1. **Project 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 adoptable.

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]

Our design borrows this conception, but we did not implement the Observer pattern provided by Java. Instead of an observer, we use the TimeTask, provided by Java and used with Timer. The TimerTask is an interface which has to be implemented by some specific classes. Those classes must override the run() methods inherited from the TimerTask. Then a Timer should schedule the TimerTask with any delay or execution periods. Suppose we override the run() to let it output the famous “hello, world”, then a Timer assigned a working period for it, assume it’s 1 second. That means the TimerTask is going to output “hello, world” once a second.

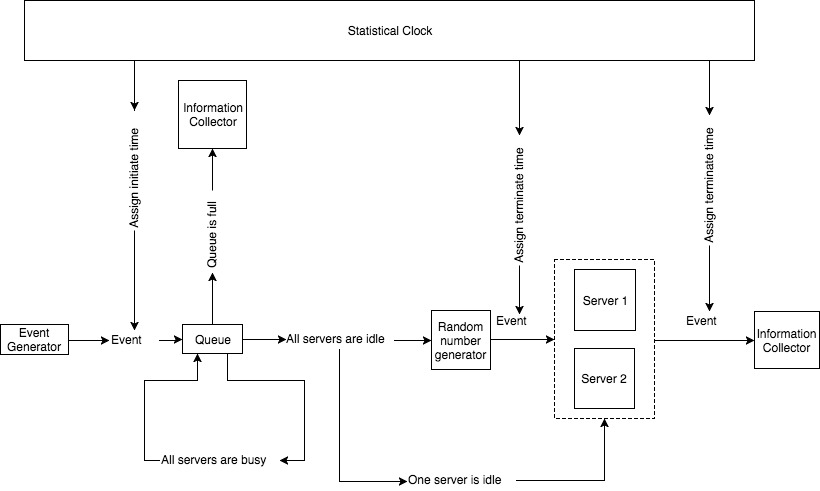
1. **Singleton**

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 the 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. **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.

***InformatinCollector:*** This is a class contains several data structures which are used to record system states in each given observation period. And it also records the execution sequence of each event.

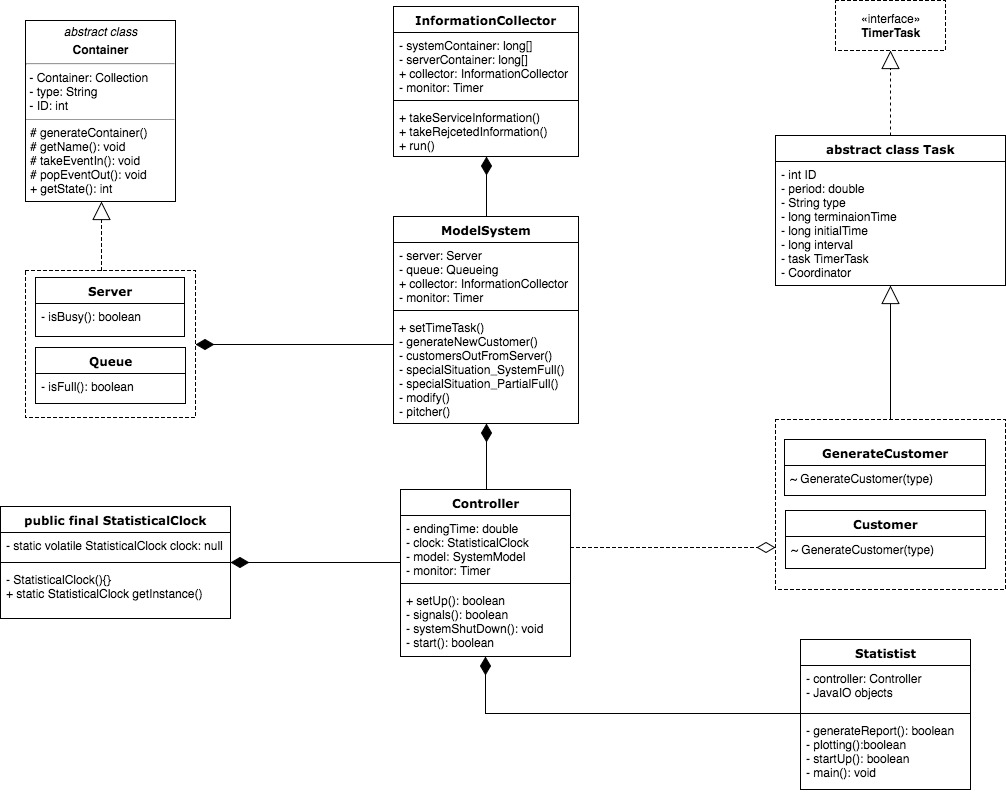
***Statistical Clock:*** A Singleton class provides the current value of simulated time. In other words, the statistical clock gives each event a time stamp.

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

***Random Number Generator:*** This generator will generate random numbers based on a uniform distribution function. [5] Except being used to generate the expected service time or the interval between two customers, the generator will also be used to decide a customer should be processed by which server when both of them are idle.

1. **UML**

The following picture shows the UML of this program.



***Task (abstract class implements TimerTask):*** Task is the soul of the whole program. It’s hard to define exactly what a task is for it’s too abstract to be described. Just image there is a clinic, a customer exists is a task, waits for service is a task, leaves the clinic for waiting too long is a task, leaves with the treatment has been done is a task, and even a customer itself can be treated as a task. As soon as an event exists, it must can be run or operated independently based on time. Thus, the Task implements the TimerTask interface provided by Java. The Controller holds a Timer for scheduling the interval time between a new generated customer and the customer will be generated next. The Server holds a Timer for scheduling a service time for a customer accepted by a server.

***GenerateCustomer (class extends Task):*** Since the Task class implements the TimerTask interface, so the GenerateCustomer class itself is also a TimerTask. Once the program starts, a GenerateCustomer object should be generated first, and a Timer hold by the Controller then assigns an expected time provided by the Random Number Generator based on the given rate. And the run() method inherits from the TimerTask interface has been overridden to inform the Controller for generating a new customer when the time reaches the expected time.

***Customer (class extends Task):*** A Customer object will not be the first task be generated when the program starts. Its only be generated when a GenerateCustomer object ran its run() method. The Customer object is also a TimerTask. When it enters a Server, the Server assigns a expected service time based on the given rate. And the run() method hold by the Customer object has been overridden to inform the server which holds the object to pop it out when the expected service time ends.

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

***Queueing (class extends Container):*** An ArrayList with fixed capacity hold by the ModelSystem class.

***Server (class extends Container):*** An integer array has size one. The Server class has a Timer. Once a customer enters a server, the server schedules a service time for the customer.

***InformationCollector(class):*** This class was designed for only one purposes: collecting the system states periodically. This class implements the TimerTask also. A Timer hold by the MoldelSystem schedules the period for the collector once the program starts.

***ModelSystem(class):*** The ModelSystem is somehow a container also. It likes a plant in which the queue and the servers are placed. This class was designed for future development: the amount of queue and server can be increased or decreased. Thus, we can simulate multiple situations with one model.

***Controller(class):*** The controller has an instanced static ModelSystem object. If any object of this project wants to call a method own by another object, the only approach is calling the ModelSystem object owned by the Controller then call the expected object own by the ModelSystem object.

All classes mentioned above do not open to public. Their access modifiers are default or protected. The only object opens to the real world is the Statist class.

***Statist(class):*** This is the only one class provides public methods to the real world. The main purpose of this class is taking data from the InfomationCollector, generating reports and plotting those data out. Maybe the plotting function and the reports generating function will be move to another class. Then the Statist will have only the main method and some other methods open to the outside for setting up some parameters, like the amounts of servers and queues, or the arriving rate and the service rate.

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));

}

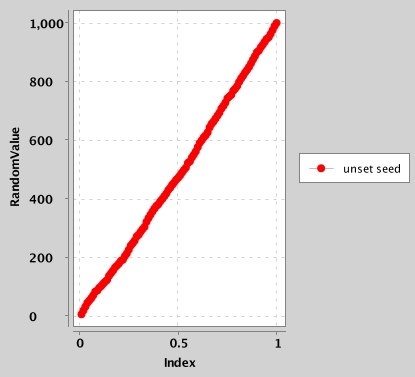
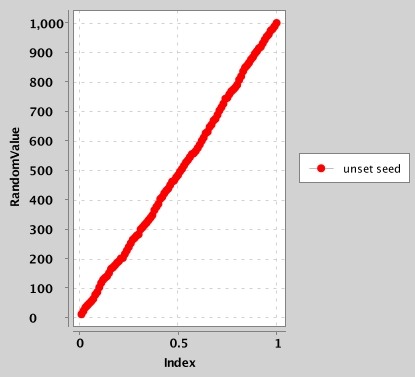
There is nothing in there that takes a variable amount of time, but that's in a big part due to the fact that it's dealing only with fixed-length numbers. So that's Java's random number generator, which isn't even a random number generator but a pseudo random number generator and not a very good one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

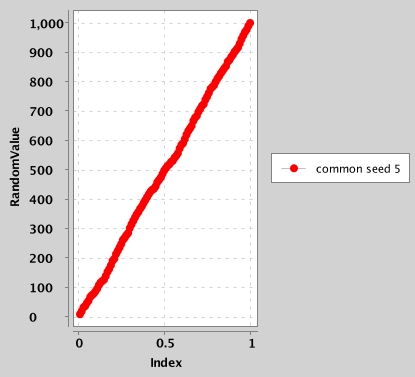
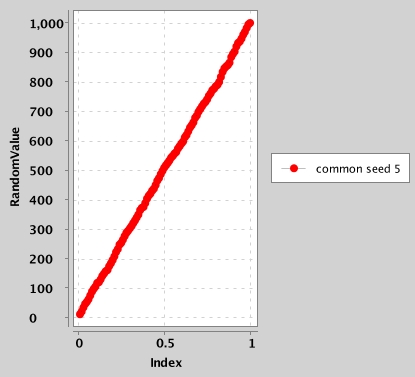
1. **Does your RNG generate random numbers?**

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.

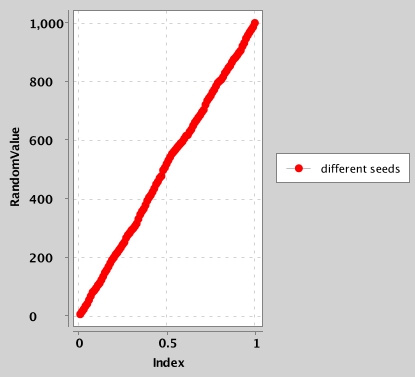
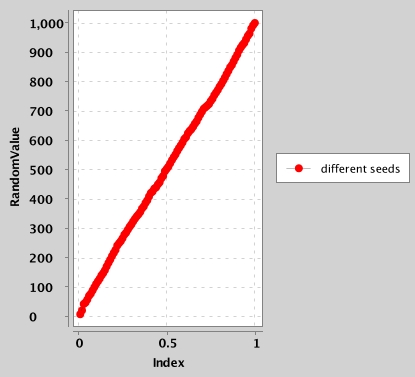
To check if it’s working. We plot out empirical CDFs for total six sequences shown as the following pic. And source code for plotting these CDFs, please refer to TestRNG.calls, see the plotting() method.



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?**

If a random number generator provided by any high-level programming language, it will give us the same result while a huge amount of random numbers is generated without resetting the seed. This is because of the so-called pseudo random number generator. The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's seed (which may include truly random values). [6]

If two random number sequences were generated with different seeds, they supposed to be different. To check if they are really identical, we can calculate the ratios of amounts of the numbers which have the same values in each sequence. Then we plot the ratios out by letting the y-axis indicates probability, the x-axis indicates those numbers. We can see if they are identical through comparing those two curves. If they are totally the same, then we got two duplicated sequences.

The most naïve approach to see if they are the same is to sort them first, and compare them one by one.

[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/>