## Change Log

## Note: the diagrams in this specification are no longer up-to-date or complete. Please refer to Project 5-8 Diagrams – 2020MMDD.pdf for the most current set of diagrams.

|  |  |
| --- | --- |
| Date | Description |
| 4/21/2020 | * Added TSA Security Simulation section describing the Project 8 submission. |
| 4/18/2020 | * SimQueue: corrected specification for assignDestination and assignServer methods (bottom of page 13) which incorrectly said these functions should accept 0 arguments. * SimulationAnalysis: corrected interpolation argument from ‘lower’ to ‘higher’. Also changed hint for calculating the inverse cumulative probability to include Pandas.DataFrame.quantile. |
| 4/16/2020 | * SimQueue: edited/modified the description. No additional information, just a reordering to (hopefully) improve the readability. |
| 4/10/2020 | * Server: added new attribute and getter property for availableSince which should be set to the simtime when a transition is made to AVAILABLE, but should be set to math.inf when in any state *except* AVAILABLE. This will be used by a *new* Assigner method that can will select the server instance that has been available the longest (thereby distributing the customers more fairly among the servers). * SimQueue:   + Renamed removeDestination to removeCustomerDestination (to be more consistent with name of the *add* function)   + Removed setAssignDestination and setAssignServer methods. They are not needed because we are providing properties. * SystemExit:   + Modified acceptArrival to take parameters in order of simtime, Customer to correspond to acceptArrival specified by CustomerDestination. |
| 4/09/2020 | * Server: added additional description of Server’s methods, including processEvent * Simulation: added additional description of Simulation’s methods, including run * SimulationAnalysis: added additional description of SimulationAnalysis’s methods, including analyzeSystemPerformance and compareSystemPerformance. |
| 4/02/2020 | * Server: added boolean return values to pauseService and resumeService (returns False if change cannot be made, True otherwise) |
| 3/23/2020 | * Highlighted some key phrases that emphasize the design provided is intentionally incomplete to encourage you to develop the necessary problem-solving/programming skills to solve problems on your own. * Added some example code for Distribution.getEvent() that tests the result of RNG() and ensures a single return value (rather than an ndarray). * Removed the print statements in the Lambda expression example. Your code should not have print statements in it, but some of you were basing your code very literally on the example code. Hopefully, by now, most of you realize that the type of code we’re writing here should not have print statements throughout the code (which makes for messy execution and can considerably slow execution; in general I/O operations are considerably slower than calculations). * Added an explicit statement that you may have to add private attributes in order to implement some properties (whereas others may be derived properties and not require private attributes). * Renamed Queue to SimQueue so that the class description matches the class diagram. * Added a discussion of using enums (i.e. enumerations). * Added isValid() method to SimulationStage class. |
| 3/20/2020 | * Enhanced Experience class description   + Corrected a few minor typos in the Customer class description |
| 3/17/2020 | * SourcePopulation   + Modified addCustomerDestination return type (True if CustomerDestination was added, False otherwise)   + Added additional info/hints for the methods * Customer   Corrected typo in \_\_init\_\_ argument name |
| 3/16/2020 | * SimulationStage   + Renamed method argument *time* to *simtime* * CustomerDestination   + Renamed method argument *time* to *simtime*   + Reordered acceptArrival arguments   + Added boolean return (always false for a SimulationStage, but reflecting acceptance status for Queues and SystemExits) * Queue   + Renamed class to SimQueue to avoid conflict with \_queue.Queue that prevented interactive debugging (ran ok without the debugger)   + Renamed constructor argument from *destinationAssignFunction* to *assignDestination* * SourcePopulation   + Added isValid()   + Changed destId (argument to removeCustomerDestination) from *character* to *string* |
| 3/15/2020 | * Customer   + Added note clarifying getExperiences() return value   + Changed method argument name *time* to *simtime* |
| 3/14/2020 | * Distribution   + Added is\_valid method – boolean function returns True if RNG is a valid function that requires 0 arguments, False otherwise.   + Renamed the constructor argument to dist\_spec to more accurately reflect the functionality as specified in the description (because we will be programmatically creating the lambda expressions from the distribution specific in the constructor).   + Added hint to avoid problems with unittest evaluations. * Experience   + Added return type to makeRow()   + Added specification of property return value when an attribute does not yet exist (i.e. when systemTime is requested before logServiceCompletion has been called). * Added example of creating a single row Pandas DataFrame for use in makeRow. |

One of Python’s key selling points is the ease with which we can perform numerical analyses and manipulate data using NumPy and Pandas. This makes Python well-suited for building simulations. For this term project, we are going to simulate a queuing network, like the TSA operation at PDX. This experience will serve you well as *so many* (almost all, really) real world systems can be simulated as queuing networks.

## Implementation Approach

To build our simulation, we are going to use Python’s object-oriented capabilities, NumPy, and Pandas to develop the classes shown in Figure 1. There will be four separate assignment submissions for this project:

* Part 1: implement the Distribution, Experience, Customer, SimulationStage (supplied to you), and SourcePopulation, classes
* Part 2: implement the CustomerDestination (supplied to you), SystemExit, ServerEvent, ServerState, and Server classes
* Part 3: implement the QueueEvent, Queue, Simulation, and SimulationAnalysis classes
* Part 4: implement a simulation of PDX’s TSA operation.

Because each of these submissions builds on the work of the previous submission, you cannot afford to run behind on any submissions (so please check the course calendar for due dates), *and* you must ensure that your classes work properly for each submission (i.e. if they didn’t work in a previous submission, you must correct them for the next one). Otherwise, you will be trying to play catch up during the entire semester.

You will work with one partner for this term project (we’ll deal with an odd number of students if we have to). You must do the following to complete this project:

1. Carefully understand the *business* problem and the purpose of the simulation. I have placed a chapter on queuing theory on eReserves (you can access through the libraries course site). The material in this chapter may be useful for 1) general understanding of queuing systems, 2) facilitating your own testing by giving you the means to calculate *expected* results for a particular queue (e.g. Poisson arrivals, exponentially distributed service times, two servers, etc.). However, there are severe limitations in the results that can be determined using such standard queuing models (thus the reason for simulating queuing systems), so there are *many* queue configurations for which you will not be able to find a standard result.
2. Understand the nature of the simulation and how the components must work
3. Develop a project plan and assign tasks to project team members. Your project plan should cover all of the submissions and should include (these are high-level groups of tasks, your actual plan should have more than six tasks):
   1. Design
   2. Programming
   3. Testing
   4. Running the full simulations
   5. Analyzing simulation results
   6. Preparing your analysis.

Be sure to submit both your Python code and the required write up. You will not receive full credit for just one or the other.

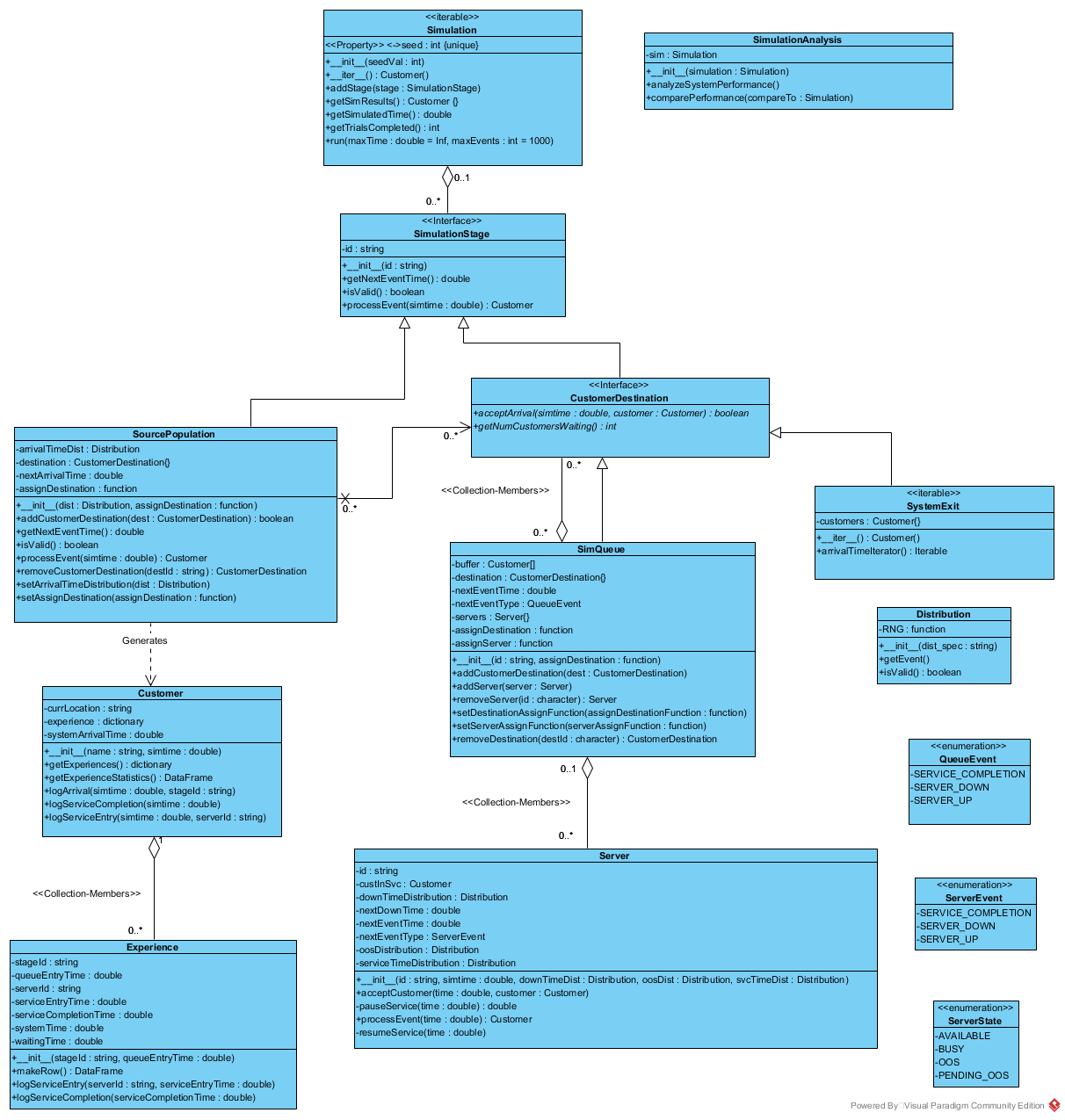


Figure 1 - Queue Simulation Class Diagram

The following provides an overview for each of the classes (*partial* design) in Figure 1. The class diagram in Figure 2 should be considered to be a *partial* or *incomplete* class diagram; the intent of the diagram is to provide you with a *big picture* for the simulation project. You will have to fill in the omitted details in order to implement the simulation.

I have intentionally omitted the property specifications in the class specifications in this diagram in order to, 1) hopefully, paint a clearer picture of the classes/objects and relationships, 2) encourage your development in thinking about the necessary elements for your classes. Adding all of the properties would add detail/noise to the diagram, potentially impacting your ability to get the big picture negatively. Additionally, if I *completely* specify the design for you, then I do all of the problem solving, which is the most important skill you must develop to be an effective simulation model. I have, however, listed property information in the descriptions below.

Keep the following in mind about defining properties for your classes:

* The properties described in *this* document should be considered as required elements. In other words, the automated testing performed on your classes may verify the presence and correct function of these specified properties. To implement the properties, you may have to add corresponding private attributes if the property does not return a *derived* value
* You may add any additional properties you wish that you 1) believe will facilitate your testing, or 2) make logical sense from a usage perspective
* Your use of properties should be consistent with our basic principles of encapsulation and representation:
  + Private vs public
  + Derived vs stored
  + Read only vs read/write.

Additionally, you should provide, at a minimum, the following overloaded methods for *all* classes:

* \_\_repr\_\_
* \_\_str\_\_

Additional detail required to implement the classes will be discussed in class.

## Customer

Every queue has *customers*. They may be people, cars, packages, or digital packets, but there are customers. This class will encapsulate all of the data about a single customer as it progresses through the system. Customer instances will be passed from SimulationStage to SimulationStage just as they would move through a real world system.

A customer object will maintain a dictionary of Experience instances. Each Experience instance will contain the information for the Customer’s experience in a single SimulationStage (e.g. in a Queue), including waiting time and total time spent in the queue. Please see the Experience class description (below) for additional information.

### Properties:

* arrivalTime – getter
* name : string - getter/setter
* totalSystemTime – getter
* totalWaitTime – getter.

Each Customer object maintains an *experience* list (implemented using a Python dictionary). Each Experience in the list will be indexed by the *id* of the stage it represents. The Customer will log every event in its journey through the system; these records will store relevant information for the stage (by stage type):

* SourcePopulation[[1]](#footnote-1)
  + Arrival time – there is only one time associated with the source population, so this can be maintained as a single value by the customer
* Queue (the following data for each queue in the customer’s journey)
  + Queue id (queueId)
  + Queue entry time (i.e. arrival time) (queueEntryTime)
  + Service entry time (serviceEntryTime)
  + Server (serverId)
  + Service completion time (serviceCompletionTime)
  + Waiting time (i.e. time spent waiting in *that* queue, Service entry time – Arrival time) (waitTime)
  + System time (i.e. entire time spent in *that* queue, Service completion time – Arrival time) (systemTime)

These data will be used to calculate queue performance and customer experience metrics in the simulation analysis. The getExperienceStatistics() method should construct a Pandas DataFrame with the columns above (described under Queue), using the camel case names specified in parentheses for each attribute. The value of the arrival time associated with the source population and the arrival time in the first stage *should be* the same. Therefore, the arrival time associated with the SourcePopulation does not need to appear in customer experience statistics (which are by queue) but can be used to test the validity/consistency of the data if desired.

Note: getExperiences() should return a *copy* of the Experience dictionary, so that the Customer object is not giving access to its *mutable* experiences dictionary.

## CustomerDestination

When service is completed for a customer, the customer must leave the queue for its next destination. This destination may be another Queue or a SystemExit. CustomerDestination represents the *public interface* that must be implemented by both Queue and SystemExit.

In Java, this would be a real interface, but Python does not provide for interfaces. However, you will implement your Queue and SystemExit classes so that the functions defined by CustomerDestination are implemented exactly the same way in both Queue and SystemExit. The following describes the return behavior of any abstract methods specific to this class since Python doesn’t support true abstraction (there *is* a way to introduce abstract classes and methods in Python, but it involves using meta classes, so we won’t go there…):

* acceptArrival – performs no actions, returns None
* getNumCustomersWaiting – returns math.inf (indicating that a customer cannot enter that destination)

## Distribution

NumPy’s *random* module provides functions for a large number of distributions as part of its base implementation. However, each of these functions requires its own set of parameters (differing in number and order of specification). This variety makes creating a *generic* interface difficult.

The Distribution class will take care of this for our simulation. The class will accept a *distribution specification* (e.g. normal(100, 20) or triangular(10, 20, 40)) as an argument during construction. The distribution specification you provide will be a numpy.random function call that will generate *a single random value* from a specified distribution. Because *all of the arguments required by the NumPy functions will be incorporated in this funct*ion, the simulation will be able to request random event occurrences without any knowledge of the underlying distribution. We discussed this technique in class on 3/12 (example in the class work .py from 3/12), and it is shown below the property descriptions.

Properties:

* RNG (i.e. random number generator): function – getter/setter

Hint: in getEvent, be sure your return value is a float, not an ndarray of floats. You may have to use self.\_RNG()[0] instead of just self.\_RNG(), otherwise. The unittest.assertAlmostEqual does not know how to round an ndarray, so all of the comparisons will fail. The best approach would be to test the result of the RNG() call to see if it’s an ndarray and return the correct result. For example:

rv = self.\_RNG()  
 **if** isinstance(rv, np.ndarray):  
 **return** rv[0]  
 **else**:  
 **return** rv

You may want to use the callable(*arg*) built-in function (returns True if *arg* is callable, False otherwise) to determine if your lambda expression can be called (otherwise it was constructed incorrectly) and the try/except structure to test whether your assembled lambda expression will actually execute:

**try**:

# code to execute. All code in this block will execute as long as there are no errors

**except** (NameError, SyntaxError):

*# This code will execute if there are any Name (i.e. object not found) or syntax (i.e. you have an invalid statement) errors*

Finally, you *obj* is a callable object (i.e. a function), then you can query information about its arguments using the following:

* Required argument count: obj.\_\_code\_\_.co\_argcount
* Required argument names: obj.\_\_code\_\_.co\_varnames

Be careful when confirming the argument count for your generating functions. If the function you are passing is an instance method of a class, the argcount and varnames will have one *extra* argument, reflecting that *self* is always the first argument to an instance method. *Your code should be able to correctly validate the number of arguments for both standalone functions and instance methods.*

### Example of using lambda expressions

Lambda expressions are Pythons version of anonymous functions (i.e. small, unnamed functions). Lambda expressions are an appropriate method for our random number generating functions. In our case, we will dynamically create the lambda expression from a *distribution specification* (e.g. normal(100, 30) or triangular(10, 20, 40)). In practice, the distribution specifications will be a valid numpy.random function call, simplifying the work required for creating the random number generating function.

*#%%  
# one reason the lambda expression is better for our purpose is that  
# we can store the "distribution generation specification" as a string.  
# for example, for our triangular distribution with min 10, mode 20, max 40:*dist = **"triangular(10, 20, 40)"***# by default, numpy random functions will generate a single value. As a result  
# the above doesn't really even look like "code". This would be a familiar looking  
# specification for anyone that understands probability distributions  
  
#%%  
# now we can turn this "distribution specification" into a random number  
# generating function programatically (the call to strip trims off any  
# whitespace characters at the beginning and end of the dist string:*funcstr = **"lambda : np.random."** + dist.strip()  
RNG = eval(funcstr)  
  
  
*#%%  
# we COULD even do this in one statement*RNG = eval( **"lambda : np.random."** + dist.strip())

## Experience

This is a *helper* class to facilitate the Customer’s tracking its experience through a single CustomerDestination (basically Queues). By creating a separate class to encapsulate the data about a stage experience, the *Customer* can easily track its experience through any number of stages *by creating, and storing, an Experience instance for each stage it enters*.

Experience objects do not need to be aware of any other objects in the system, nor do they need to *create* any data (other than calculating two attributes from data they already have). Consequently, writing and testing your Experience class requires none of the other classes.

The Experience class is fairly straightforward:

1. \_\_init\_\_ - when constructed, the Experience instance’s constructor accepts stageId and and queueEntryTime as arguments, which it will store as attributes
2. logServiceEntry – when called:
   1. this method will accept serverId and serviceEntryTime as arguments, which it will store as attributes
   2. Because the Customer has entered service, the Experience object will calculate and store waitingTime = serviceEntryTime – queueEntryTime
3. logServiceCompletion – when called:
   1. this method will accept serviceCompletionTime as an argument, which it will store as an attribute
   2. Because the Customer has completed service, the Experience object will calculate and store systemTime = serviceCompletionTime – queueEntryTime
4. makeRow – this is probably the most challenging method (but just because you’re having to use Pandas, which is new), rather than that it is difficult. When called, makeRow assembles and returns a one-row Pandas DataFrame, containing one value for each of the Experience’s attributes. This will facilitate the Customer object’s construction of a DataFrame containing all of its experiences. See the *DataFrames* discussion below for information on how to create the DataFrame.

How will this work in the simulation? While this isn’t strictly necessary for writing/testing the Experience class, having some context will allow you to better understand the Experience class itself.

1. SourcePopulation will determine *when* it is time for a Customer to arrive at the system.
2. When it is time for the Customer to arrive, SourcePopulation’s processEvent method will:
   1. Create a Customer instance (representing the new Customer)
   2. Assign the Customer to a destination (i.e. a SimQueue) by calling the destination’s acceptArrival method
3. When the SimQueue accepts the Customer it will call the Customer’s logArrival method, so that the customer will know which queue (i.e. know the stageId) and the entry time
4. When it is time for the Customer to enter service, the SimQueue will ask a Server to begin service, i.e. call Server’s acceptCustomer method, which will call the Customer’s logServiceEntry method, informing the customer of the Server’s id and the serviceEntryTime
5. When the Server completes the service, it will inform the Customer that service is complete by calling the Customer’s logServiceCompletion method, passing the time at which service was completed
6. After service is complete, the SimQueue will then dispatch the Customer to its next destination, either another SimQueue or a SystemExit.

At each stage of the Customer’s journey, the Customer object will record information about that stage in the Experience object created for that stage (one Experience instance per SimulationStage through which the customer passes). Specifically, in steps 3, 4, and 5 above, the Customer object will:

1. Instantiate a new Experience object (which will log the stageId and queueEntryTime)
2. Call the Experience object’s logServiceEntry to record the serviceEntryTime and serverId
3. Call the Experience object’s logServiceCompletion to record the serviceCompletionTime.

In this way, the Customer offloads (see, even objects *outsource*) some of the detailed work to the Experience class, allowing it to focus on the big picture and efficiently manage the data about multiple Experiences.

We will be looking at *interaction* diagrams shortly, which will help us visually describe/present the collaborations between objects.

### Properties

All attributes should have getters. If an attribute has not been defined/calculated at the time of access, return math.nan.

### DataFrames

The following code (assuming you imported Pandas as pd) creates a Pandas DataFrame with four columns and one row:

vars = [**'col 1 value'**, **'col 2 value'**, **'col 3 value'**, **'col 4 value'**]  
labels = [**'col1Title'**, **'col2Title'**, **'col3Title'**, **'col4Title'**]

In the example above, the data was already contained in a list, as were the column labels. Note that to create the DataFrame, we don’t just specify vars for the data, but [vars]. That is because a DataFrame typically has multiple rows and Pandas expects an initializing list to have one entry for every row.

Let’s say your data is stored in individual variables, say var1 through var4, and you have your column labels in a list. You would then use the following to create the single row DataFrame:

df = pd.DataFrame([[var1, var2, var3, var4]], columns=labels)

In this case, we initialized the DataFrame with a *list within a list*. The inner list is created from the variables var1-var4. The outer list consists of one entry (i.e. one row), which contains our variables.

## SimQueue

SimQueue is the star of the simulation. In real world systems, nearly *everything* can be modeled as a queue which means the SimQueue class is important. It also means that it will be one of the most important (if not *the* most important) classes in the simulation implementation.

As they are CustomerDestinations, SimQueue objects will have individual ids (provided at construction) and will allow for multiple servers. A destination assignment function will be supplied at construction. This function will accept a dictionary of destinations as an argument and will be used by the SimQueue to assign a customer to a server when multiple servers are available.

SimQueue objects will also contain zero to many Server instances (maintained in a dictionary). Servers will be added to a SimQueue instance using the addServer method. This method should ensure that the passed object is, in fact, a Server instance.

When one or more Server instances is available with one or more customers waiting in the buffer, the SimQueue instance will use its assignServer function (which accepts a dictionary of servers as an argument) to determine which Server object should accept the next waiting customer. *This functionality is referred to as advancing customers to service.*

SimQueue objects will maintain a buffer for holding *waiting* customers, and should be able to accommodate any number of waiting customers. SimQueue objects will also maintain a dictionary of destinations. The destination assignment function will determine to which of the destinations a customer will be forwarded after service is completed for the customer. When *acceptArrival* is called, the SimQueue will add the Customer to its waiting buffer (assuming the SimQueue is valid and the passed object is actually a Customer instance). Before returning, acceptArrival will determine if there are waiting customers (which there now will be at least 1) and available servers, advancing as many Customers into service as can be accommodated by the available servers).

SimQueues will be able to tell the simulation (via a call to getNextEventTime), the time at which the queue’s next event will occur. Because all of a queue’s events are really Server events, SimQueue will do this by *polling* all of its servers for their next event times. The minimum of these will be the time at which the SimQueue’s next event will occur.

SimQueue’s processEvent will be called by the simulation when the next event times indicate that the queue’s event is next. Because all of SimQueue’s events are really Server events, SimQueue will do this by determining *which* of its servers is supposed to process its event next and then it will ask *that* server to process its event (by calling the server’s processEvent method). A SimQueue instance doesn’t really care *which* type of Server event is taking place (as that’s the *server’s* job to manage). However, if the Server’s event was a service completion, then the Server will return the Customer for whom it completed service from the call to processEvent. Therefore, if SimQueue receives a Customer instance back from processEvent, it must determine which CustomerDestination to which the Customer will be dispatched and call that destination’s acceptArrival method (similar to what SourcePopulation does when a customer arrives).

The final action for processEvent is to determine whether any waiting Customer needs to advance to service. If there are Customers waiting and one or more Servers available, then as many waiting Customers as there are available servers should be advanced to service, one at a time.

In order to be valid, a SimQueue instance must have at least one customer destination and at least one server.

### Properties

* assignDestination : function – getter/setter (setter must ensure argument is a function that accepts 1 argument, or 2 with the first argument being ‘self’ if the method is an instance method)
* assignServer : function – getter/setter (setter must ensure argument is a function that accepts 1 arguments, or 2 with the first argument being ‘self’ if the method is an instance method).

## QueueEvent

The Python Enum class allows for defining true enumerations which are useful for specifying a finite set of possible values for a variable (we’ll discuss enumerations in class). The QueueEvent enum represents the valid types of events that a Queue must track:

* SERVICE\_COMPLETION
* SERVER\_DOWN (i.e. exit/pause service)
* SERVER\_UP (i.e. begin/resume service).

## Server

A Server object has a unique identifier. Unlike the standard queuing models we study in Foundations of Operations Management (e.g. M/M/1, M/M/c, etc.), a real-world queue’s servers all behave differently. By representing servers with individual Server objects, each with its own service time distribution, our simulation will be able to accommodate this variable behavior.

A Server object must know which Customer it serves (custInSvc). As with real-world servers, our Server objects will allow for going out of and returning to service randomly (e.g. machine breakdowns, bio breaks, etc.). Therefore, each server will have Distribution instances for generating service times (serviceTimeDistribution), the time until the next service outage (downTimeDistribution), and the duration of a service outage (oosDistribution).

Server objects must also know when their next event will occur nextEventTime and the type of the event nextEventType. Server state is the most complex of any class in this system. A Server object can be in any of the following states:

* Available
* Busy
* Pending out of service
* Out of service.

The allowable state transitions are described by the state machine diagram in Figure 2. The diagram tells us that only two transitions are possible when a Server is available: 1) Busy, or 2) out of service. We’ll discuss of the state machine diagrams more in class.

The pauseService method will be used to manually set the time at which a server will temporarily leave service. Similarly, resumeService can be used to manually set time at which the server will return to service.

### Properties

* id : string – getter
* custInSvc : string - getter
* isAvailable : boolean – getter
* isBusy : boolean – getter
* lastAvailableTime : double – getter
* nextEventTime : double – getter
* nextEventType : ServerEvent - getter
* status (derived) : ServerState
* serviceTimeDistribution : Distribution – setter
* downTimeDistribution : Distribution – setter
* oosDistribution : Distribution - setter

## ServerEvent

Similar to QueueEvent, ServerEvent is an enumeration that represents the set of valid types of server events:

* SERVICE\_COMPLETION
* SERVER\_DOWN (i.e. exit/pause service)
* SERVER\_UP (i.e. begin/resume service).

## ServerState

An enumeration that defines the valid states in which a Server can be:

* AVAILABLE
* BUSY
* OOS (i.e. out-of-service)
* PENDING\_OOS (i.e. scheduled to leave service as soon as service for the current customer is complete).

## Simulation

A Simulation instance will run a simulation by coordinating the actions of other objects (e.g. SourcePopulations, Queues, etc.). After being instantiated, one or more SourcePopulation, Queue, and SystemExit instances will be *added* to the Simulation instance (using the addStage method). In this way, Simulations can be dynamically configured so that this simulation object architecture can be used to run many different simulations. To add a stage, the SimulationStage object (e.g. SourcePopulation, SimQueue, or SystemExit) must already exist or be dynamically constructed and passed in the call to addStage).

At the end of the simulation, a Simulation’s SystemExit instance(s) will contain all customer objects created during the simulation. As such, a Simulation instance represents a nice neat package of all data generated by the simulation. Similar to the SystemExit, Simulation will be implemented as an iterable. This will allow access to the customers within the simulation, without having to provide access to the underlying data structures in which they are stored. Unlike the SystemExit (which must deal with the contents of only one dictionary), a Simulation may contain multiple SystemExits (it must contain at least one). Consequently, the Simulation iterable must be able to return the Customers held in *all* system exits (i.e. it will have to loop over all of its SimulationStage instances, and for each stage, iterate over its Customers if it is a SystemExit).

The constructor will take the a seed value as an argument (default value of None) which it will use to set the NumPy random number seed (np.random.seed(seedval)). Once the Simulation instance is constructed, the *setup* code will construct and add SimulationStages, in this way dynamically configuring the simulation to consist of a network of source populations, queues, and system exits. Pretty cool!

The getSimResults() should construct a and return consolidated dictionary of Customers (i.e. it should consolidate the Customers from all SystemExits into a single dictionary); getSimResults returns the actual customer objects in a dictionary.

getSimulatedTime returns the current value of the simulation clock (i.e. simtime).

getTrialsCompleted returns the total number of simulation loops completed.

The *run* method is the *main* method of the simulation:

* Implements the simulation loop
* Each time through the loop, an event will be processed by *one* SimulationStage
  + The Simulation will determine which SimulationStage is due to process an event (i.e. which one has the earliest nextEventTime)
  + Request that the SimulationStage process its event
  + *Advance* the simulation clock (i.e. simtime); simtime will become equal to the nextEventTime for whichever SimulationStage processed its event.

That’s basically it. All of our other classes are doing the bulk of the work. Simulation’s job is simply to coordinate the work performed by the other classes.

### Properties

* seed : int – getter/setter
* numStages : int - getter

## SimulationAnalysis

It’s a good idea to separate the analysis from the running of the simulation. We’ll create a simulation analysis class that will perform the performance analyses on a simulation object as well as compare the performance of two different simulations. On construction, SimulationAnalysis will be passed a Simulation instance. If the Simulation instance has completed a *run*, SimulationAnalysis will use the existing results contained by the Simulation. If the Simulation instance does not have any simulated results, the SimulationAnalysis constructor will call the Simulation’s run method.

Once the simulation is complete, we have all of the data/detailed metrics we need to analyze the system. To make it easy, we are going to calculate only the following metrics:

* # of customers who completed service
* Waiting time metrics
  + Average (AvgWaitTime)
  + Maximum (MaxWaitTime)
  + 90% confidence (90%WaitTime)
* System time metrics
  + Average (AvgSysTime)
  + Maximum (MaxSysTime)
  + 90% confidence (90%SysTime)).

In general, I would write methods that would calculate the confidence value for a level of confidence passed as an argument. This function could then be used to calculate the 90% confidence requested in the output data structure. To find the confidence values, you can use NumPy’s percentile function or Pandas’ DataFrame.quantile function with interpolation=’higher’.

analyzeSystemPerformance should return a dictionary containing the metrics above (the keys are indicated in parentheses next to the metrics above).

compareSystemPerformance should accept a Simulation as an argument and compare this *compareTo* simulation’s performance to the simulation used to construct the SimulationAnalysis object. If the *compareTo* Simulation represents a completed simulation, compareSystemPerformance will utilize the existing simulation results. If not, then compareSystemPerformance will call the compareTo simulation’s *run* method.

Again, to keep it simple, we will calculate only the differences between the two simulation’s metrics (returned by analyzeSystemPerformance). When calculating the differences, be sure to subtract the compareTo’s metric from the \_sim metric. compareSystemPerformance should return a dictionary containing the differences for all six of analyzeSystemPerformance’s metrics (using the same keys as in the analyzeSystemPerformance dictionary).

If we were really interested in analyzing the queuing network, we’d be interested in analyzing much more, which would be appropriate for a course on operations or queuing but is less relevant for a general course on simulation.

### Properties

* sim : Simulation – getter

## SimulationStage

Each SourcePopulation, Queue, and SystemExit represents a stage in a simulation. SimulationStage represents the public interface that SourcePopulations, Queues, and SystemExits must provide.

The following describes the return behavior for the two abstract functions:

* getNextEventTime – performs no action, returns math.inf
* processEvent – performs no action, returns None.

### Properties

* id : string – getter.

*SimulationStage will be provided to you.*

## SourcePopulation

Every queue has one or more source populations which represent sets of potential customers for the queue. We will use SourcePopulation to represent the potential customers for our queuing system; SourcePopulation instances will be responsible for generating arriving Customer instances.

A SourcePopulation instance will take a Distribution for generating customer arrival times on construction. SourcePopulation instances will also keep track of the time at which the next arrival will occur, as well as a set of possible destinations for the arrivals. A SourcePopulation instance must allow for multiple potential destinations; a SourcePopulation might have multiple destinations when its arrivals are segregated into different queues on arrival (think the TSA pre-check/General boarding lines at the airport).

The processEvent() method is defined as returning a Customer object. This is to allow the SourcePopulation to differentiate between customers who arrived at and entered the system vs. those that arrived at the system and balked (i.e. decided not to enter). If a customer enters the system, processEvent() should return None. If the customer balks, processEvent should return the customer object. For *this* project, we will assume that customers do not balk, so processEvent will always return None.

To process an event, the SourcePopulation should:

1. Ensure that it’s actually time for a customer to arrive. If the passed simtime < nextArrivalTime, the SourcePopulation should not process its event
2. Generate a new Customer instance:   
   construct the Customer’s name as “*id-#”* where *id* is the SourcePopulation’s id (inherited from SimulationStage) and # is sequentially incremented number within each SourcePopulation (i.e. the first customer would have name “*id*-1”, the second “*id-*2”, and so forth
3. Use the assignDestination function to determine which SimulationStage should receive the new Customer
4. Ask the SimulationStage to accept the new arrival (i.e. call acceptArrival).

To be valid, a SourcePopulation must have an arrivalTimeDistribution (i.e. a Distribution instance) and at least one CustomerDestination. The setter methods, setArrivalDistribution and setAssignDestination should ensure that the Distribution and assignment functions meet the specification (i.e. an actual Distribution instance, a function that accepts a single argument).

If a SourcePopulation is invalid:

* getNextEventTime should return math.nan
* processEvent should return None. We haven’t covered exceptions yet, or we’d have it raise an exception. Be sure to check that your SourcePopulation instance is valid after you instantiate it.

Hint: look at the pop method of the Python dictionary for removing CustomerDestinations from the destination dictionary.

## SystemExit

Eventually, all customers will exit the *system*. Typically, Customers will exit after all service has been completed, prematurely before beginning service (referred to as reneging), or they may leave prior to entering the system (balking). In *this* simulation, we will allow only for customers entering and completing service in the system; the simulation infrastructure may be readily extended (in a number of different ways) to account for balking and reneging behaviors.

We will create SystemExit to be an Iterable. In other words, we will overload the \_\_iter\_\_ method which will enable a SystemExit instance (e.g. seobj) to be used in a for loop to access Customer instances one at a time (in the order they arrived in the SystemExit):

for cust in seobj:  
 # do stuff with cust

The arrivalTimeIterator() method will return an iterable that will allow sequentially accessing Customer objects in the order in which they arrived at the system (rather than the order in which they arrived at the system exit).

These *iterables* will allow the SimulationAnalysis to access the Customers at a SystemExit without requiring access to the dictionary containing those customers. An alternate approach would be for the iterator to return the customer experience DataFrame. However, for *this assignment* we will primarily be looking at summary metrics such as total wait time and system time; having access to Customer properties that provide the summary metrics will be sufficient and, if desired, the Customer instance can be asked for its experience DataFrame.

# TSA Security Simulation

Given the challenges this semester, you will construct an *abbreviated* version of the TSA Security operation (i.e. queuing network) at the airport:

* Two source populations (TSA pre-check and regular)
* Seven (identical) single-server queues
* Two system exits.

Figure 2 shows the configuration of the system. Here are additional details:

* The random seed should be set to 100
* The TSA pre-check source population has an average inter-arrival time of 30 seconds
* The regular source population has an average inter-arrival time of 10 seconds
* *All* servers have:
  + down time ~ triangular distribution with min 14,400 secs, max 18,000 secs, and mode 14,400 seconds
  + out of service time ~ triangular distribution with min 300 secs, max 1,200 secs, and mode 600 secs
  + service time distribution ~ mean service time 30 seconds

Submission (Keep the write-up short; you only need short answers/paragraphs for the questions. This isn’t a dissertation):

1. Setup and run the simulation described above with maxEvents=500
2. Run your simulation analysis and print each of the metrics in the analysis, for example, in the following format:

NumCustomers =

AvgWaitTime =

MaxWaitTime =

AvgSystemTime =

MaxSystemTime =

90%WaitTime =

90%SystemTime =

1. Consider the queuing network in Figure 3. Between Figure 2 and Figure 3, which more closely resembles (remember these are *abbreviated* networks, so they won’t be true representation) the nature of the TSA screen operation at PDX? Why?
2. Compare and discuss the performance of the two networks assuming that all distributions are the same as in your first simulation; the only difference between the two simulations is the number of waiting lines. What can you conclude about these two network configurations? Does TSA have it right (i.e. have they chosen their screen queuing network to yield the shortest waiting times)?



Figure 2 - Abbreviated TSA Security Queuing Network



Figure 3 - Alternative Configuration for TSA Security Queuing Network

# Using Enums

The example below illustrates the creation and use of an enumeration (i.e. an enum). Enums are useful for ensuring that a variable can assume only the designated set of values. For example, an Enum for the days of the week should only be able to assume values for Sunday, Monday, …, Saturday. If we use a string variable, there’s no way to restrict the values to just the seven valid days. The same holds for a standard numeric.

However, when using the enum, our code can only use the seven *valid* values for a day of the week. By convention, we always use all capital letters for enum names (see example below).

**from** enum **import** Enum  
  
*# an enum represents a finite set of related values. We use enums  
# to ensure that a variable can assume only valid values.  
  
# to define a new enum, we inherit from the enum class and  
# specify values names and values for each of the possible  
# values of the enum.***class** DOW(Enum):  
 SUNDAY = 0  
 MONDAY = 1  
 TUESDAY = 2  
 WEDNESDAY = 3  
 THURSDAY = 4  
 FRIDAY = 5  
 SATURDAY = 6  
  
*# the enum allows us to make assignments and do comparisons using  
# using clearly defined values, but with much greater efficiency  
# than assigning and comparing string literals*day = DOW.TUESDAY  
  
*# comparisons are much clearer than day == 2 and more efficient than  
# day = 'Tuesday'*print(day == DOW.TUESDAY)  
print(day **is** DOW.TUESDAY)  
  
*# when we print the variable, we see that it's TUESDAY rather than the  
# actual value of 2 held by the variable*print(day)  
  
*# we can access the enum value as a string (i.e. its name)*print(day.name)  
  
*# we can access the underlying numeric value that is actually stored*print(day.value)  
  
*# we can use the enum to find the name of the nth day of the week*print(DOW(0))  
print(DOW(2))  
  
*# There's a lot more that can be done with enums, but these are the  
# basics.*

# State and State Transitions

Figure 2 and Figure 3 show sample state machine and activity diagrams. The state machine diagram provides a detailed view of the states a server may have and documents the transitions between states. The sample activity diagram reflects the logic required when making a transition to the Available state.

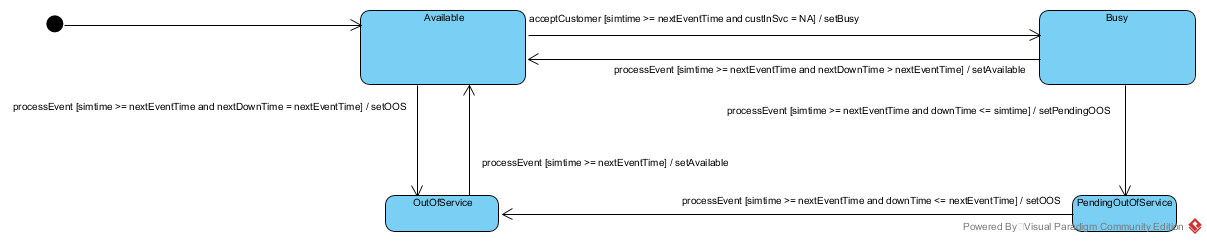


Figure 4 - Server State Machine Diagram

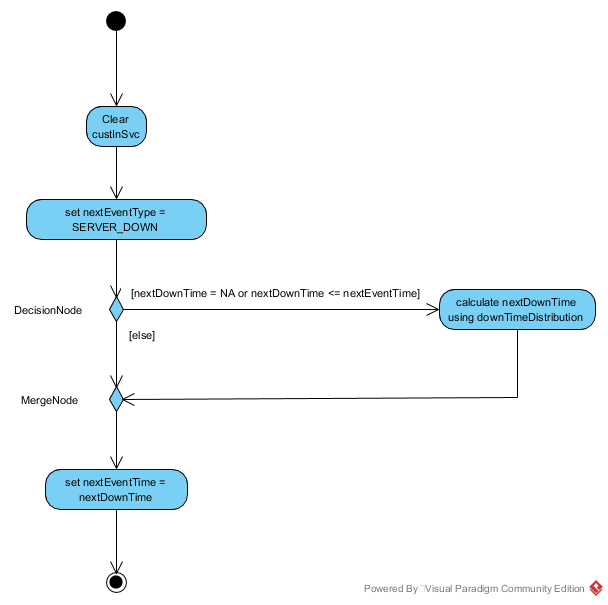


Figure 5 - Sample Activity Diagram: setAvailable

1. Because we are omitting balking in this simulation, we will not create an Experience instance for the *system* arrival. In other words, you will generate an Experience instance for every Queue a customer enters, but not for the SourcePopulation or SystemExit. [↑](#footnote-ref-1)