsibility. Continuing the examples above, the modeler needs to build a population of 10,000,000 people, 300 of whom have been exposed. There will be 6,000,000 people in Region 1 with 250 people exposed. The remaining 4,000,000 will be in Region 2 with 50 people exposed. People’s ages are distributed uniformly from zero to sixty years. People who are aged 30 and over have a 75% chance to be immune and a 35% chance of having been vaccinated. People under 30 have a 25% chance to be immune and an 80% chance of having been vaccinated. The modeler chooses to reflect this information as a set of global property values:

PropertyDefinition propertyDefinition = **new** PropertyDefinition(Integer.**class**, 6\_000\_000);

experimentBuilder.defineGlobalProperty(GlobalProperty.***REGION\_1\_POPULATION***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Integer.**class**, 4\_000\_000);

experimentBuilder.defineGlobalProperty(GlobalProperty.***REGION\_2\_POPULATION***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Integer.**class**, 250);

experimentBuilder.defineGlobalProperty( GlobalProperty.***INITIAL\_REGION\_1\_EXPOSED***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Integer.**class**, 50);

experimentBuilder.defineGlobalProperty( GlobalProperty.***INITIAL\_REGION\_1\_EXPOSED***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Double.**class**, 0.75);

experimentBuilder.defineGlobalProperty(GlobalProperty.***IMMUNITY\_THIRTY\_AND\_OVER***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Double.**class**, 0.25);

experimentBuilder.defineGlobalProperty(GlobalProperty.***IMMUNITY\_UNDER\_THIRTY***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Double.**class**, 0.35);

experimentBuilder.defineGlobalProperty( GlobalProperty.***VACCINATED\_THIRTY\_AND\_OVER***, propertyDefinition);

propertyDefinition = **new** PropertyDefinition(Double.**class**, 0.80);

experimentBuilder.defineGlobalProperty(GlobalProperty.***VACCINATED\_UNDER\_THIRTY***, propertyDefinition);

The modeler creates a component to initialize the population. In this component’s init() method could be the following:

// The model has two regions

Region[] regions = Region.*values*();

// retrieve the population counts for each region

**int**[] regionPopulations = **new** **int**[2];

regionPopulations[0] = environment.getGlobalPropertyValue(GlobalProperty.***REGION\_1\_POPULATION***);

regionPopulations[1] = environment.getGlobalPropertyValue(GlobalProperty.***REGION\_2\_POPULATION***);

// retrieve the number of exposed people in each region

**int**[] exposedPopulations = **new** **int**[2];

exposedPopulations[0] = environment.getGlobalPropertyValue(GlobalProperty.***INITIAL\_REGION\_1\_EXPOSED***);

exposedPopulations[1] = environment.getGlobalPropertyValue(GlobalProperty.***INITIAL\_REGION\_2\_EXPOSED***);

// retrieve the age related probabilities for immunity and vaccination

**double** immune30AndOverProbability = environment.getGlobalPropertyValue(GlobalProperty.***IMMUNITY\_THIRTY\_AND\_OVER***);

**double** immuneUnder30Probability = environment.getGlobalPropertyValue(GlobalProperty.***IMMUNITY\_UNDER\_THIRTY***);

**double** vaccinated30AndOverProbability = environment.getGlobalPropertyValue(GlobalProperty.***VACCINATED\_THIRTY\_AND\_OVER***);

**double** vaccinatedUnder30Probability = environment.getGlobalPropertyValue(GlobalProperty.***VACCINATED\_UNDER\_THIRTY***);

**int** maxAge = environment.getGlobalPropertyValue(GlobalProperty.***MAX\_AGE***);

RandomGenerator randomGenerator = environment.getRandomGenerator();

// loop through the regions

**for** (**int** i = 0; i < regions.length; i++) {

Region region = regions[i];

// determine the population of the region and the number of people

// who are exposed

**int** populationCount = regionPopulations[i];

**int** exposedPopulationCount = exposedPopulations[i];

**int** exposedCount = 0;

**for** (**int** j = 0; j < populationCount; j++) {

//the first people in the region will be exposed

Compartment compartment;

**if** (exposedCount < exposedPopulationCount) {

compartment = Compartment.***EXPOSED***;

exposedCount++;

} **else** {

compartment = Compartment.***SUSCEPTIBLE***;

}

//create the person

PersonId personId = environment.addPerson(region, compartment);

//set the person's age

**int** age = randomGenerator.nextInt(maxAge);

environment.setPersonPropertyValue(personId, PersonProperty.***AGE***, age);

//based on the person's age, set their immunity and vaccination status

**if** (age < 30) {

**boolean** immune = randomGenerator.nextDouble() < immuneUnder30Probability;

environment.setPersonPropertyValue(personId, PersonProperty.***IMMUNE***, immune);

**boolean** vaccinated = randomGenerator.nextDouble() < vaccinatedUnder30Probability;

environment.setPersonPropertyValue(personId, PersonProperty.***VACCINATED***, vaccinated);

} **else** {

**boolean** immune = randomGenerator.nextDouble() < immune30AndOverProbability;

environment.setPersonPropertyValue(personId, PersonProperty.***IMMUNE***, immune);

**boolean** vaccinated = randomGenerator.nextDouble() < vaccinated30AndOverProbability;

environment.setPersonPropertyValue(personId, PersonProperty.***VACCINATED***, vaccinated);

}

}

After each person is added to the simulation, the code above sets three person-property values. This causes the person to be evaluated against all indexes that are triggered by that person property. This can lead to a person being evaluated multiple times against some of the indexes. Many models have 20 person properties or more. Such a bulk-load of people and their properties should precede the addition of indexes whenever possible to avoid this evaluation overhead.

Order of initialization amongst components can be used to force person loading before indexes are added. Components are initialized in GCM by their types: global components first, then regions, compartments and finally materials producers. Within each component type order is determined by the order of addition to the experiment.

By ensuring that the global component for population loading is added first to the experiment builder, the modeler guarantees that the index evaluations will not slow down the loader.

//add the population loader as the first global component

experimentBuilder.addGlobalComponentId(GlobalComponent.***POPULATION\_LOADER***, PopulationLoader.**class**);

experimentBuilder.addGlobalComponentId(GlobalComponent.***IMMUNIZER***, Immunizer.**class**);

…

## **Adding a Population Index**

Population indexes are added by via environment:

Filter filter = …;

environment.addPopulationIndex(filter, key);

Indexes are accessible to any component via the key identifier used in creating the index. Once added they cannot be updated and should the filter criteria need to change, a new population index will have to be created. Information about the index is accessed through the environment:

List<PersonId> = environment.getIndexedPeople(key);

**int** indexSize = environment.getIndexSize(key);

**boolean** personContained **=** environment.personInPopulationIndex(personId,key);

**boolean** indexExists = environment.populationIndexExists(key);

People may be selected randomly from a Population Index:

PersonId selectedPerson = environment.getRandomIndexedPerson(key);

## **Removing a Population Index**

Population indexes may be removed only by the component that created them.

environment.removePopulationIndex(key);

## Filters

Filters are defined by the Filter interface found in the Filter class:

**public** **static** **interface** Filter {

**public** **boolean** evaluate(ObservableEnvironment observableEnvironment, PersonId personId);

}

The evaluate() method passes a person identifier to the filter to either pass or fail. The Observable Environment reference is used by the filter to get information about the person and differs from the environment in that it does not expose any mutation methods to the filter.

## Filter construction

Filters are constructed using a compositional pattern where filters are added to filters. Filters can be created directly using the either 1) the Filters utility or 2) by using a FilterBuilder.

Using the Filters class

The Filters class contains 15 static methods for creating filters. These include filters for the logical operations AND OR and NOT. Other filters are specific to selecting person properties, regions, resources, groups, and compartments.

Example: Constructing a filter for people in region 1 who have either been vaccinated or are immune.

Filter filter =

Filters.*and*(

Filters.*region*(Region.***REGION\_1***),

Filters.*or*(

Filters.*property*(PersonProperty.***VACCINATED***, Equality.***EQUAL***, **true**),

Filters.*property*(PersonProperty.***IMMUNE***, Equality.***EQUAL***, **true**)

)

);

Another way to accomplish this filter:

Filter vaccinated = Filters.*property*(PersonProperty.***VACCINATED***, Equality.***EQUAL***, **true**);

Filter immune = Filters.*property*(PersonProperty.***IMMUNE***, Equality.***EQUAL***, **true**);

Filter vaccinatedOrImmune = Filters.*or*(vaccinated,immune);

Filter region = Filters.*region*(Region.***REGION\_1***);

Filter filter = Filters.*and*(region,vaccinatedOrImmune);

Using the FilterBuilder

The FilterBuilder class contains methods that mirror capabilities of Filters but can be a bit easier to use:

FilterBuilder filterBuilder = **new** FilterBuilder();

filterBuilder.openAnd();

filterBuilder.addRegion(Region.***REGION\_1***);

filterBuilder.openOr();

filterBuilder.addProperty(PersonProperty.***VACCINATED***, Equality.***EQUAL***, **true**);

filterBuilder.addProperty(PersonProperty.***IMMUNE***, Equality.***EQUAL***, **true**);

filterBuilder.closeLogical();// closes the OR

filterBuilder.closeLogical();// closes the AND

# Chapter 11: Resources

Resources represent the discrete objects or dosages that may be distributed to people. They can represent durable items that can be re-assigned such as a hospital bed or consumable items such as a vaccine dose. Individual units of resources are not given unique identifiers like a person, group, region or compartment and thus do not carry any property values.

Resources are often created during a simulation run from raw materials. However, resources may also be added to the experiment as part of the initial setup or added directly to the simulation during the run. Resources may be owned by material producers, regions and people.

## **Defining a Resource**

Resources are added to the simulation via the Experiment by their Resource Id. ResourceId is a marker interface that helps unambiguously identify resources. Resource.java, used here as an example, is an enumeration that implements ResourceId and provides a convenient structure to hold the resources:

**public** **enum** Resource **implements** ResourceId {

***HOSPITAL\_BED***, ***VACCINE\_DOSE***, ***IBUPROPHEN\_DOSE***, ***RESPIRATOR***;

}

Adding the hospital bed to the experiment:

experimentBuilder.addResource(Resource.***HOSPITAL\_BED***);

## **Creating Resources**

Resources may be created in three ways:

1. By adding the resource to the experiment
2. By creating the resource directly on a region
3. By transforming staged materials on a materials producer

Adding a resource to the experiment:

experimentBuilder.addRegionResourceLevel(Region.***REGION\_3***, Resource.***HOSPITAL\_BED,*** 300);

experimentBuilder.addMaterialsProducerResourceLevel(MaterialsProducer.***VACCINE\_PRODUCER***, Resource.***VACCINE,*** 120000);

Creating the resource directly on a region:

experimentBuilder.addResourceToRegion(Resource.***HOSPITAL\_BED,*** Region.***REGION\_3***, 300);

Transforming staged materials on a materials producer:

StageId stageId = environment.createStage();

environment.moveBatchToStage(formaldehydeBatchId,stageId);

environment.moveBatchToStage(mixedProductBatchId,stageId);

environment.convertStageToResource(stageId, Resource.***VACCINE***, 2400);

The new vaccine doses are owned by the materials producer.

## **Transferring Resources**

Resources may be transferred in four ways:

1. From materials producer to region
2. From region to region
3. From region to person
4. From person to region

From materials producer to region:

environment.transferProducedResourceToRegion(MaterialsProducer.***VACCINE\_PRODUCER***,Resource.***VACCINE***, Region.***REGION\_3***,5000);

From region to region:

environment.transferResourceBetweenRegions(Resource.***VACCINE***, sourceRegionId,destinationRegionId,5000);

From region to person: People get resource from their current region. This is usually performed by a compartment or a global component.

environment.transferResourceToPerson(Resource.***VACCINE***, personId,1);

From person to region: People must return a resource to their current region. This is usually performed by a compartment or a global component.

environment.transferResourceToPerson(Resource.***HOSPITAL\_BED***, personId,1);

## **Destroying Resources**

Resources may be destroyed(removed/expended) in two ways:

1. Removal from a region
2. Removal from a person

Remove resource from a region:

environment.removeResourceFromRegion(Resource.***VACCINE***, 1000);

Remove resource from a person:

environment.removeResourceFromPerson(Resource.***VACCINE***, 1);

## **Resource Properties**

Resources may have property values. These properties are associated with the entire resource and not the specific amounts that have been distributed to regions and people.

experimentBuilder.defineResourceProperty(Resource.***VACCINE***,ResourceProperty. ***EXPIRATION\_DATE***, new PropertyDefinition(Double.class, 1000, true));

# Chapter 12: Materials

Materials represent the raw physical constituents that are assembled or processed to become resources. They are created, grouped and transformed by Materials Producer Components. The end product of this activity is the production of the resources that are distributed to people.

## **Defining a Material**

Materials are added to the simulation via the Experiment by their Material Id. MaterialId is a marker interface that helps unambiguously identify materials. Material.java, used here as an example, is an enumeration that implements MaterialId and provides a convenient structure to hold the materials:

**public** **enum** Material **implements** MaterialId {

***ALUMINUM\_SALT***, ***THIMEROSAL***, ***EGG***, ***FORMALDEHYDE***, ***MSG, MIXED\_PRODUCT***;

}

Adding the thimerosal to the experiment:

experimentBuilder.addMaterial(Material.***THIMEROSAL***);

## **Batches**

A batch is a non-negative amount of a material and is owned by Materials Producer. Batches may be created, destroyed, split, joined, transformed and exchanged. Units of measure for materials are implied by the type of material.

## **Stages**

A stage is a grouping of batches that may transformed into a new batch or the transformed into a discrete amount of resource. The stage represents the transformative process for the conversion of raw materials into resources. Stages may be transferred between material producers by marking a stage as being offered for trade. Stages that are offered will not be transformed until they are no longer offered either through offer cancellation or by transfer to another materials producer.

## **Materials Producers**

A material producer is a component that was added to the experiment as a Materials Producer:

experimentBuilder.addMaterialsProducerId(MaterialsProducer.***VACCINE\_PRODUCER***, SomeComponent.class);

Only materials producers may work with batches and stages. Their primary purpose is to orchestrate the transformation of materials into the resources that are used by people.

## **Creating a Batch**

**Creating a batch in the experiment**

Batches may be added directly to Materials Producers via the experiment.

BatchId convertdBatchId = experimentBuilder.addBatch(batchId, Material.***EGG***, 3.4, MaterialProducer\_1);

**Creating a batch in the simulation**

A materials producer may create a batch. The batch will be automatically associated or owned by the materials producer. Note that GCM provides the batch id as a response to the creation of the batch.

BatchId batchId = environment.createBatch(Material.***EGG***, 1200.0);

## **Destroying a Batch**

A materials producer may destroy a batch that it owns.

environment.destroyBatch(batchId);

## **Splitting/Transferring a Batch**

The material content of a batch may be transferred to another batch of the same material type. Both batches must be owned by the same materials producer and the amount must not leave the source batch with a negative level.

environment.shiftBatchContent(sourceBatchId, destinationBatchId, amount);

## **Joining Batches Into a Stage**

Batches may be grouped together onto a stage. All batches and stages must be owned by the Materials Producer that is performing the staging of the batches. Like batch ids, the stage id is created by GCM.

StageId stageId = environment.createStage();

environment.moveBatchToStage(eggBatchId,stageId);

environment.moveBatchToStage(alumimumSaltBatchId,stageId);

## **Transforming Batches**

Once a group of batches are associated via a stage, they may be transformed in two ways:

1. Creation of a new single batch of perhaps a new material type
2. Creation of a discrete amount of resource

Both transformations effectively destroy the batches and their associated stage.

**Transforming batches into a new batch:**

environment.convertStageToBatch(stageId, Material.***MIXED\_PRODUCT***, 2.4);

The batches and the stage are removed from the simulation and a new batch is created that will be owned by the Materials Producer that performed the conversion.

**Transforming batches into a resource:**

StageId stageId = environment.createStage();

environment.moveBatchToStage(formaldehydeBatchId,stageId);

environment.moveBatchToStage(mixedProductBatchId,stageId);

environment.convertStageToResource(stageId, Resource.***VACCINE***, 2400);

The batches and the stage are removed from the simulation and the resource is added to the Materials Producer that performed the conversion. The resource is now ready to be transferred to a region(s):

environment.transferProducedResourceToRegion(MaterialsProducer\_7, Resource.***VACCINE***, Region.Region\_1,1200);

environment.transferProducedResourceToRegion(MaterialsProducer\_7, Resource.***VACCINE***, Region.Region\_2,800);

environment.transferProducedResourceToRegion(MaterialsProducer\_7, Resource.***VACCINE***, Region.Region\_3,400);

## **Exchanging Stages**

Stages, and the batches they contain, may be exchanged between Materials Producers. To do so, the owning Materials Producer must mark the stage as being “offered” to other Materials Producers:

environment.setStageOffer(stageId,true);

While the stage is offered, its contents cannot be mutated and it may be transferred by any Materials Producer:

environment.transferOfferedStageToMaterialsProducer(stageId,MaterialsProducer\_7);

The stage will now be owned by the new Materials Producer and the stage will be updated so that it is no longer in the offered state.

## **Batch Properties**

Batches may have property values. The properties of a batch are defined by the material of the batch:

experimentBuilder.defineBatchProperty(Material.***EGG***,BatchProperty.***EXPIRATION\_DATE***, new PropertyDefinition(Double.class, 0, true));

Property values for new batches are set to the default values for the associated properties and are not effected by transfers, splits or other operations on the batch.

environment.setBatchPropertyValue(batchId, BatchProperty.***EXPIRATION\_DATE***,157);

# Chapter 13: Output and Reports

## **Background**

**A model’s primary purpose is to produce output that can be analyzed. We will discuss the generation and management of output on three levels, with each successive level reducing the burden on the modeler:**

* Modeler managed output
* GCM managed output
* GCM reports

## Modeler Managed Output

The modeler is always free to create files, write to the console, write entries into a database, or commit any of a host of I/O related activities of their own design and implementation. However, there are two significant issues that arise that can complicate such implementations:

1. **Concurrency**: GCM is usually executing individual simulations at the same time.
2. **Components are thread-confined**: GCM components are the likely source of model output and their instances are created for each simulation so that the components can execute within the simulation’s thread.

Some common problems the modeler must solve:

1. Managing files outside of the operation of the experiment execution
2. Communicating details relating to IO such as file references to the components via global properties
3. Creating thread-safe mechanisms to gather outputs from multiple simulation runs during the experiment’s execution
4. Creating extensive post-processing mechanisms to aggregate data
5. Providing sufficient abstraction of such capabilities to reduce complexity and repetition in component code

## GCM Managed output

GCM includes two interfaces to ease some of these burdens:

1. **Output Item**: A simple interface that just contains the scenario and replication ids
2. **Output Item Handler**: An interface for the receivers of output items. These receivers live outside/above of the experiment.

Together, these interfaces will address:

1. Providing an abstraction layer between the data being outputted and the ultimate disposition of that data.
2. Providing a simple lifecycle that focuses the burden of concurrency on the output item handler

The output item interface contains the scenario and replication ids so that the content of the output item can be associated with the specific simulation instance that generated it. Note that this interface is marked with @Immutable, indicating that implementers of the OutputItem must be immutable classes.

@Immutable

**public** **interface** OutputItem {

**public** ScenarioId getScenarioId();

**public** ReplicationId getReplicationId();

}

Immutable classes have four requirements:

1. They cannot be mutated
2. All their fields are declared final
3. All returned values from its methods are immutable or defensive copies. Primitives like int and double are immutable as are the wrapped primitives such as Integer and Boolean. Strings are immutable as well.
4. They are properly constructed, i.e. they do not pass a reference to themselves during construction.

In practice, creating an output item implementer is quite simple. In the following example we will create an output item implementer that is recording the vaccine level for a region. Note that this is a properly designed immutable class, with all its fields declared final and its constructor properly implemented.

**public** **final** **class** VaccineInventoryOutputItem **implements** OutputItem {

//All public methods are getters

**private** **final** ScenarioId scenarioId; //field is final

**private** **final** ReplicationId replicationId; //field is final

**private** **final** **double** time; //field is final

**private** **final** **long** vaccineInventory; //field is final

**private** **final** RegionId regionId; //field is final

**public** VaccineInventoryOutputItem(**final** ScenarioId scenarioId, **final** ReplicationId replicationId, **final** **double** time, **final** RegionId regionId, **final** **long** vaccineInventory) {

**super**();

//note that “this” is never passed to another object

**this**.scenarioId = scenarioId;

**this**.replicationId = replicationId;

**this**.time = time;

**this**.regionId = regionId;

**this**.vaccineInventory = vaccineInventory;

}

**public** **long** getVaccineInventory() {

//vaccineInventory is a primitive, so it is safe

**return** vaccineInventory;

}

**public** RegionId getRegionId() {

//RegionId is an immutable class, so it is safe

**return** regionId;

}

@Override

**public** ReplicationId getReplicationId() {

//ReplicationId is an immutable class, so it is safe

**return** replicationId;

}

@Override

**public** ScenarioId getScenarioId() {

//ScenarioId is an immutable class, so it is safe

**return** scenarioId;

}

**public** **double** getTime() {

//time is a primitive, so it is safe

**return** time;

}

}

Continuing the example, the modeler adds code to each region component that occasionally records the vaccine inventory and sends the vaccine inventory output item on for further handling.

Region regionId = Region.***REGION\_1***;

**long** vaccineInventory = environment.getRegionResourceLevel(regionId,Resource.***VACCINE***);

VaccineInventoryOutputItem vaccineInventoryOutputItem =

**new** VaccineInventoryOutputItem(

environment.getScenarioId(),

environment.getReplicationId(),

environment.getTime(),

regionId,

vaccineInventory);

environment.releaseOutputItem(vaccineInventoryOutputItem);

The environment will find any output item handlers that can process the VaccineInventoryOutputItem and pass the item to them.

We now need to create an output item handler for the VaccineInventoryOutputItem. Output item handlers must:

1. Be thread safe
   * The simplest way to be thread safe it to make the output item handler stateless, i.e. it has no fields.
2. Handle the start and stop of an experiment
   * An experiment progress log is passed at the start of the experiment and it used to continue an experiment that had been halted. For most implementations this can be ignored. Details on the experiment progress log are covered below.
3. Handle the start and stop of the individual simulations that make up the experiment
4. Indicate the output item subclasses that it processes
5. Process instances of those output item subclasses.
   * This method does not have to fully process the output item. For example, it could cache the item for future use.

Here is a simple implementer of the OutputItemHandler that will display the VaccineInventoryOutputItems in the console:

**public** **final** **class** VaccineOutputItemHandler **implements** OutputItemHandler {

@Override

**public** **void** openSimulation(ScenarioId scenarioId, ReplicationId replicationId) {

System.***out***.println("Simulation has started: scenario = " + scenarioId + " replication = " + replicationId);

}

@Override

**public** **void** openExperiment(ExperimentProgressLog experimentProgressLog) {

//ignoring the experiment progress log

System.***out***.println("Experiment has started");

}

@Override

**public** **void** closeSimulation(ScenarioId scenarioId, ReplicationId replicationId) {

System.***out***.println("Simulation has ended: scenario = " + scenarioId + " replication = " + replicationId);

}

@Override

**public** **void** closeExperiment() {

System.***out***.println("Experiment has ended");

}

@Override

**public** **void** handle(OutputItem outputItem) {

//Using a string builder to create a line in the console

VaccineInventoryOutputItem vaccineInventoryOutputItem = (VaccineInventoryOutputItem)outputItem;

StringBuilder sb = **new** StringBuilder();

sb.append(vaccineInventoryOutputItem.getScenarioId());

sb.append(vaccineInventoryOutputItem.getReplicationId());

sb.append(vaccineInventoryOutputItem.getTime());

sb.append(vaccineInventoryOutputItem.getRegionId());

sb.append(vaccineInventoryOutputItem.getVaccineInventory());

System.***out***.println(sb);

}

@Override

**public** Set<Class<? **extends** OutputItem>> getHandledClasses() {

//returning the Vaccine Inventory Output Item class as the

//single output item sub-type that interests us

Set<Class<? **extends** OutputItem>> result = **new** LinkedHashSet<>();

result.add(VaccineInventoryOutputItem.**class**);

**return** result;

}

}

The modeler adds the output handler to the experiment and GCM coordinates the output items and the handler during the experiment execution.

experimentExecutor.addOutputItemHandler(**new** VaccineOutputItemHandler());

## GCM Reports

While output items and handlers alleviate some of the burden faced by the modeler, other substantial issues remain:

1. **Tabular Files**: Most output needs to be directed to tabular files. File access needs to be synchronized.
2. **Experiment Context**: Output files need the scenario/replication pair generated each line as well as the specific property values and other settings that defined the scenario

To address these problems, GCM includes two implementers of output item and output item handler that are designed to work together.

1. **ReportItem**: An output item descendant class that contains an ordered set of string-based, name-value pairs.
2. **NIOReportItemHandler**: An output item handler that is able to write report items into a tab-delimited file.

A subtle problem arises when writing output from components. Components generally do not care about the cause of a data change, rather they are concerned with data values. For example, a region may gain resources from materials producers, through self-generation, or by resource returns from people. The region is not concerned as to the source of the resource, but only with what needs to be done with the resource.

Output reports are not focused on specific values but on the actions that have taken place. GCM supports 36 actions (mutations to the state of the simulation) that components can perform during a model run. These form a more natural basis for reporting than the observation system of components.

GCM solves these problems with the Reports. A Report:

* Is created by each simulation instance and is thus thread-confined and free from synchronization overhead
* Transforms the 36 actions of the simulation into report items
* Provides substantial aggregation of data in order to lower the burden on the output item handler and ease post-processing by the modeler

## The Report Lifecycle

GCM contains 17 pre-made reports that cover all actions in the simulation. Working with these reports is generally simple:

* The modeler adds a report to the experiment via the experiment executor and specifies the file where the report should be written and perhaps a few settings that tailor the report
* GCM manages the report, report items and report item handler
* The reports are written with content from each simulation instance from the experiment along with headers and table columns that show the experiment settings that differentiate the scenarios

**Simple Reports:** The simple reports generally show property and resource value changes and are essentially trace logs.

1. **Batch Status Report:** Shows the evolving state of each batch of materials, including property values and life cycle actions such staging and batch destruction.
2. **Compartment Property Report:** Shows the compartment property value assignments over time.
3. **Global Property Report:** Shows the global property value assignments over time.
4. **Materials Producer Property Report:** Shows the materials producer property value assignments over time.
5. **Materials Producer Resource Report:** Shows the materials producer resource levels over time.
6. **Region Property Report:** Shows the region property value assignments over time.
7. **Resource Property Report:** Shows the resource property value assignments over time.
8. **Stage Report:** Shows the evolving state of each stage (group of material batches), including life cycle actions such stage custody transfer, conversion to resources and stage destruction.

**Periodic Reports: The periodic reports are produced periodically on either an hourly, daily or end of simulation time period. Production of report items is subject to stimulating events and thus there is no guarantee that items will be produced on every iteration of the reporting period. These reports are generally highly aggregated.**

1. **Compartment Population Report:** Shows the number of people in each region/compartment pair in a periodic manner. Only non-zero person counts are reported
2. **Compartment Transfer Report:** Shows the number of times a person is transferred from one compartment to another within a region. Only non-zero transfer counts are reported.
3. **Group Population Report:** Shows the number of groups having a particular number of people for a given group type.
4. **Group Property Report:** Shows the number of groups having particular values for each group property for a given group type. Only non-zero group counts are reported.
5. **Person Property Interaction Report:** Shows the number of people exhibiting each tuple of person property values for a given region/compartment pair. Only non-zero person counts are reported.
6. **Person Property Report:** Shows the number of people exhibiting a particular value for each person property for a given region/compartment pair. Only non-zero person counts are reported.
7. **Person Resource Report:** Shows the number of people who have/do not have any units of a particular resource with a region/compartment pair.
8. **Region Transfer Report:** Shows the number of times a person transferred from one region to another within a compartment. Only non-zero transfers are reported.
9. **Resource Report:** Shows the creation, transfer or consumption of resources within a region/compartment pair. Some activities have no compartment association and will have a blank compartment field. Only activities with non-zero action counts are reported.

## **Adding Reports**

Reports are added to the experiment executor. Periodic reports will require a period specification and several of the reports can be constrained to focus the content of the report.

Examples:

Most only require a path:

experimentExecutor.addBatchStatusReport(path);

experimentExecutor.addCompartmentPropertyReport(path);

experimentExecutor.addGlobalPropertyReport(path);

experimentExecutor.addMaterialsProducerPropertyReport(path);

experimentExecutor.addMaterialsProducerResourceReport(path);

experimentExecutor.addRegionPropertyReport(path);

experimentExecutor.addResourcePropertyReport(path);

experimentExecutor.addStageReport(path);

Some require a path and a period:

experimentExecutor.addCompartmentPopulationReport(path, ReportPeriod.***DAILY***);

experimentExecutor.addCompartmentTransferReport(path, ReportPeriod.***HOURLY***);

experimentExecutor.addGroupPopulationReport(path, ReportPeriod.***END\_OF\_SIMULATION***);

experimentExecutor.addRegionTransferReport(path, ReportPeriod.***DAILY***);

Some can accept vararg parameters:

experimentExecutor.addPersonPropertyInteractionReport(path, ReportPeriod.***DAILY***, PersonProperty.***AGE***, PersonProperty.***HEIGHT…***);

experimentExecutor.addPersonPropertyReport(path, ReportPeriod.***DAILY***, PersonProperty.***IMMUNE,*** PersonProperty.***PEVIOUSLY\_INFECTED…***);

experimentExecutor.addResourceReport(path, ReportPeriod.***HOURLY***, Resource.***RESOURCE\_1***, Resource.***RESOURCE\_2…***);

A few take more complex arguments:

**boolean** reportPeopleWithoutResources = **true**;

**boolean** reportZeroPopulations = **false**;

experimentExecutor.addPersonResourceReport(path, ReportPeriod.***HOURLY***, reportPeopleWithoutResources, reportZeroPopulations, Resource.***RESOURCE\_1***,Resource.***RESOURCE\_3***);

GroupPropertyReportSettingsBuilder groupPropertyReportSettingsBuilder = **new** GroupPropertyReportSettingsBuilder();

groupPropertyReportSettingsBuilder.addGroupPropertyId(GroupType.***HOME***, GroupProperty.***SHARED\_IMMUNITY***);

groupPropertyReportSettingsBuilder.addGroupPropertyId(GroupType.***HOME***, GroupProperty.***TRANSMISSION\_RATE***);

GroupPropertyReportSettings groupPropertyReportSettings = groupPropertyReportSettingsBuilder.build();

experimentExecutor.addGroupPropertyReport(path, ReportPeriod.***DAILY***, groupPropertyReportSettings);

## **Experiment Columns**

When an experiment is defined, any of the scenario variables that had multiple assigned values define a dimension of the experiment and are thus included as experiment columns in the reports. The experiment columns can be turned on or off for all reports. The default behavior is that they are turned on.

experimentExecutor.setDisplayExperimentColumnsInReports(**true**);

The experiment columns may be reported independently to a single file:

experimentExecutor.addExperimentColumnReport(path);

It is usually convenient to have experiment columns turned on for all reports. However, this can lead to very large files since the number of variables that define the experiment space can be numerous and the data in the experiment columns is repetitive. To reduce the file sizes, the modeler can choose to turn off the experiment columns and opt to report those columns once to the experiment column report.

## **Custom Reports**

GCM accepts custom reports:

experimentExecutor.addCustomReport(Path path, Class<? **extends** Report> reportClass, Object... initializationData)

The report class reference is to a modeler-generated Report implementing class that **must** have an empty constructor. The initialization data are a list of objects that will be presented to each report instance and are used to tailor the report’s behavior. All initialization objects **must** be thread safe.

### **Implementing initialization data**

The best approach to creating initialization data is to use immutable classes since they are automatically thread safe. An immutable class is one where **all** fields are declared final and no reference to the instance of class is shared during the instance’s construction.

### **Implementing a custom report**

There are four behaviors to implement when creating a custom report:

1. Initialization
2. Registration for state changes
3. Handling state change events
4. Finalization

### **Initialization**

The init() method of Report must be implemented. It is invoked exactly once at the start of the simulation. The observable environment is a non-mutable interface into the full environment that is used by non-component contributions to GCM. It provides all of the data needed to help the report initialize while not allowing the report to affect the simulation. The initial data is a set of objects passed to the report to aid in tailoring the report’s behavior.

**public** **void** init(**final** ObservableEnvironment observableEnvironment, Set<Object> initialData);

### **Registration for state changes**

A report registers to observe various state changes via the getListenedStageChanges method:

**public** Set<StateChange> getListenedStateChanges();

StateChange is an enumeration used to indicate to GCM which types of data change should be handled by each report. It contains the following elements:

* MATERIALS\_PRODUCER\_RESOURCE\_ADDITION
* MATERIALS\_PRODUCER\_RESOURCE\_TRANSFER
* STAGE\_OFFERED
* STAGE\_TRANSFERRED
* STAGE\_CONVERTED\_TO\_BATCH
* STAGE\_CONVERTED\_TO\_RESOURCE
* STAGE\_CREATION
* BATCH\_CREATION
* STAGE\_DESTRUCTION
* BATCH\_DESTRUCTION
* BATCH\_SHIFT
* BATCH\_STAGED
* BATCH\_UNSTAGED
* BATCH\_PROPERTY\_VALUE\_ASSIGNMENT
* PERSON\_ADDITION
* PERSON\_REMOVAL
* GROUP\_ADDITION
* GROUP\_REMOVAL
* GROUP\_MEMBERSHIP\_ADDITION
* GROUP\_MEMBERSHIP\_REMOVAL
* REGION\_ASSIGNMENT
* COMPARTMENT\_ASSIGNMENT
* COMPARTMENT\_PROPERTY\_VALUE\_ASSIGNMENT
* GLOBAL\_PROPERTY\_VALUE\_ASSIGNMENT
* GROUP\_PROPERTY\_VALUE\_ASSIGNMENT
* PERSON\_PROPERTY\_VALUE\_ASSIGNMENT
* REGION\_PROPERTY\_VALUE\_ASSIGNMENT
* MATERIALS\_PRODUCER\_PROPERTY\_VALUE\_ASSIGNMENT
* RESOURCE\_PROPERTY\_VALUE\_ASSIGNMENT
* PERSON\_RESOURCE\_ADDITION
* REGION\_RESOURCE\_ADDITION
* REGION\_RESOURCE\_REMOVAL
* PERSON\_RESOURCE\_REMOVAL
* PERSON\_RESOURCE\_TRANSFER\_TO\_REGION
* REGION\_RESOURCE\_TRANSFER\_TO\_PERSON
* INTER\_REGION\_RESOURCE\_TRANSFER

Each member of this enumeration is documented in GCM and covered below as well. A report will receive all and only events that correspond to the state changes it returns in the getListenedStateChanges() method.

### **Handling state change events**

There is a handler method defined in Report for each of the 36 state changes. The general pattern for these methods is:

**public** **void** handle???(ObservableEnvironment observableEnvironment, **final** Type1 instance1, Type2 instance2, …);

The observable environment is provided so that the report can retrieve any additional information needed. The data change that stimulates the report’s handle method is executed **before** the report’s handle method is invoked. Since GCM does not store past data state (except for property assignment times) there may be some need to pass information to the handle method that reflects the state of data prior to the data change. For example, if a person moves from one compartment to another, the observable environment will only be able to provide the current compartment for the person and not the previous compartment. GCM provides the previous compartment value as an auxiliary argument for the report’s convenience. For example, the handle method for compartment assignments is:

**public** **void** handleCompartmentAssignment(ObservableEnvironment observableEnvironment, **final** PersonId personId, **final** CompartmentId sourceCompartmentId);

The source compartment is provided directly since it cannot be determined from the observable environment.

### **Finalization**

A report is finalized at the end of the simulation to allow the report to produce any final content. Finalization is performed by the close() method which is invoked exactly once at the end of the simulation run.

**public** **void** close(ObservableEnvironment observableEnvironment);

### **Abstract Report**

GCM provides the class AbstractReport as a default implementer of Report. It provides:

* An empty implementation of init()
* An empty implementation of close()
* No implementation of getListenedStateChanges()
* An implementation of each of the handle???() methods that throw an exception if invoked

Custom report classes should generally extend AbstractReport for convenience. The custom report class will override each handle method it needs leaving the rest to throw exceptions. GCM will only invoke those methods that are needed by the custom report as registered via the getListenedStateChanges() method that the custom report must implement.

### **Generating Report Items**

All reports generate ReportItem objects and send them to the embedded output item handler for writing to the relevant report files. Report items are generated via the ReportItemBuilder class and sent to the output item handler via the observable environment.

Example of Report item generation:

**final** ReportItemBuilder reportItemBuilder = **new** ReportItemBuilder();

//1. add a report header

reportItemBuilder.setReportHeader(getReportHeader());

//2. set the report type

reportItemBuilder.setReportType(getClass());

//3. set the scenario and replication data

reportItemBuilder.setScenarioId(observableEnvironment.getScenarioId());

reportItemBuilder.setReplicationId(observableEnvironment.getReplicationId());

//4. add the data columns entries

reportItemBuilder.addValue(personId);

reportItemBuilder.addValue(observableEnvironment.getTime());

reportItemBuilder.addValue(activity);

reportItemBuilder.addValue(details);

//5. build the report item

ReportItem reportItem = reportItemBuilder.build();

//6. send the report item to the output handler

observableEnvironment.releaseOutputItem(reportItem);

There are six steps in generating a report item:

1. **Add a report header:** Report headers are created via the ReportHeaderBuilder class that collects just the unique column names associated with the report. The resulting report header object is immutable and thus thread safe. A custom report should create a single instance of the report header and reuse the header for every report item it creates to reduce memory and runtime.
2. **Set the report type:** Set the report type the to the class type of the custom report. The embedded output item handler uses this class reference to identify which file will receive the report item.
3. **Set the scenario and replication data:** Set the scenario and replication identifiers to the value of the simulation instance.
4. **Add the data columns:** Add the values for each column in the report in the order consistent with the header.
5. **Build the report item:** Invoke the build() method on the builder.
6. **Send the report item to the output handler:** Use the observable environment to send the report item

### **Example Custom Report**

Included in this documentation is the file *PersonLifeReport.java* that serves as an example of a custom report.

This custom report implementation is focused on all the events associated with one person. This will require that the report is registered for the state changes associated with people:

* COMPARTMENT\_ASSIGNMENT
* GROUP\_MEMBERSHIP\_ADDITION
* GROUP\_MEMBERSHIP\_REMOVAL
* PERSON\_ADDITION
* PERSON\_REMOVAL
* PERSON\_PROPERTY\_VALUE\_ASSIGNMENT
* PERSON\_RESOURCE\_ADDITION
* PERSON\_RESOURCE\_REMOVAL
* PERSON\_RESOURCE\_TRANSFER\_TO\_REGION
* REGION\_RESOURCE\_TRANSFER\_TO\_PERSON
* REGION\_ASSIGNMENT

The report will act as a trace log where each of the causes of change above is associated with a string descriptor. It will have four columns:

* Person – the id of the person being tracked
* Time – the time when the event occurred
* Activity – the descriptor of the event
* Details – auxiliary data that provides additional context

The report must have one handler method for each of the eleven registered state change events. Each will check that the person in question is the person that is the focus of the report. If so, it will create a report item and send it to the output handler via the observable environment.

# Chapter 14: Auxiliary Reports

GCM can produce four auxiliary reports:

1. Console Output
2. Profiling Report
3. Planning Queue Report
4. Memory Report

## Console Output

GCM produces console output showing the progress of an experiment. You can control this output with the command to the experiment executor:

experimentExecutor.setConsoleOutput(**true**);

As the experiment progresses the report will produce log items for each percent of progress:

4 of 15 scenario replications, 26% complete. Expected experiment completion in 0:00:06

When the run is finished the report will itemize the total number of simulation instances executed and report the time it took:

Experiment finished with 150 of 150 successfully completed in 0:00:35

If any simulations encounter an exception, the report will dump a stack trace and indicate which scenario and replication caused the failure:

Failure for s cenario 13 under replication 10

Simulation failure for scenario 13 and replication 10

java.lang.RuntimeException: something bad happened

at barda.gcm.test.manual.demo.components.Immunizer.init(Immunizer.java:89)

at barda.gcm.simulation.EventManagerImpl.execute(EventManagerImpl.java:124)

at barda.gcm.simulation.Context.execute(Context.java:202)

at barda.gcm.simulation.Simulation.execute(Simulation.java:105)

At the end of experiment run that contained exceptions the report will list the first 100 failures

Experiment finished with 145 of 150 scenario replications successfully completed in 0:00:31

Failed simulations

Scenario 2 Replication 1

Scenario 1 Replication 2

Scenario 15 Replication 8

Scenario 14 Replication 9

Scenario 13 Replication 10

## Profile Report

Profiling gives the modeler insight into how much time is being consumed by the various internal mechanisms of GCM as well as for contributed elements such as components and filters. The profile report records various statistics for each method invocation in GCM. The statistics are:

* The scenario and replication
* The name of the method and its class
* The active component when the method is invoked
* The total duration in milliseconds for the method
* The number of times the method was invoked
* The min, mean, max and standard deviation of invocation durations

ADD OUTPUT EXAMPLE

The data are collected in a call hierarchy so that the invocation of each method under different call stacks can be differentiated. For example, the method PersonPropertyManager.getPropertyValue() will likely be invoked in several different conditions from different invoking parent methods. The report captures this via parent-child id columns. The parent-child ids will often repeat across multiple simulation runs, but cannot be relied upon to identify individual methods. For example, getPropertyValue() may be given id = 356 for several replications, but a new replication may invoke a previously unused method, resulting in getPropertyValue() having id = 360.

**Engaging the report**

The profile is turned on by giving the experiment executor a file path to record the report.

experimentExecuter.setProfileReport(path);

Generally, the profile will slow down GCM by a factor of approximately five. It is also advisable to use the main thread so that processor time sharing does not complicate the analysis of the report.

experimentExecuter.setThreadCount(0);

**Report Fields**

* Scenario – identifies the simulation instance
* Replication – identifies the simulation instance
* Id – the integer identifier that is assigned to the method
* Parent Id – the integer identifier that is assigned to the parent method
* Depth – the depth of the call stack, relative to the Context class of GCM
* Component – the component that was active when the method was invoked. If no component was active, such as during simulation initialization, “GCM” is indicated
* Class – the class that owns the method
* Method – the method’s name
* Total Time – the total number of milliseconds used by the method for the given call stack
* Count – the number of invocation of the method
* Min – the minimum time of execution of the method
* Max – the maximum time of execution of the method
* Mean – the mean time of execution of the method
* Standard Deviation – the standard deviation of execution times

## Planning Queue Report

The planning queue report gives the modeler insight into the size of the planning queue over time.

ADD A LOT MORE DETAIL

**Why is the queue important?**

The planning queue may have just a few plans or may have millions of plans depending on modeling choices made when designing components. Most of the simulation’s memory usage scales linearly to the number of people. The planning queue however can often cause a severe rise in memory consumption, especially at the beginning of a simulation run.

**What are keys and why are they important?**

When a plan is added to GCM by a component, an optional key may be included. This key allows the component to either retrieve or remove the plan before its execution. GCM uses a map of key to plan to track such plans which can be memory-intensive when the queue depth is large. Thus, the modeler should only use keys when they are needed by a component to function properly. To aid in analysis, the planning queue report contains a key field that indicates when keys were included with the various types of plans.

The planning queue report records various statistics:

* Scenario and Replication
* Planning class type and owning component
* Key status
* Various statistics about the reported planning class, component and key triplet of values

**Engaging the report**

The planning queue report is turned on by giving the experiment executor a file path to record the report and supplying a threshold number of queue changes that will trigger reporting.

long threshold = 10000;

experimentExecuter.setPlanningQueueReport(path, threshold);

GCM will produce a full accounting of the planning queue each time the number of additions or deletions from the queue reaches the threshold.

**Report Fields**

* Scenario – identifies the simulation instance
* Replication – identifies the simulation instance
* Component – the component that caused the queue to change
* Class – the class type of the plan
* Keyed – a Boolean showing the presence of a key value
* Start Time – the simulation time when the statistics were started
* End Time – the simulation time when the statistics were ended
* Count – the number of additions/deletions from the queue represented by each report line
* Min – the minimum size of the queue subset during the time period
* Max – the maximum size of the queue subset during the time period
* Mean – the mean size of the queue subset during the time period
* Standard Deviation – the standard deviation of the queue subset size during the time period

## Memory Report

The memory report gives the modeler insight into how much memory is being consumed by the various internal mechanisms of GCM and is organized by class and instance. The memory report records various statistics:

* Scenario and Replication
* Class type and its description
* The model time when the memory was recorded
* The number of bytes directly owned by the class and its subordinates

The byte counts are collected in a parent-child hierarchy on a periodic basis.

* Context – top level simulation object
  + Replication
  + Scenario
  + Stochastics Manager
  + Component Manager
    - Each component Identifier
  + Reports Manager
  + Environment
  + Memory Report Manager
  + Observation Manager
  + Materials Manager
  + Resource Manager
  + Property Definition Manager
  + Property Manager
    - Each person property
  + Person Location Manger
  + Person Group Manger
  + Event Manager;
  + Indexed Population Manager
    - Each indexed population — named by the creating component
  + Observable Environment
  + Person Id Manager
  + External Access Manager
  + Mutation Resolver
  + Output Item Manager
    - Each output item handler – named by the handler’s class type
  + Profile Manager
  + Planning Queue Report Item Manager

**Engaging the report**

The memory report is turned on by giving the experiment executor a file path to record the report and supplying a memory report interval (measured in simulation days):

**double** memoryReportInterval = 10;

experimentExecuter.setMemoryReport(path, memoryReportInterval);

GCM will produce a full accounting of memory at each occurrence of the memory report interval. The memory report will slow down GCM significantly. It is advisable to use a reasonably small scenario (low number of people) since most results will scale linearly.

Byte counts in the report reflect a best effort from GCM to account for memory usage. In practice it is difficult to fully account for all objects and internal mechanisms of the JVM. JVM implementors are not bound to use specific byte structures and modern compilers can be quite subtle. This report is best used in comparative analysis.

**Report Fields**

* Scenario – identifies the simulation instance
* Replication – identifies the simulation instance
* Id – the integer identifier that is assigned to the object
* Parent Id – the integer identifier that is assigned to the parent object
* Depth – the depth of the call stack, relative to the Context class of GCM
* Class – the class that owns the method
* Descriptor – the description of the class/instance
* Time – the simulation time when the line of the report was recorded
* Self Byte Count – the number of bytes associated directly with the object
* Child Byte Count – the number of bytes associated with the object’s children
* Total Byte Count – the total number of bytes associated with the object

### Java 8

To use the memory report, GCM must be run under Java 8. Newer versions of Java restrict the use of reflection in a way that is incompatible with the report.

# Chapter 15: Test Plan

**Synopsis**

GCM provides a simulation core, a data model and various conveniences to a compartment modeler. As such, the testing of GCM covers only the behavior of GCM proper and not the behavior of any one model or contributed component.

**Statistics for GCM production code**

* 25,000 lines of Java code(non-test)
* 17,000 lines of comments
* 181 classes
* 2188 public methods

## **General Goals of Testing**

GCM uses unit testing to individually test the public methods of classes in the simulation. Only the concrete methods that are introduced in GCM are tested. Thus interfaces and methods that are introduced by Object, Enum and other core Java classes are generally not tested. Unit tests are conducted using the JUnit framework.

**Proxy tests**

There is an emphasis on testing the parts of the simulation that the modeler would use directly. For example, the class PersonLocationManger.java is used to manage the region and compartment assignments for each person. Instances of this class are never presented to the modeler. Instead the modeler’s components get all region and compartment information from the Environment.java interface. Since GCM tests the Environment’s implementation class completely, there is no need to conduct a direct test of the PersonLocationManger.java class. Such classes are considered to be tested by proxy. Exceptions to this proxy rule are made in cases where GCM tests do not cover the full range of capability of some data structure classes.

**Test Automation**

GCM uses JUnit to conduct 576 individual tests contained in 37 test classes. Testing is automated and is performed each time GCM is compiled. The automation is performed by SuiteTest.java.

**Statistics for GCM test code**

* 26,000 lines of Java code
* 11,000 lines of comments
* 37 JUnit test classes
* 576 unit tests

## **Test Requirements**

Each JUnit test functions as a stand-alone capability. All preconditions are tested to show that they throw the expected Exception type with the expected details. Post condition and invariant condition testing is generally executed across one to several example cases. Tests make assertions that validate the tests themselves. For example, a test showing that people can be removed from a group must first test that the population of the simulation is not empty.

About 80% of the test suite is contained in the test class AT\_EnvironmentImpl.java (The AT\_ prefix designates that this is an automated test). It tests the Environment that is presented to modeler-contributed components. This is the modeler’s single interface into querying and mutating data in GCM. To test the environment, a simulation instance must be created containing a small, but representative scenario. Components are created within the test and given tasks that contain the JUnit assertions. The simulation is executed and results are gathered from the components to show that all sub-tests were executed.

**Example Environment Test:**

**[Each test method documents the specific source method being tested]**

/\*\*

\* Tests {@link Environment#destroyBatch(BatchId)}

\*/

@Test

**public** **void** testDestroyBatch() {

/\*

\* Show that batches can be destroyed and that the environment

\* acknowledges that we destroyed the batches

\*/

**[Each JUnit test that uses a simulation instance needs a random seed value and needs repeatability across hundreds of tests. Tests may come and go but must remain repeatable. The test suite accomplishes this goal the seed provider mechanism]**

**final** **long** seed = SEED\_PROVIDER.getSeedValue(193);

RandomGenerator randomGenerator = *getRandomGenerator*(seed);

**[The scenario is filled with a standard variety of components, property definitions, resources, etc. Each test may employ some variant of this data to better fit its needs]**

ScenarioBuilder scenarioBuilder = **new** UnstructuredScenarioBuilder();

*addStandardTrackingAndScenarioId*(scenarioBuilder, randomGenerator);

*addStandardComponentsAndTypes*(scenarioBuilder);

**[The assertions are executed by components that are in a running simulation. Each component is given zero to many task plans that the component will execute on a schedule. These task plans are created below and gathered in a task plan container. The task plan container is added to the scenario as a global property so that the components can retrieve and execute the task plans accordingly]**

TaskPlanContainer taskPlanContainer = *addTaskPlanContainer*(scenarioBuilder);

**[The scenario now contains all the data. Note that the task plan container is mutable and be filled with task plans. Mutable data structures are generally discouraged in GCM since they will likely violate concurrency policies when running GCM as a multi-threaded experiment.]**

Scenario scenario = scenarioBuilder.build();

Replication replication = *getReplication*(randomGenerator);

**int** testTime = 1;

**[A task plan contains a task, the component id and a time value. The task utilizes a functional interface to simplify typography. In this task, the component will test batch destruction. It first creates the batch and asserts that the batch indeed exists. Next it destroys the batch and asserts that the batch no longer exists.]**

taskPlanContainer.addTaskPlan(TestMaterialsProducerId.***MATERIALS\_PRODUCER\_1***, testTime++, (environment) -> {

**final** BatchId batchId = environment.createBatch(TestMaterialId.***MATERIAL\_1***, 1);

*assertTrue*(environment.batchExists(batchId));

environment.destroyBatch(batchId);

*assertFalse*(environment.batchExists(batchId));

});

**[Precondition tests rely on Model Exceptions being thrown when a precondition is violated. The Model Exception contains a Simulation Error Type (an enum value) that further details the cause of the exception. GCM guarantees that the simulation fully recovers from Model Exceptions and the state of the simulation is unaffected by the invalid request. Thus, the simulation instance can continue to be used for subsequent tests.]**

// precondition tests

taskPlanContainer.addTaskPlan(TestMaterialsProducerId.***MATERIALS\_PRODUCER\_1***, testTime++, (environment) -> {

// if the batch id is null

*assertModelException*(() -> environment.destroyBatch(**null**), SimulationErrorType.***NULL\_BATCH\_ID***);

// if the batch id is unknown for the materials producer

*assertModelException*(() -> environment.destroyBatch(**new** BatchId(-1)), SimulationErrorType.***UNKNOWN\_BATCH\_ID***);

// if the batch is part of an offered stage

**final** BatchId batchId = environment.createBatch(TestMaterialId.***MATERIAL\_1***, 1);

**final** StageId stageId = environment.createStage();

environment.moveBatchToStage(batchId, stageId);

environment.setStageOffer(stageId, **true**);

*assertModelException*(() -> environment.destroyBatch(batchId), SimulationErrorType.***OFFERED\_STAGE\_UNALTERABLE***);

});

**[A final precondition test requires that a component attempt to destroy a batch not under its control. The first component will produce the batch and the second component will attempt to destroy the batch.]**

taskPlanContainer.addTaskPlan(TestMaterialsProducerId.***MATERIALS\_PRODUCER\_1***, testTime++, (environment) -> environment.createBatch(TestMaterialId.***MATERIAL\_1***, 1));

taskPlanContainer.addTaskPlan(TestMaterialsProducerId.***MATERIALS\_PRODUCER\_2***, testTime++, (environment) -> {

**final** List<BatchId> inventoryBatches = environment.getInventoryBatches(TestMaterialsProducerId.***MATERIALS\_PRODUCER\_1***);

*assertTrue*(inventoryBatches.size() > 0);

**final** BatchId batchId = inventoryBatches.iterator().next();

// if invoker is not the materials producer component

*assertModelException*(() -> environment.destroyBatch(batchId), SimulationErrorType.***COMPONENT\_LACKS\_PERMISSION***);

});

**[The task plans are now loaded into the scenario. A simulation instance is created and executed, causing each task above to be executed at the scheduled time.]**

Simulation simulation = **new** Simulation();

simulation.setReplication(replication);

simulation.setScenario(scenario);

simulation.execute();

**[Finally, we show that each task was executed and that the failure to throw an assertion error strongly indicates that the test passes]**

*assertAllPlansExecuted*(taskPlanContainer);

}

## **Meta Level Testing**

GCM is large enough that it can be difficult to show the completeness of testing. To remedy this, GCM uses two annotations applied at the class level: @Source and @UnitTest.

Each source class is annotated with an @Source annotation. The @Source annotation has a single enum parameter indicating the testing requirements. Each test class has an @UnitTest annotation that documents the source class it is testing.

**@Source Testing Requirements:**

* PROXY – The source class does not have instances directly exposed to the modeler and is fully tested by proxy via another class such as Environment
* REQUIRED – The source class requires testing and it expected that it will have a corresponding test class
* UNEXPECTED – The source class could have a test class, but it is of low priority.
* UNREQUIRED – Some classes do not require any testing. For example, enumerations that do not introduce behavior do not generally require testing.

**Test Plan Script**

GCM contains a manual test (TestPlanScript.java) that utilizes these annotations to build a report to console showing the following:

* Summary of source class status
* Overloaded methods in the source classes
* Does each proxied source class have a proxy class that corresponds to another known source class?
* Does each proxied source class link to a tested source class?
* Does each source method have a test method?
* Does any test class not have a legitimate source class?
* Does each source class that has a non-default proxy class have a PROXY status?
* Does each source class that requires a test class have one?
* Does any source class have multiple test classes?
* Does the SuiteTest have all of the automated tests and only the automated tests?
* Does any source class that should not have a test class have one?

For question items above, the report either prints an affirmation that the meta-test passes or a listing showing the question and tabular results showing possible errors.