**CIS 162 Project 5**

**GVSU Market Place Simulation**

**Project Due Dates**

* Part 1 is due at the start of lab on Tuesday, November 21, 2017. You will not turn in any code but will run the unit test (JunitTest1). Completed steps 1 - 7. (20 %)
* Part 2 and demonstration will be during lab on Tuesday, December 05, 2017. You will run Junit test (JunitTest2) and demo the GUI.
* **No late submissions are accepted for this project.**

**Before Starting the Project**

* Read chapters 8 and 10
* This is a complex project! It is important that you read this entire project description and to get started right away.

**Learning Objectives**

After completing this project you should be able to:

* *use* ArrayLists to maintain and process collections of objects
* *use* PriorityQueues to maintain and process collections of objects
* *use* arrays to maintain and process collections of objects
* *develop* an application with multiple classes

**Class Inheritance**

You need a basic understanding of class inheritance to complete this project (read 13.1 – 13.3). Inheritance is a programming strategy used when multiple classes are similar to each other. For example, trucks, cars and dune buggies all have similar characteristics shared by many vehicles. Elephants, dogs and monkeys have similar traits shared by other mammals. Instance members and methods shared by all classes are combined into a special class called the base class.

public class DerivedClass extends BaseClass{

// define instance variables shared by all derived classes

// define methods shared by all derived classes

// define virtual methods if a derived class must implement

**Computer Simulations**

Computers are often used to simulate complex scenarios such as weather patterns, nuclear reactions, climate change and potential spread of infectious diseases. Many of the simulations have the same basic design. Events are scheduled to occur at a particular time in the future. As the events occur they create additional events. This sequence continues until an ending condition of the simulation.

A list of future events is similar to a “to do” list you might create for today. The eventual sequence of tasks is not known at the start of the simulation. You may start the day with a number of tasks to complete at particular times such as go to class, meet a friend for lunch or go to work. Throughout the day, you might add unexpected tasks such as picking up your younger sister from ballet practice. This event might create an additional task such as take your sister to dinner.

**Our GVSU Market Place Simulation**

You will create an application to simulate customers arriving and departing the GVSU Market Place on a busy football Saturday. It will be used by the store manager to predict how long customers will be waiting in line and how many cashiers to schedule. Customers enter the store, browse for items, and then some customers get in line with their purchases.

After shopping, customers get in a single line and wait for their turn at one of the cashiers. Customers eventually move to the next available cashier. After the customer pays for her purchases, she leaves and the next customer in line (if any) moves to the cashier. This continues all day long until closing time. Customers in line at closing are served but no new customers can get in line. The store opens at 10:00 am and closes at 6:00 pm.

**Simulation Parameters**

The manager can manipulate three parameters to influence the simulation.

1. The *average customer arrival time* predicts how often a customer gets in line. This can be based on store history. If customers arrive every 2 minutes on average, the simulation will generate random customer arrival times above and below 2 minutes.
2. The *average service time* is how long a cashier takes to serve one customer. Service times vary based on the number of purchases and the speed of the cashier.
3. The *number of cashiers* on duty determine how many customers can be served simultaneously. The manager wants to schedule the fewest number of cashiers without causing long lines.

**Simulation Results**

The manager will see several results: 1) the number of customers served during the day, 2) the average customer wait time in line, 3) the length and time of the longest line and 4) time of the last customer departure. Wait time for a customer includes the amount of time in line but NOT the service time at the cashier. Simulations generate events randomly resulting in slightly different results each time.

**Store Events**

Event times are maintained as *a double* representing the number of minutes into the day starting at midnight. For example, thirty minutes into the simulation is 12:30 am. Four hundred minutes into the simulation is 6:40 am. Another example, 5:36 pm is 1,056 minutes into the simulation.

* An *arrival event* generates a new Customer object and places it in line. If no customers are currently in line, the new customer proceeds directly to an available cashier (if any).
* A *departure event* relates to a particular cashier. Customers pay the cashier and leave the store. This frees up the cashier for the next customer in line (if any).

**Customers**

Customers are represented with a simple class that maintains the time the customer gets in line. This time is used to determine how long the customer waited in line.

**Cashiers**

Cashiers are represented as a single array of Customer. If no customer is currently with the cashier, the array element contains null. Otherwise, the array element contains a Customer reference. For example, the following array shows four cashiers with customers at location 0 and 2 but not at 1 or 3.

{Customer, null, Customer, null}

**Future Event Queue**

A list of future events is maintained in a Java data structure called a PriorityQueue. It is a special version of an ArrayList that maintains objects in sorted order. Regardless of the order that future events are added, they are removed in the order they should occur. Using PriorityQueues is identical to using ArrayLists with the exception of the method used to remove the next item. Review the following sample code:

GVevent evt;

PriorityQueue <GVevent> futureEvents;

futureEvents = new PriorityQueue <GVEvent> ();

futureEvents.add(evt);

evt = **futureEvents.poll**(); // remove item from front of list

**Step 1: Create a New BlueJ Project**

**Step 2: Use existing class *GVevent***

Rather than writing your own GVevent class, we are providing the completed class for you. Create a new class in BlueJ called GVevent and delete all of the provided code. Copy and paste the provided code from (GVevent.java) into the newly created class. It should compile with no errors. Do not make any changes to this code. However, you should read the internal documentation to understand how it will be used in this project.

**Step 3: Use existing class *GVrandom***

Rather than writing your own GVrandom class, we are providing the completed class for you. Create a new class in BlueJ called GVrandom and delete all of the provided code. Copy and paste the provided code from (GVrandom.java) into the newly created class. It should compile with no errors. Do not make any changes to this code. However, you should read the internal documentation to understand how it works.

**Step 4: Create a class called *Customer* (4 %)**

Implement a Customer class to maintain the time the customer arrived in line (double).

* public Customer (double time) – initialize the instance variable.
* public void setArrivalTime (double t) – set the instance variable.
* public double getArrivalTime () – return the arrival time
* include a brief main method to test this class

**Step 5: Create a GVarrival class (4 %)**

Refer to zyBook 13.1 – 13.3 for basic information about inheritance. Implement a class called GVarrival that extends GVevent.

* public GVarrival (MarketPlace store, double time) – invoke the base class constructor to set the MarketPlace object and the event time.

super(store, time);

* public void process () – use the MarketPlace object in the base class to invoke customerGetsInLine(). One line of code but will not compile until after Step 7.

**Step 6: Create a GVdeparture class (4 %)**

Implement a class called GVdeparture that extends GVevent.

* public GVdeparture (MarketPlace store, double time, int id) – invoke the base class constructor to set the MarketPlace object, the event time, and the cashier ID.

super(store, time, id);

* public void process () – use the MarketPlace object in the base class to invoke customerPays().One line of code but will not compile until after Step 7.

**Step 7: Start the *MarketPlace* class (8 %)**

This is a complex class that relies heavily on the other classes such as Customer, GVarrival, GVdeparture and GVrandom to simulate the arrival and departure of customers throughout the day.

**Instance Variables and Constructors**

A class should contain several instance variables that represent the state of the object.

* Include instance variables for the simulation parameters: average time between customer arrivals (double), average cashier service time (double), number of cashiers (int) and whether the checkout area should be displayed (boolean).
* A double to represent the current time simulation time. It starts at the store’s opening time (10:00 am) and moves forward throughout the day until closing time (6:00 pm).
* An ArrayList of Customer to keep track of customers waiting in line.
* An array of Customer to represent the cashiers.
* A PriorityQueue of GVevent to maintain the list of future events.
* A single instance variable of type GVrandom to generate random arrival and service times throughout the simulation.
* A single String instance variable to contain the results. The String can then be printed to the terminal window within BlueJ or displayed within a GUI. **There should be no print statements in this class**.
* Use private final int members to represent the OPEN (600) and CLOSE (1080) times. These values represent 10:00 am and 6:00 pm.
* You will need several additional instance variables to track and calculate simulation results. This is left for you to figure out.

**Constructor**

Constructors provide initial values to these variables.

* public MarketPlace () – initialize the instance variables. Instantiate the objects. Start values are three cashiers, 6.6 average service time, 2.5 average arrival time and the checkout area should NOT be displayed.

**Accessor and Set Methods**

It is good practice to provide *accessor* methods for most instance members. These methods are informally called *getter* methods and allow access to the state of the object.

* public int getNumCashiers() – return the number of cashiers (one of the simulation parameters). One line of code.
* public double getArrivalTime() – return the average customer arrival time (one of the simulation parameters).
* public double getServiceTime() – return the average cashier service time (one of the simulation parameters).
* public int getNumCustomersServed() – return the number of customers served during the most recent simulation (determined after simulation). One line of code.
* public String getReport() – simulation results are not printed to the screen directly. Instead, all results are appended to a String instance variable. One line of code.
* public int getLongestLineLength() – return the length of the longest line during the simulation. One line of code.
* public double getAverageWaitTime() – return the average wait time of all customers during the simulation. One or two lines of code.
* public void setParameters(int num, double s, double a, boolean ck) – set the instance variables for the number of cashiers, average service time, average arrival time and whether the checkout should be displayed.
* public void customerGetsInLine() – for now, do not include any code in this method but the header is necessary to pass the first round of testing and for GVarrival to compile.
* public void customerPays(int num) - for now, do not include any code in this method but the header is necessary to pass the first round of testing and for GVdeparture to compile.

**See “Step 9: Testing” for information about Junit testing to show that Part 1 works properly.**

**Step 8: Finish MarketPlace class (50 %)**

The following methods are declared private and are only used within the class to carry out specific functions. They are invoked by other class methods. It is a programmer’s design decision to determine when methods should be private.

* private void reset() – reset all instance variables to appropriate starting values for the simulation. Clear the arrays, array lists and priority queues by instantiating new objects. However, DO NOT reset the simulation parameters for cashiers, service time, arrival time and display option.
* private int cashierAvailable() – return an available index in the cashier array (the first element that contains null.) Return -1 if all cashiers are busy.
* private double randomFutureTime(double avg) – returns a future time by adding the current simulation time to a random number generated by the GVrandom nextPoisson(avg). DO NOT modify the current simulation time. For example:

future = now + rand.nextPoisson(avg)

* private void customerToCashier(int num) – this method moves a customer from the front of the line to an available cashier and includes the following steps: 1) remove first customer in line and move to the designated cashier num, 2) update statistics for number of customers served and total wait time for all customers, 3) generate a future time for when the Departure event should occur for this cashier and 4) instantiate a new GVdeparture event and add it to the priority queue. Your variables will have different names but here is an example:

futureTime = randomFutureTime(avgServiceTime);

nextEvent = new GVdeparture(this, futureTime, num);

theEvents.add(nextEvent);

**Primary Methods**

After completing all of the service methods defined above, the primary methods can be written.

* public void customerGetsInLine() – this method simulates a customer getting in line. Several actions are taken: 1) instantiate a new Customer and add it to the customer line, 2) if necessary, update statistics to reflect the time and length of the longest line, 3) if a cashier is available and there is at least one customer waiting in line, move the next customer to the available cashier, 4) if current time is before closing, generate a new Arrival event and add it to the priority queue. Note, this method shoule call internal methods: availableCashier() and customerToCashier().
* public void customerPays(int num) – this method simulates a customer completing a transaction with the cashier and leaving the building. 1) move the next customer in line (if any) to the cashier OR set the cashier to null to indicate no current customer. Note, this method calls another internal method: customerToCashier(num).
* public void startSimulation() – this is the primary method that controls the simulation from beginning to end: 1) reset all instance variables, 2) instantiate the first Arrival event to occur at opening and add it to the event priority queue, 3) use a while loop to continually retrieve the next event in the priority queue, update the current time and invoke process(), 4) if selected, update the results to show the checkout area. This continues until there are no remaining events. 5) After the loop, create the report. Note, this method invokes the service methods reset(), showCheckoutArea(), createReport().

The following excerpt will not compile but shows the basic simulation process.

while(!theEvents.isEmpty()){

GVevent e = theEvents.poll();

currentTime = e.getTime();

e.process();

}

**Additional Methods**

* private void showCheckoutArea () – adds a line of information to the results String to represent the current time, the cashiers and the number of waiting customers. For example, customers at cashiers 0, 2 and 3 with four waiting customers.

12:34 pm C-CC \*\*\*\*

* private void createReport() – calculate the average wait time and append the summary information to the results String (see below for sample output). Nothing is printed within this method and it is only invoked at the end of startSimulation().
* public String formatTime(double mins) – returns a String representation of the provided time. The time represents number of minutes past midnight and should use the following format: 1:23 am, 12:04 pm, 7:34 pm. The solution requires a few if statements to handle all possibilities. This method is public to support testing.

**Coding Style (10 %)**

* Good programming practice includes writing elegant source code for the human reader. Follow the GVSU [Java Style Guide](http://www.cis.gvsu.edu/java-coding-style-guide/).

**Sample Output**

Your results may be different but the end of a simulation should look similar to the following:

*previous output removed*

5:59pm CCCC \*\*\*\*\*

6:00pm CCCC \*\*\*\*

6:00pm CCCC \*\*\*\*\*

6:00pm CCCC \*\*\*\*

6:03pm CCCC \*\*\*

6:04pm CCCC \*\*

6:05pm CCCC \*

6:07pm CCCC

6:08pm \_CCC

6:09pm \_\_CC

6:17pm \_\_\_C

6:19pm \_\_\_\_

SIMULATION PARAMETERS

Number of cashiers: 4

Average arrival: 1.5

Average service: 5.2

RESULTS

Average wait time: 3 mins

Max line length: 10 at 1:33pm

Customers served: 308

Last departure: 6:19pm

**Step 9: Software Testing (5 %)**

Software developers must plan from the beginning that their solution is correct. BlueJ allows you to instantiate objects and invoke individual methods. You can carefully check each method and compare actual results with expected results. However, this gets tedious. Another approach is to write a main method that calls all the other methods.

**Your Testing**

Create a new class called MarketTest. Write a main method to automatically test various methods (see zyBook 9.2). Provide enough tests to convince yourself and others that everything seems to work. Here is a minimal example.

public static void main(String args[]){

System.out.println("Start testing...");

// does store start with 3 cashiers?

MarketPlace myStore = new MarketPlace();

assert(myStore.getNumCashiers() == 3) : "Start with 3 cashiers";

// how many customers served with default arrival time

myStore.startSimulation();

int before = myStore.getNumCustomersServed();

// are parameters updated correctly?

myStore.setParameters(2, 4, 2, false);

assert(myStore.getNumCashiers() == 2) : "Change to 2 cashiers";

// how many customers served with quicker arrival times?

myStore.startSimulation();

int after = myStore.getNumCustomersServed();

assert(before < after) : "Should be more customers";

System.out.println("Testing complete.");

}

**Junit Testing**

JUnit is a Java library that helps to automate software testing. Two JUnit tests are provided on BlackBoard. Follow these instructions to create and use each test.

1. Name your class from step 7 MarketPlace. **Exact spelling is required**.
2. Complete ALL requirements described in Step 2 – 7 or some of the tests in JunitTest1 will fail.
3. Complete ALL requirements described in Step 2 – 8 or some of the tests in JunitTest2 will fail.

## Alternative #1

1. Create a new class in your BlueJ project that is of type **Unit Test**. Call it JunitTest1 and delete all of the provided code. Leave the window open while you continue with step #2.
2. Click on the JunitTest1.java link provided in BlackBoard. This should open the code in a new Window.
3. Cut and paste **all** of the java source code into the newly created class in BlueJ.
4. Compile the class and it should work!

## Alternative #2

1. Download JunitTest1.java from Blackboard and make note where you save it. In BlueJ, select from the menu <Edit> <Add class from file…>, select the file you downloaded and press <Add>.
2. The file should compile with no errors and the class icon will be green.

## Running Junit Test

After the Unit Test has compiled, right click on the green icon and select Test All. A number of automated tests will run. Those that pass will have a green check mark. Those that fail will have a gray X. Click on any of the test names for clues as to what might have gone wrong with your solution. Fix the code and try again!

**Step 10: Create a GUI (15 %)**

Create a fully functional GUI that allows the store manager to try different scenarios. Modify your Project 4 GUI as needed for this project. Duplicate the basic solution shown in Figure 1.

* Define an instance variable of type MarketPlace.
* Instantiate the MarketPlace object in the constructor.
* Provide a menu item to clear the results area.
* Provide a menu item to quit the application.
* For each text field, retrieve String and convert it to a double before the value can be passed to the simulation.

String text = serviceTextField.getText();

double a = Double.parseDouble(text);

* When the user clicks on “Simulate”, call setParameters(), startSimulation() and display the results. Your solution will be slightly different but the following shows the general approach.

theStore.setParameters(c,s,a,displayCB.isSelected());

theStore.startSimulation();

results.append(theStore.getReport());

* Detect if any of the text fields are missing information or contain errors. Display an appropriate warning message (Figure 2). Feel free to use the following helper method that returns true if the String parameter contains a valid number.

private boolean isValidNumber(String str){

boolean isValid = true;

// see if str contains a valid double

try{

Double.parseDouble(str);

// there must have been an error converting to double

}catch (Exception e){

isValid = false;

}

return isValid;

}

** **

**Figure 1. Basic GUI Layout Figure 2. Advanced GUI with error messages**

**Grading Criteria**

There is a 50% penalty on programming projects if your solution does not compile.

* Stapled cover page with your name and signed pledge. (-5 % if missing)

**Late Policy**

* **No late submissions are accepted for this project.**

**Turn In**

A professional document **is stapled** with an attractive cover page. Do not expect the lab to have a working stapler!

* Cover page - Provide a cover page that includes your name, a title, and an appropriate picture or clip art for the project
* Signed Pledge – The cover page must include the following signed pledge: "I pledge that this work is entirely mine, and mine alone (except for any code provided by my instructor). " Also, provide the names of people you worked with and explain the level of cooperation. You are responsible for understanding and adhering to the [School of CIS Guidelines for Academic Honesty](http://www.cis.gvsu.edu/Academics/Honesty/).
* Time Card – The cover page must also include a brief statement of how much time you spent on the project. For example, “I spent 7 hours on this project from January 22-27 reading the book, designing a solution, writing code, fixing errors and putting together the printed document.”
* Sample Output – cut and paste the results from the STU main method.
* Source code - a printout of your elegant source code:

MarketPlace.java

MarketGUI.java

MarketTest.java

* Demo – be prepared to demo your project on a lab computer or your laptop.