1. **gProject Background**
2. **Language being employed**

Due to the components of such a discrete events model, it’s obviously that an object design pattern can make the model be easily built and maintained. Nowadays, there are a bunch of Object Oriented high level languages like Python, Java, C#, etc. Based on personal experience, the language Java is adopted to build and simulate this model.

The advantage of Java is noticeable. First, it’s Object Oriented, that means the model could be built with a high reusability. Second, Java is a cross-platform language, that means the model can be done on a Unix system but run on a Windows system. Third, the garbage collection function of Java prevents the programmer from being tricking by manage the dynamic memory.

However, the disadvantage of Java is not negligible neither. The virtual machine of Java makes cross-platform possible. As a payoff, this mechanism makes the size of a Java program much larger than a C program which implements the same functions. Also, the garbage collection mechanism does not release memories no longer in use instantly. It collects and releases those memories periodically. Suppose we are running a model involves billions of data sets, this probably slows down the program and products some incorrect outcomes. Java is poor at doing statistics as well. The Random class provided by Java library generates random numbers based on a naïve algorithm, as well as there are not so many data visualization libraries for Java. The most popular library is the so called “JRI” library with the latest version has been released almost 4 years ago. This library is quite fussy, if you are coding with an IDE, then the running environment and class path have to be set up for every new class.

Nonetheless, this project requires us to simulate a M/M/2/2+5 model with some small data sets, thus Java could be adopted.

1. **Design Pattern**

A simulator contains multiple components, such as statistic counter, blocking counter, events, events generator, etc. They are some abstract patterns. Like the word “event”, in our project it refers to the Discrete Events, maybe it could be used to identify a continuous event in some other projects. Object Oriented Design patterns makes our project extendable. Based on data patterns, a systematic model development procedure is defined. [1]

1. **Observer Pattern**

The observer pattern is a software design pattern in which an object, also called “subject” maintains a list of observers, and notifies them by calling one of their “update()” method when any state changes. We have to define “subject” and observer, so that when a subject changes state, all registered observers are notified and update automatically. The responsibility of a subject is to maintain a list of observers and to notify them. The responsibility of an observer is to register itself on a subject and update itself when notified. [2]

In our project, we will focus on times: the service times an event spends in a server, the interval time between the next coming event and a new generated event. Actions of the two servers and the common queue all depend on the status of events in the servers. Once a service is done, the server should pop the event out and inform the queue to offer a new event. So, we have to make the instances of events should be observable to the servers. In our project, the servers play the role of the observers, and those events are observed.

1. **Singleton**

The singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. [3]

Why do we need a singleton here? It’s all because of time. An event could be sent to the queue, it could be rejected also, if the queue is full. In my design, an event itself does not contain a time counter. Thus, we need a static, or global, time counter to assign the initial time and the terminal time to it. A singleton defines a public static operation (getInstance()) that returns the sole instance of the class.

1. **Factory Pattern**

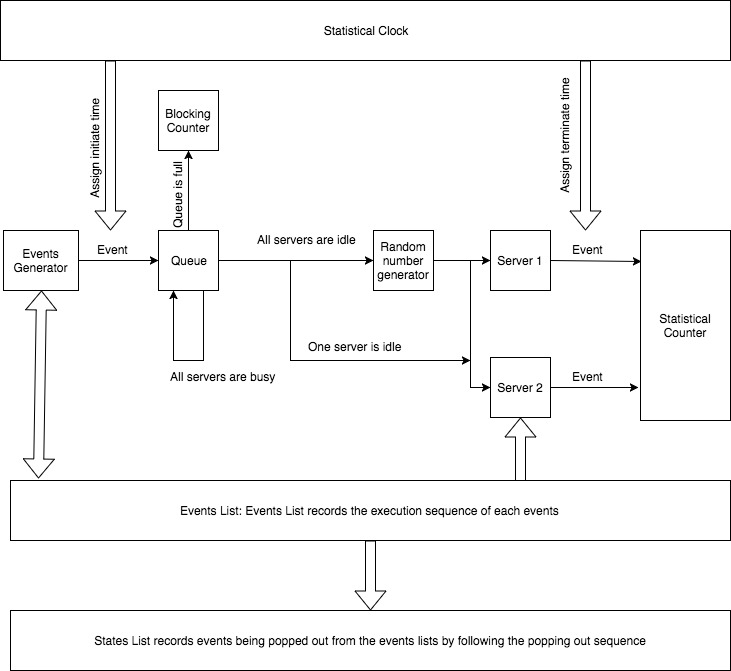
The factory method pattern uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method rather than by calling a constructor. [4]

A simulation model may have different types of events. As I mentioned about, it could be a discrete event, a continuous event. It also might be used to mimic a bus, a airplane or a patient. Factory pattern capsulate all constructor in an abstract class or an interface. Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created. [4]

In this project, only one type of event will be created. However, it’s still worth trying to implement this philosophy into designing our model.

1. **Analyze the program**
2. **Execution procedure**

The following picture illustrates the execution procedure of this program:



1. **Illustrations on components**

Generally speaking, a simulation model (discrete event simulation) should contains the following components: [5]

*System State, Simulation Clock, Event List, Statistical Counter, Initialization Routine, Timing Routine, Event Routine, Library Routine, Report Generator and Main Program. [5]*

According to the flow chart above, the function of main components and how they are implemented will be briefly introduced. Some components may not be listed above.

***Event Generator:*** In this project, the events generator produces three types of events. Two of them are generated to identify an upcoming event should be “generating a new event”, “popping an event in server”. The rest one is “Customer”. The first two are going to be sent to event list, the last one will try to enter the queue then one of the servers.

***Event List:*** A list containing the next time when each type of event will occur. Events in this list are sorted by a min-heap (priority queue).

***System State*:** The collection of state variables necessary to describe the system at a particular time. A linked list will be used to record these variables based on the FIFO mechanism. That means an event being popped out from the events list should always be attached to the tail of the linked list.

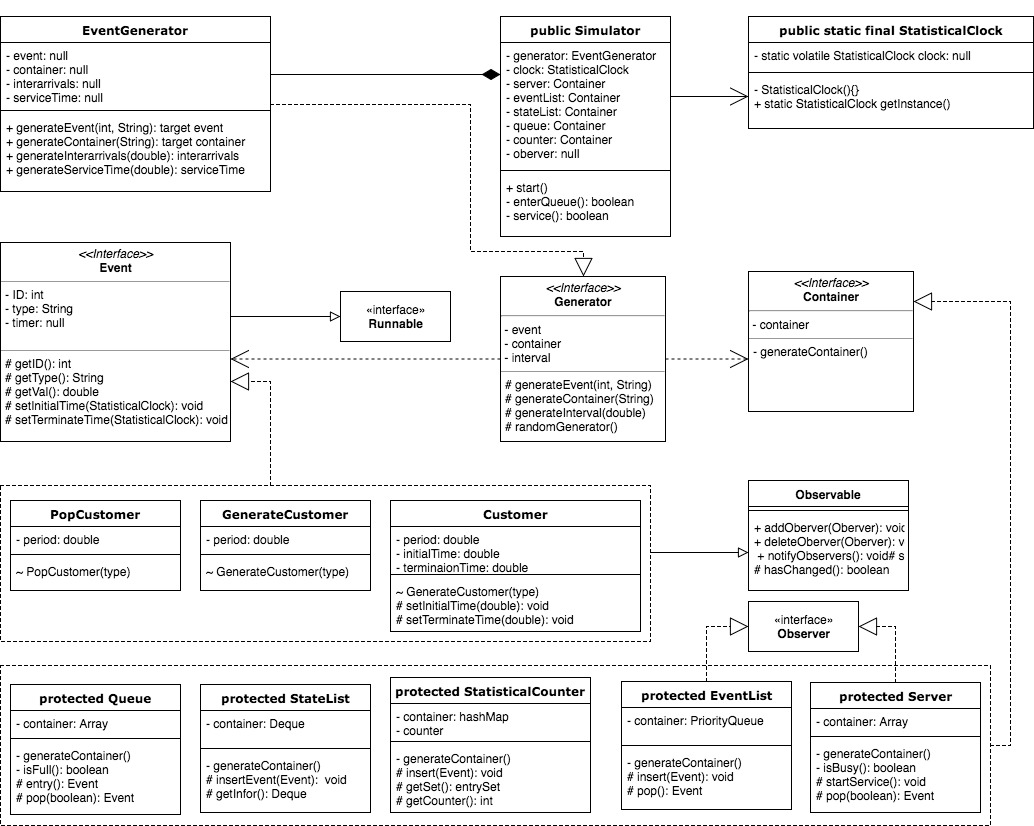
***Statistical Clock:*** A variable giving the current value of simulated time. In other words, the statistical clock gives each event a time stamp.

***Queue:*** The area where accepted event waits for service while both two services are busy. It’s implemented by initiating an array with a fixed size.

***Random Number Generator:*** This generator will generate random numbers based on employing a uniform distribution function. [5] Excepting the function for generating a random time, the generator will be used to decide an event should be processed by which server when both of them are idle.

1. **UML**

The following picture shows the UML of this program.



***Event (interface):*** Event is the soul of the whole program. It’s hard to define exactly what an event is for it’s too abstract to be described. Just image this is a clinic, a customer exists is an event, waits for service is an event, leaves the clinic for waiting too long is an event, leaves with the treatment done is an event, and even a customer itself can be treated as an event. So, any event just need to implement this interface and override particular methods or values. As soon as an event exists, it must can be run or operated independently. Thus, the event implements the Runnable interface provided by Java. Since some events need to be observed, two classes, EventList and Server, extend the Observable class provided by Java.

***Generator (interface):***As mentioned in the first chapter, the factory method pattern is implemented in this project. Here, the Generator is an abstract factory defines different getXXXEvent() for different types of events. If we are simulate a complex system which might require tons of generators for hundreds of events, the we will create some generators by implementing this interface and overriding getXXXEvent() accordingly. However, due to our project is not that complex, only one general generator will be instanced.

***Container (interface):*** No matter the queue, the servers or the lists are all containers where events are stored and recorded. We don’t need to define details, almost the same, of each container separately. Thus, put an interface contains all common features and behaviors is fine.

***Simulator (class):***The Simulator Class is the brain of the whole program. It collects state information and decides the next step to go. Such a design can low down the couplings between objects.

1. **Declarations**

\* Why are interfaces preferable?

I define three interfaces in my project. It quite obvious that I prefer interfaces to abstract classes. The reasons for this are listed below:

1. Interfaces are easier to define. An interface contains only a bunch of undefined methods or values.
2. An interface makes those classes which implementing it more flexible. For example, the EventList and Server have to extend the Observable class. If I defined the Container as a superclass, then they lose the possibility to extend the Observable class. Theoretically, the number of interfaces implemented by a class is almost unlimited but only one super class can be extended.

\* Why is a min-heap adopted to implement a EventList?

A min-heap can provide us the object which has the minimum value among objects in the heap in *O(1)* time complexity. Then it can be sorted with the next minimum value on the top in *O(logn)* time complexity.

The advantage of min-heap is obvious as described above. But, a min-heap is a discrete data structure on the logic level, in real world, it’s build up based on array, a linear data structure. With the number of objects in a heap increases, such a data structure cannot use memory efficiently.

1. **Report**
2. **Test the random number generator**

Nowadays, most high-level programming languages provide powerful random number generators. However, most computer generated random numbers use pseudorandom number generators (PRNGs), also known as deterministic random bit generator (DRBG), which are algorithms that can automatically create long runs of numbers with good random properties but eventually the sequence repeats. [6] Although sequences that are closer to truly random can be generated using hardware random number generators, pseudorandom number generators are important in practice for their speed in number generation and their reproducibility, and, the hardware random number generator is out of our scope.

Java does provide a Random Number Generator in its *Random* class. According to the docs, java.util.Random.next is implemented as follows:

synchronized protected int next(int bits) {

seed = (seed \* 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);

return (int)(seed >>> (48 - bits));

}

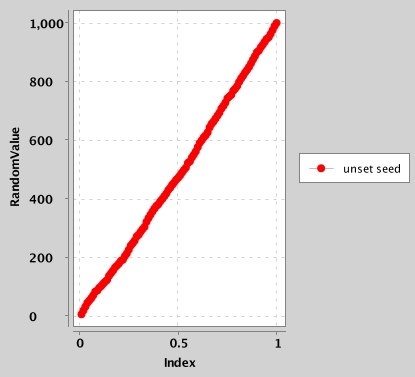
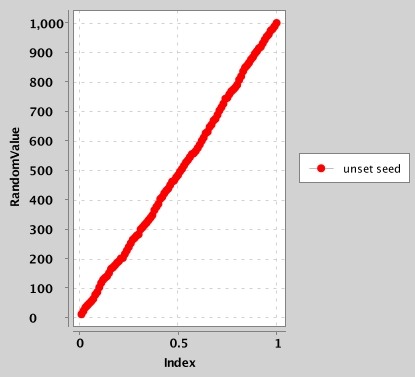
There is nothing in there that takes a variable amount of time, but that's in a big part due to the fact that it's dealing only with fixed-length numbers. So that's Java's random number generator, which isn't even a random number generator but a pseudo random number generator and not a very good one at that, as noted. [7]

As a fantastic language in statistics, R offers us a variety of solutions for random number generator. [8] Thus we decided to use a RNG provided by R in our code. Functions of R can be called in Java by importing an external library *JRI.* It is a Java/R Interface, which allows to run R inside Java applications as a single thread. Basically, it loads R dynamic library into Java and provides a Java API to R functionality.[9]

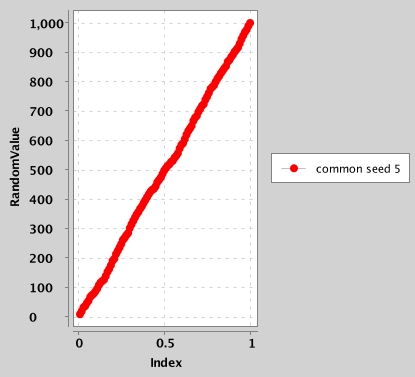
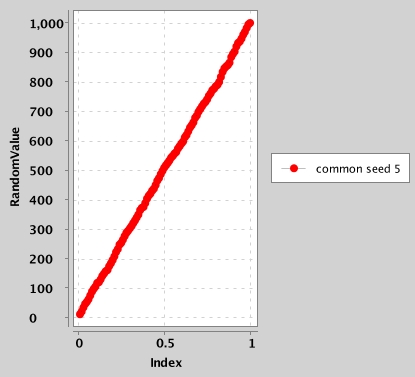
1. **Does your RNG generate random numbers?**

Yes, it works quite well. We tested our RNG with three different situations: unsetting the seed, setting a common seed for all sequences and setting an identical seed for each sequence. We generated two 1000 sized sequences for each situation.

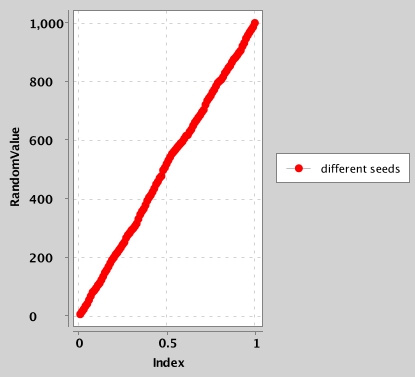
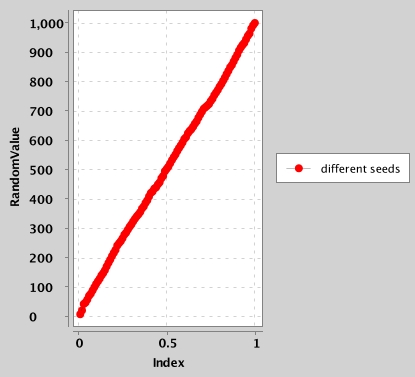
To check if it’s working. We plot out empirical CDFs for total six sequences shown as the following pic. And source code for plotting these CDFs, please refer to TestRNG.calls, see the plotting() method.



RN without setting seed



RN with common seed 5



RNs with different seeds

1. **How do you initialize the seed of your RNG?**

Seed can be initialized in three ways:

1. Default initialization. That means we do not set a seed.
2. Inputting a seed manually. This method allows the user to input a common seed for all runs or to set different seeds for each run. Due to the R language is employed, we set seed and generate random numbers all though RInitializing a seed with the following function:

 public static List<Double> getInstance(int n, boolean flag, double seed) {

    list = new ArrayList<Double>();

       if (flag == true) {

        ENGINE.eval("set.seed(" + seed + ")");

        }

        for (int i = 0; i < n; i++) {

    list.add(ENGINE.eval("runif(1)").asDouble());

    }

        return list;

    }

1. **Generate two sequences of 1000000 numbers each, for every sequence use a**

**different seed. Are these two sequences different? How do you know this?**

[1]. Franziska Kl¨ugl and Lars Karlsson, Towards Pattern-Oriented Design of Agent-based Simulation Models

[2]. <https://en.wikipedia.org/wiki/Observer_pattern>

[3]. <https://en.wikipedia.org/wiki/Singleton_pattern>

[4]. <https://en.wikipedia.org/wiki/Factory_method_pattern>

[5]. Edward Chlebus, CS555 Lecture Notes

[6]. <https://en.wikipedia.org/wiki/Random_number_generation>

[7]. <https://stackoverflow.com/questions/7291911/javas-random-number-generator-complexity-of-generating-a-number>

[8]. <https://gist.github.com/MonkmanMH/7740998>

[9]. <https://www.rforge.net/JRI/>