Final Report: Design and Implementation of a Power Distribution System Simulator

and Tra Ngoc Nguyen, Hoang Thai Duong, Nguyen The Viet, Nguyen Hai Duc

`cs2015_{nguyen.tn, duong.ht, viet.nt, duc.nh}@student.vgu.edu.vn`

Vietnamese German University, Vietnam

Abstract[1]

The objective of this project is to develop a power distribution simulator for a small power grid. The resulting application should allow the user to simulate and visualize the stability of a power grid with various parameters and show the simulation result as a graph. This simulator is written in Java programming language. The Program consist of 2 Graphical User Interfaces (GUIs). One of which is used to input parameters and the other show the output of those parameters in the form of graphs. The graph shows power demand and power production of consumers and generators in every 2 hours (one iteration) and total power demand and power production in that day. In addition, the simulator also displays the frequency of the grid in each iteration and profit the generators make in a day.

[1]Author: Tra Ngoc Nguyen.

Contents

Listings

List of Figures

Chapter 1

Introduction

1.1 Motivation[1]

This power balance simulation project is a Project Module for bachelor students of Computer Science at the Vietnamese-German university. The main purpose of the whole project is for learning. Moreover, power is becoming more important as demand for energy is increasing day by day. In reality, energy consumption changes from time to time, which has a huge impact on energy supply. Therefore, the simulator exists as a way to monitor the demand-supply relationship of power. From the simulation, analysis in the field of statistics could be applied to handle all unbalanced cases. Additionally, the program would be useful to teach children how a power grid is like and how power is balanced. Due to the significant development of technology, electronic vehicles such as Tesla cars will soon arise and dominate the market, which leads to electricity stations for those. The simulation will be helpful when it can simulate any new type of energy without setting up in reality. This avoids the problems of time consuming and cost intensive in such an efficient way. The usage of this project can be broadened as it depends on the purposes of users.

[1]Author: Hoang Thai Duong.

1.2 Objective[2]

The objective of this project is to develop a mains frequency simulator for a small metropolitan power grid. The resulting application should allow the user to simulate and visualize the stability of a power grid with various parameters. Simulations can be used to analyze and evaluate various control mechanisms. They allow to generate unorthodox consumer profiles or abnormal scenarios, that are not easy to reproduce in real world environments.

[2]Author: Hoang Thai Duong.

1.3 Requirements: [3]

In the course of this project, we must meet the following requirements:

1.3.1 Generator Model Requirements

MUST be registered at the Control Model

MUST provide an interface to measure the momentary power

MUST provide the minimum and maximum supply

MUST provide the maximum supply-change per iteration

MUST provide an interface for the Control Model to request a supply change

MAY change supply due to internal or external conditions (e.g. time, weather, price ..)

MAY turn automatically off after a nr. of iterations (e.g. battery, pump power plant)

1.3.2 Consumer Model Requirements

MUST be registered at the Control Model

MUST have an On and Off state, each with defined power

MAY be registered as cluster with other consumers (e.g. households)

MUST provide an interface to measure the power

MUST provide an interface for remote changes

MAY change state due to internal or external conditions (e.g. time, weather, price ..)

[3]Author: Tra Ngoc Nguyen

1.3.3 Control Model Requirements

MUST be able register an arbitary number of generators and consumers

MUST be able to un-register each component

MUST compute the total demand every iteration

MUST compute the total cost every iteration

MUST compute the mains frequency every iteration (50Hz minus the difference of demand-supply, whereas 10% equals 1Hz)

MUST unregister 10% Generators, if mains frequency $> 51Hz$ (overload)

MUST unregister 15% of the consumers if mains frequency $< 49Hz$ (blackout)

MUST unregister components if mains frequency $> 51Hz < 49Hz$ for 3 Iterations (defect)

MAY request state changes in consumers for demand side management

MAY request supply and state changes in generators for demand side management

MAY request to start or shutdown generators for demand side management

MAY change the electricity price

1.3.4 Graphical User Interface Requirements

MUST be capable to control current the iteration

MUST be capable to show current demand, supply and frequency

MUST be capable to show the number of consumer and generators

MUST be capable to show current weather and electricity price

MUST be capable to show name and power of individual consumer or generator

1.4 Outline [4]

This report consist of 2 major part - First, a detail analysis of our software. This include software architecture, use-case analysis, class diagram. The second portion will be Code explanation base on the analysis in the previous chapter. As said. The rest of the report will be organized as follow:

Chapter 2 : Software analysis

Chapter 3 : Code explanation

Chapter 4 : Conclusion

[4]Author:Tra Ngoc Nguyen

Chapter 2

Software Analysis [1]

2.1 Introduction

In this chapter we explain which pattern we use to develop this simulation software, which use-cases we expected to implement to the program and the class diagram of this application. These information will be the base line we use to create the software

[1]Author:Tra Ngoc Nguyen

2.2 Use-cases Diagram

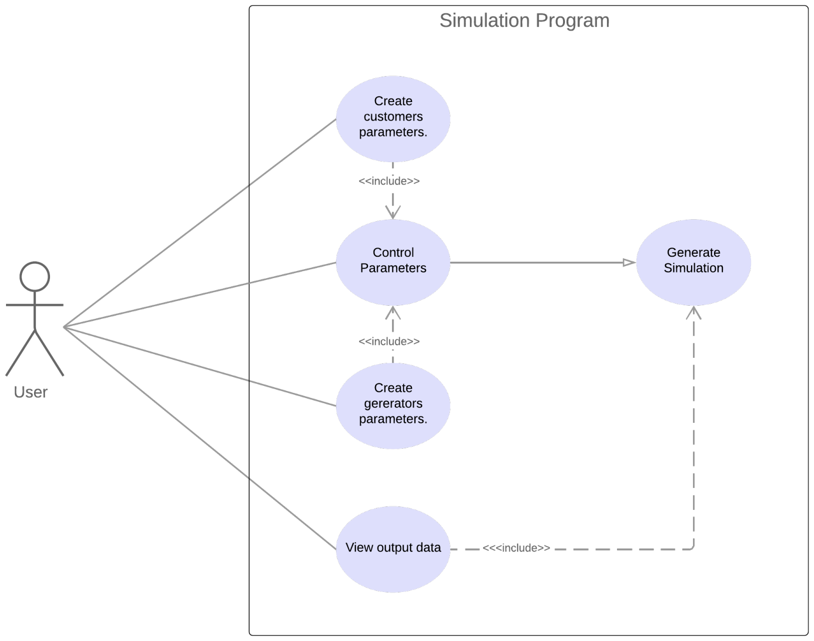
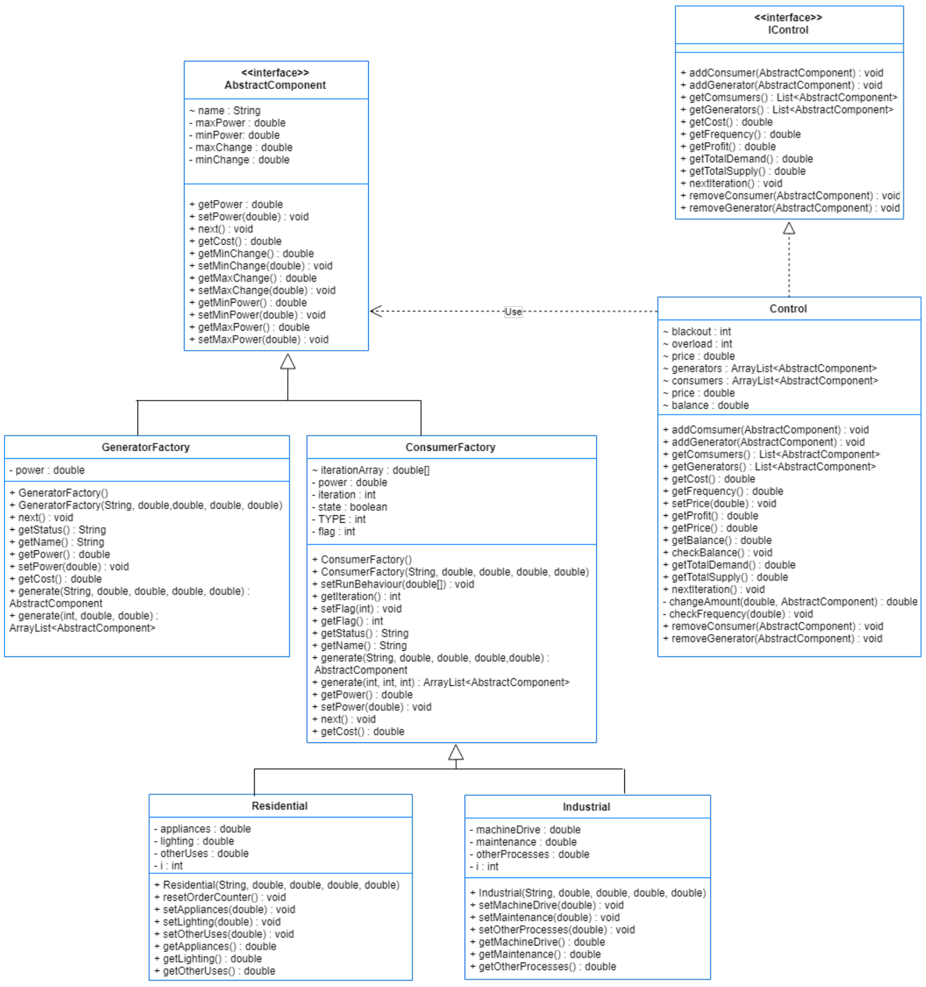


Figure 2.1: Propose Use-case diagram of this program

Figure 2.2 show how the user interact with the program. We want to minimize user interaction to the Consumer and Generator package. By limiting user accessibility to only `parameters` that user can input to the program, we can minimize the complexity of data flow. In this program, the user can modify the parameters that influence the creation of different object in the program such as: Consumer, Generator. The user can also change simulation's parameter while the simulation is running. The detail of these parameter will be explained in the next chapter where we analyze our code. The information provided by the user then will be used to run the simulator. The output data then will be showed so that the user can see and evaluate the solution.

2.3 Class Diagram [2]



<<interface>>
IControl
+ addConsumer(AbstractComponent) : void
+ addGenerator(AbstractComponent) : void
+ getComsumers() : List<AbstractComponent>
+ getGenerators() : List<AbstractComponent>
+ getCost() : double
+ getFrequency() : double
+ getProfit() : double
+ getTotalDemand() : double
+ getTotalSupply() : double
+ nextIteration() : void
+ removeConsumer(AbstractComponent) : void
+ removeGenerator(AbstractComponent) : void
<<interface>>
AbstractComponent
~ name : String
- maxPower : double
- minPower: double
- maxChange : double
- minChange : double
+ getPower : double
+ setPower(double) : void
+ next() : void
+ getCost() : double
+ getMinChange() : double
+ setMinChange(double) : void
+ getMaxChange() : double
+ setMaxChange(double) : void
+ getMinPower() : double
+ setMinPower(double) : void
+ getMaxPower() : double
+ setMaxPower(double) : void
Use
Control
~ blackout : int
~ overload : int
~ price : double
~ generators : ArrayList<AbstractComponent>
~ consumers : ArrayList<AbstractComponent>
~ price : double
~ balance : double
+ addComsumer(AbstractComponent) : void
+ addGenerator(AbstractComponent) : void
+ getComsumers() : List<AbstractComponent>
+ getGenerators() : List<AbstractComponent>
+ getCost() : double
+ getFrequency() : double
+ setPrice(double) : void
+ getProfit() : double
+ getPrice() : double
+ getBalance() : double
+ checkBalance() : void
+ getTotalDemand() : double
+ getTotalSupply() : double
+ nextIteration() : void
- changeAmount(double, AbstractComponent) : double
- checkFrequency(double) : void
+ removeConsumer(AbstractComponent) : void
+ removeGenerator(AbstractComponent) : void
GeneratorFactory
- power : double
+ GeneratorFactory()
+ GeneratorFactory(String, double,double, double, double)
+ next() : void
+ getStatus() : String
+ getName() : String
+ getPower() : double
+ setPower(double) : void
+ getCost() : double
+ generate(String, double, double, double, double) : AbstractComponent
+ generate(int, double, double) : ArrayList<AbstractComponent>
ConsumerFactory
~ iterationArray : double[]
- power : double
- iteration : int
- state : boolean
- TYPE : int
- flag : int
+ ConsumerFactory()
+ ConsumerFactory(String, double, double, double, double)
+ setRunBehaviour(double[]) : void
+ getIteration() : int
+ setFlag(int) : void
+ getFlag() : int
+ getStatus() : String
+ getName() : String
+ generate(String, double, double, double,double) : AbstractComponent
+ generate(int, int, int) : ArrayList<AbstractComponent>
+ getPower() : double
+ setPower(double) : void
+ next() : void
+ getCost() : double
Residential
- appliances : double
- lighting : double
- otherUses : double
- i : int
+ Residential(String, double, double, double, double)
+ resetOrderCounter() : void
+ setAppliances(double) : void
+ setLighting(double) : void
+ setOtherUses(double) : void
+ getAppliances() : double
+ getLighting() : double
+ getOtherUses() : double
Industrial
- machineDrive : double
- maintenance : double
- otherProcesses : double
- i : int
+ Industrial(String, double, double, double, double)
+ setMachineDrive(double) : void
+ setMaintenance(double) : void
+ setOtherProcesses(double) : void
+ getMachineDrive() : double
+ getMaintenance() : double
+ getOtherProcesses() : double

Figure 2.2: Class Diagram of this program

Figure 2.2 describes the structure of the system by showing the system's classes, their attributes, methods and the relationships among objects. The system uses factory pattern to create objects, the two classes GeneratorFactory and Con-

[2] Author : Nguyen Hai Duc

sumerFactory implement the AbstractComponent interface. There are 2 types of Consumer, Residential and Industrial. Control class uses AbstractComponent interface to generate objects of consumer and generator. The class Control, implements IControl interfaces, contains all methods needed for controlling the system (manage objects, calculate balance, check frequency, etc).

2.4 Conclusion

This chapter describe how our group analyze the problem, implement what we have learned to design the structure of this program. We have come up with the Class Diagram for the core software and multiple Use-cases for the functionality of the software. The information we have showed above will help us greatly in making a working program.

Chapter 3

Modules

3.1 Introduction [1]

In this chapter, the code of the application will be detaily explained. The chapter is divided into four part corresponding to four main classes which have been explained in chapter 2.

[1] Author: Tra Ngoc Nguyen

3.2 Consumer Module

3.2.1 Initial implementation [2]

The ConsumerFactory class is used for creating an instance of consumer to act as an independent entity in the system. It implements AbstractComponent interface and contain following variables and methods:

3.2.1.1 Variables

- **static run_patt**
  This variable is an array that store the electricity usage pattern of several instances of consumer. The value of the array can be uniquely set for a group of consumers.
- **name, maxPower, minPower, maxChange, minChange**
  These variables, which are inherited from AbstractComponent, are initial variables which are required to be set when the consumer is created. These variables identify the name of the consumer, it's maximum and minimum power usage per iteration, it's maximum and minimum power change per iteration.
- **iteration**
  This variable show which iteration is this consumer is currently in.
- **power**
  This variable show how much power/electricity this consumer is using in its current iteration.
- **state**
  This variable show whether this consumer is active or non-active(use minimum power) in the current iteration.

3.2.1.2 Method

- **static void setRunBehavior(double[] run_patt)**
  This is a static method in the ConsumerFactory Class. The method receives an array as an argument. The input array will be checked for compatibility. If the array is not compatible, it will throw an exception.

[2] Author: Tra Ngoc Nguyen

Listing 3.1: Code for compatibility check in **setRunBehavior** method

```
  if (run_patt.length != 12) {
      throw new IndexOutOfBoundsException("The Run Pattern is
          not compatible with this program. Please try again");
    }
```

The code snippet above show how the method handles incompatible run pattern. If the run pattern not equal to number of iterations per day, in this case twelve per day. The method will throw an IndexOutOfBound exception. This exception will prevent the application from running.

- **public ConsumerFactory(String name, double maxPower, double minPower, double maxChange, double minChange)**
  This method is the constructor of ConsumerFactory class. This method create an instance of consumer and assign initial values to its initial variables.

- **public static AbstractComponent generate(String name, double maxPower, double minPower, double maxChange, double minChange)**

- **public static ArrayList¡AbstractComponent¿ generate(int amount, int avg_max_Power, int deviation)** The two methods above are designed to be used by the Control class to create a (set of) consumer(s). The first method receives five argument to set the initial value for one consumer. The second method accept 3 arguments to create a set (group) of consumers. The first argument is the number of consumers in the set, the second argument set the maximum power per consumer per iteration, the third argument set how the power of these consumers different from each other.

Listing 3.2: Code for randomize max power in **generate** method

```
ArrayList<AbstractComponent> consumers = new
    ArrayList<AbstractComponent>();
for (int i = 1; i <= amount; i++) {
      double val_max = (-1 + Math.random() * (1 - (-1))) +
          (double) avg_max_Power;
      consumers.add(new ConsumerFactory("c" + i, val_max, 1,
          val_max, val_max));

   }
```

The Listing above show how the method generate consumers with different max power. For each consumer, the method generates a max power value by generate a random real number in range [-1,1]. This random number then will multiply to the deviation and add to the average max power to create a new max power for a consumer.

- **public void setPower(double power)**
  This method receives an argument with the purpose to forcefully change the power usage of an instance of consumer. In addition, the method can receive a special value that when the method receives this value, it will forcefully shutdown the consumer in this current iteration

  Listing 3.3: Code for force shutdown the consumer

  ```
  if (power == -1) {
  this.state = false;
  this.power = 0;
  ```

  The listing show that, when the method receive the value of -1, it will change the state of the consumer to false and set it power to 0

- **public void next()**
  This method advances the iteration of this instance of consumer. After moving to the next iteration, the method then set whether this consumer is active or not in this iteration.

  Listing 3.4: Code for set stage in **next** method

  ```
     this.iteration++;
      if (Math.random() <
          ConsumerFactory.run_patt[this.iteration]) {
         this.state = true;
         this.power = this.getMaxPower();
  } else {
         this.state = false;
         this.power = this.getMinPower();
  }
  ```

  The listing above show how the method decide which consumer is active or not. According to the requirement, the power of an instance of consumer in an iteration equal `max power multiply by x` with x range in [0,1]. This however, is very problematic to code into the program. Instead, we use the random distribution property of `Math.random()` method to set the power in the set of consumer since $\sum_1^x modifier * maxPower = modifier * \sum_1^x maxPower$

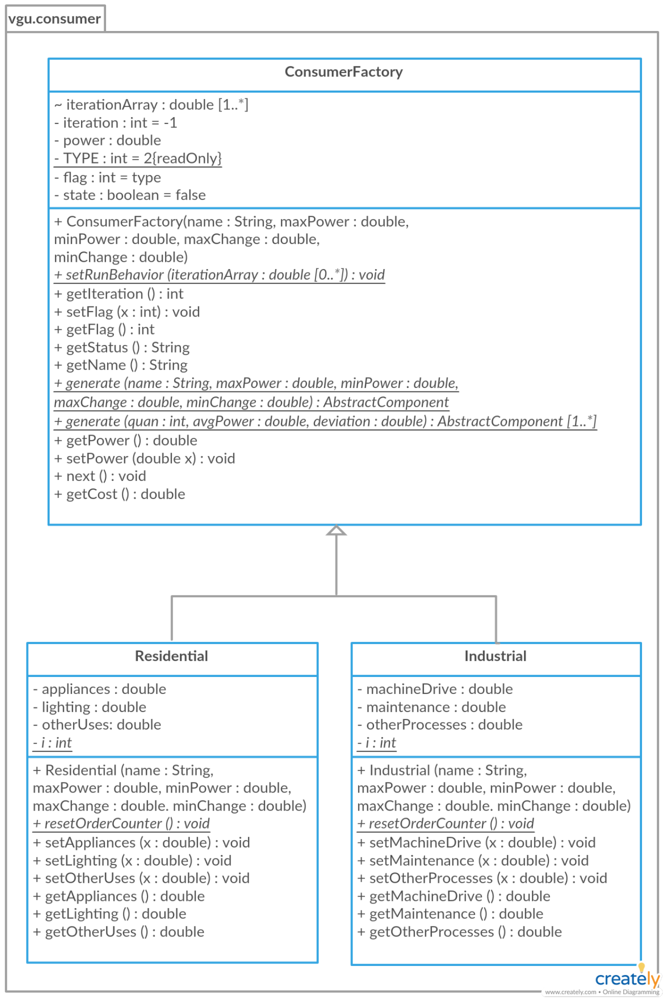
- **public String getStatus()**
  This method get the state of the consumer. This state is different from the variable **state** in variable section. This state is determine by the variable **state** and **power**

- **get/set method for variables**
  These methods are used to get the value of variables in this class.

3.2.2 Additional implementation and modifications[3]

3.2.2.1 Additional implementaion

Categorizing consumers into specific types that are ***Residential*** and ***Industrial***.


vgu.consumer
ConsumerFactory
~ iterationArray : double [1..*]
- iteration : int = -1
- power : double
- TYPE : int = 2{readOnly}
- flag : int = type
- state : boolean = false
+ ConsumerFactory(name : String, maxPower : double, minPower : double, maxChange : double, minChange : double)
+ setRunBehavior (iterationArray : double [0..*]) : void
+ getIteration () : int
+ setFlag (x : int) : void
+ getFlag () : int
+ getStatus () : String
+ getName () : String
+ generate (name : String, maxPower : double, minPower : double, maxChange : double, minChange : double) : AbstractComponent
+ generate (quan : int, avgPower : double, deviation : double) : AbstractComponent [1..*]
+ getPower () : double
+ setPower (double x) : void
+ next () : void
+ getCost () : double
Residential
- appliances : double
- lighting : double
- otherUses: double
- i : int
+ Residential (name : String, maxPower : double, minPower : double, maxChange : double. minChange : double)
+ resetOrderCounter () : void
+ setAppliances (x : double) : void
+ setLighting (x : double) : void
+ setOtherUses (x : double) : void
+ getAppliances () : double
+ getLighting () : double
+ getOtherUses () : double
Industrial
- machineDrive : double
- maintenance : double
- otherProcesses : double
- i : int
+ Industrial (name : String, maxPower : double, minPower : double, maxChange : double. minChange : double)
+ resetOrderCounter () : void
+ setMachineDrive (x : double) : void
+ setMaintenance (x : double) : void
+ setOtherProcesses (x : double) : void
+ getMachineDrive () : double
+ getMaintenance () : double
+ getOtherProcesses () : double
creately
www.creately.com • Online Diagramming

[3]Author: Hoang Thai Duong

Figure 3.1: Class diagram of package vgu.consumer.

Each type of consumer has its fields which represent what purposes energy is consumed. For simplicity, each type has only 3 main activities.

A Residential consumer has 3 power consumption activities:

- Appliances
- Lighting
- Other devices

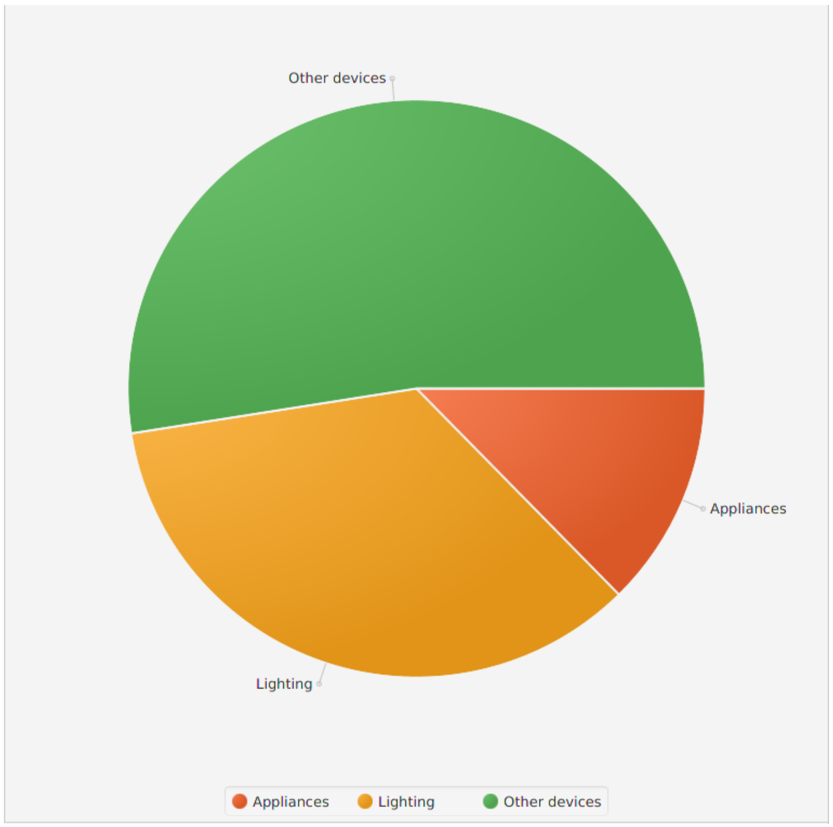


Figure 3.2: Power consumption activities of a Residential user.

Respectively, an Industrial consumer also has 3 power consumption activities:

- Machine drive
- Maintenance
- Other processes

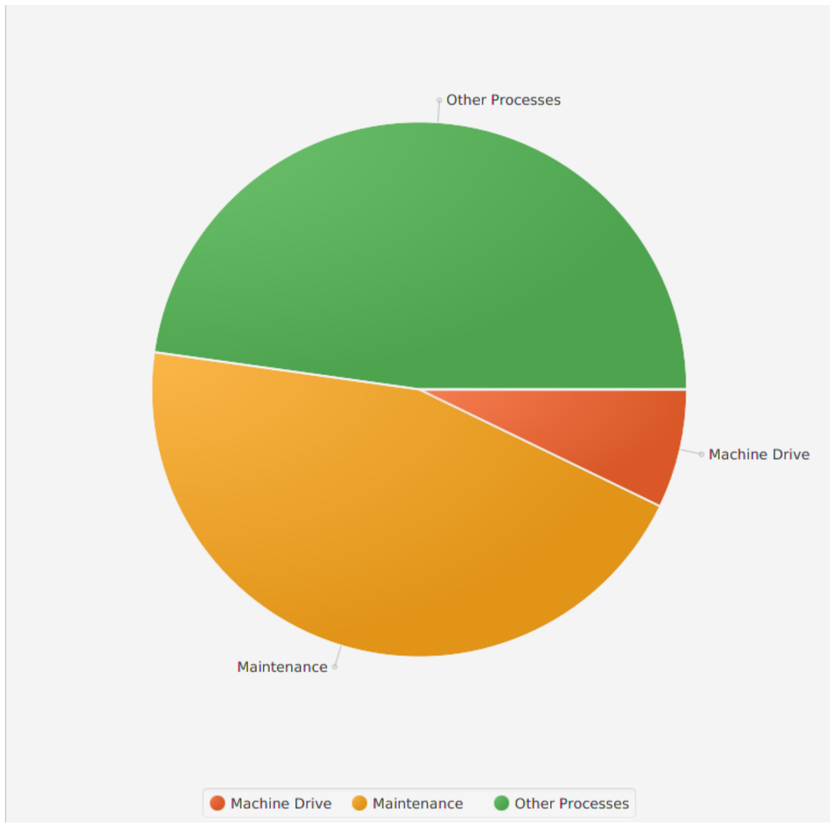


Figure 3.3: Power consumption activities of an Industrial user.

3.2.2.2 Modifications

- In the base class ***ConsumerFactory***, variable ***type*** represents total number of consumer's kinds (In the program is Residential and Industrial). Variable ***flag*** is initialized with value of ***type*** as default.

Listing 3.5: Initialization of type and flag

```
private static final int type = 2;
private int flag = type;
```

- Only when a list of multiple consumers are generated, each consumer in the list is classified by a random function. Otherwise, the function to generate a single consumer just makes consumer be of the type ***ConsumerFactory***.

Listing 3.6: Categorizing process of multiple consumers

```
int flag = new Random().nextInt(type);
  if (flag == 0) {
     consumers.add(new Residential("Residential area ", power,
         Math.ceil(power*.2), power*.6, power*.3));
  } else if (flag == 1){
     consumers.add(new Industrial("Industrial park ", power,
         Math.ceil(power*.2), power*.6, power*.3));
  }
```

- ***Residential*** has flag's value of 0 and ***Industrial*** of 1. The constructors of derived classes call the constructor of their base class by the keyword ***super*** and set their own flag's value by ***setFlag()***.

Listing 3.7: Constructor of class Residential

```
public Residential(String name, double maxPower, double
    minPower, double maxChange, double minChange) {
  super(name+i, maxPower, minPower, maxChange, minChange);
  setFlag(0);
  ++i;
}
```

- Energy consumption can only be set for consumers who are marked as "on", which means they are consuming energy. Depending on the random value, new energy is set for consumer. Previously, power was set directly by assigning the variable ***power*** with max power or min power. However, to reduce the surprising increase amount of energy consumption, new energy has to be set by the function ***setPower()***. This makes the power setting process be restricted in the range of minimum and maximum changing amount.

Listing 3.8: Function next() was modified

```
 public void next() {
   ++iteration;
   if(getStatus().equals("On")) {
      if (Math.random() <
          ConsumerFactory.iterationArray[iteration]) {
         state = true;
         setPower(getMaxPower());
      } else {
         state = false;
         setPower(getMinPower());
      }
   }
}
```

3.3 Control Module

3.3.1 Core implementation [4]

The Control class is used for controlling the components (generators and consumers) of the program as well as calculating data (price, cost, profit, supply, demand, etc). It implements IControl interface and contains the following variables and methods :

3.3.1.1 Variable

- **blackout, overload : int**
  These 2 variables of type integer are used for counting the number of continuous blackout (mains frequency ¡ 49Hz) or overload (mains frequency ¿ 51 Hz) iterations. If one of them reach 3, the system is defect, all the generators and consumers will be unregistered.
- **price : double**
  The electricity price (Euro per Watt) for Pricing Model.
- **generators, consumers : ArrayList¡AbstractComponent¿**
  These 2 array lists of type AbstractComponent are used to store all the working consumers and generators.
- **balance : double**
  The total value of profit minus cost, it is calculated after each iteration and checked for positive.

3.3.1.2 Method

- **addComsumer(AbstractComponent) : void**
  This method takes a variable of type AbstractComponent as input and adds it to the consumers array.
- **addGenerator(AbstractComponent) : void**
  This method takes a variable of type AbstractComponent as input and adds it to the generators array.
- **getComsumers() : List¡AbstractComponent¿**
  This method returns the current consumers array list of the control instance.

[4]Author: Nguyen Hai Duc

- **getGenerators() : List¡AbstractComponent¿**
  This method returns the current generators array list of the control instance.

- **removeConsumer(AbstractComponent) : void**
  This method takes a variable of type AbstractComponent as input and removes it from the consumers array list.

- **removeGenerator(AbstractComponent) : void** This method takes a variable of type AbstractComponent as input and removes it from the generators array list.

- **getCost() : double**
  This method calculates the total cost by adding costs of all the generators, single generator's cost is calculated by calling the getCost() method of the class GeneratorFactory. The result is returned as a variable of type double.

- **getFrequency() : double**
  This method calculates the frequency of the current iteration and returns the result of type double. The mains frequency is calculated using the formula : 50Hz minus the difference of demand-supply, whereas 10% equals 1Hz.

Listing 3.9: code for calculating frequency

```
double diff = (getTotalDemand() - getTotalSupply())/
        Math.max(getTotalDemand(), getTotalSupply());
    return 50.0D - diff*10;
```

- **setPrice(double) : void**
  This method changes the current electricity price to the input value if the input value is greater than 0.

- **getPrice() : double**
  This method returns the current electricity price as a variable of type double.

- **getProfit() : double**
  This method calculates the profit of the current iteration by multiplying the total demand by the electricity price and returns the profit as a variable of type double.

- **getBalance() : double**
  This method returns the current balance as a variable of type double.

- **checkBalance(double) : void**
  This method checks if the current gross profit is positive or not. If the gross profit is negative, it takes the amount needed to make the gross profit equals to zero and divides that amount by the total supply. Then

it calls the setPrice() method to set the new electricity price as the old price plus the result.

Listing 3.10: code for setting new price to cover negative amount of gross profit

```
 if(amount < 0) {
           double x = Math.abs(amount)/getTotalSupply();
           double y = Math.round(x * 100d) / 100d;
           if (y < x) {
              y += 0.01;
           }
           setPrice(price + y);
   }
*y is x after being rounded to two decimal places, if x is rounded
    down, y will be increased by 0.01 to ensure that the new gross
    profit is non-negative.
```

- **getTotalDemand() : double**
  This method calculates the total demand of the consumers by adding the power of all the consumer in the array, the power of a single consumer is from the method getPower() of the class ConsumerFactory. The result is returned as a variable of type double.

- **getTotalSupply() : double**
  This method calculates the total supply of the generators by adding the power of all the generator in the array, the power of a single generator is from the method getPower() of the class GeneratorFactory. The result is returned as a variable of type double.

- **nextIteration() : void**
  This method adjusts the power of the generators to minimize the difference between generators and consumers. First it computes the difference between the total demand and total supply, then it increases or decreases each generator base on that amount. The method changeAmount() is used to calculate the amount of power that each generator need to change.

Listing 3.11: code for adjusting the generator's supply

```
     double changeAmount = 0;
      double diff = getTotalSupply() - getTotalDemand();
      double sign = diff/Math.abs(diff);
      for (AbstractComponent g : generators) {
        changeAmount = changeAmount(diff,g)*sign;
        g.setPower(g.getPower() - changeAmount);
        diff -= changeAmount;
      }
```

After that, the method checkFrequency() is called to check for blackout/overload and defect condition and act accordingly.

Finally, the method checkBalance() is used to check the gross profit of current iteration, if the gross profit is negative, the method checkBalance() will adjust the price to cover that negative amount. If the gross profit is non-negative, it is added to the balance.

Listing 3.12: code for calculate new balance

```
checkBalance(getProfit() - getCost());
 balance += (getProfit() - getCost());
```

- **changeAmount(double, AbstractComponent) : double**
  This method calculate the amount of power that is available for changing of the given generator in the current iteration by considering its current power, max power, min power, max change and min change. Then it compares that available amount with the amount of power need to be adjusted and return the amount of power that the given generator should change.

  Listing 3.13: code for calculate the amount of power the generator need to change

```
if ( diff > 0 ) {
   availableChange = Math.min(g.getMaxChange(), g.getPower()
       - g.getMinPower());
}
else if ( diff < 0 ) {
   availableChange = Math.min(g.getMaxChange(),
       g.getMaxPower() - g.getPower());
}
if ( availableChange < g.getMinChange()) {
   availableChange = 0;
}
else if ( Math.abs(diff) < availableChange) {
   availableChange = Math.max(Math.abs(diff),
       g.getMinChange());
}
return availableChange;
```

- **checkFrequency(double) : void**
  This method takes the frequency of the iteration and checks if it is blackout (mains frequency ¡ 49Hz), overload (mains frequency ¿ 51 Hz) or defect (mains frequency ¿ 51Hz ¡ 49 Hz for 3 Iterations). If there is a blackout, it reduces the size of the consumers array by 15% and increases the blackout variable by 1. If there is an overload, it reduces the size of the generators

array by 10% and increases the overload variable by 1. If there is a defect, it clears both the generators and consumers array.

3.3.2 Additional implementation [5]

Because implementation was done in the package ***vgu.consumer***, not only class ***Control*** had to be modified in some functions to adapt with changes from class ***ConsumerFactory***, but also some new functions were defined.

- Apart from the original list ***consumers*** to store consumers, two new lists ***industrial*** and ***residential*** are created.

Listing 3.14: New lists created

```
ArrayList<AbstractComponent>
                consumers = new ArrayList<AbstractComponent>(),
                industrial = new ArrayList<AbstractComponent>(),
                residential = new ArrayList<AbstractComponent>();
```

- The function ***addConsumer()*** was modified to be able to simultaneously add new objects to the list ***consumers*** together with classifying and adding them into one of the two newly created lists based on their flag's values.

Listing 3.15: Function addConsumer() was modified

```
 consumers.add(consumer);
// categorize consumers
int flag = ((vgu.consumer.ConsumerFactory)consumer).getFlag();
if (flag == 0) {
   residential.add(consumer);
} else if (flag == 1){
   industrial.add(consumer);
}
```

- The same modification for ***removeConsumer()***. A ***ConsumerFactory*** object is removed in the list ***consumers*** so that the object in one of the two lists ***residential***, ***industrial*** has to be removed.

Listing 3.16: Function removeConsumer() was modified

```
int flag = ((vgu.consumer.ConsumerFactory)consumer).getFlag();
 if (flag == 0) {
```

[5] Author: Hoang Thai Duong

```
        residential.remove(consumer);
    } else if (flag == 1) {
        industrial.remove(consumer);
    }
consumers.remove(consumer);
```

- With the idea of dividing demanding power of a consumer into activities, the following 3 functions were implemented.

Listing 3.17: Additional functions implemented in Control

```
public void setActivityPower() {}
private void setIndustrial(double l1, double r1, double l2,
    double r2){}
private void setResidential(double l1, double r1, double l2,
    double r2){}
```

- Each activity in an iteration has its range of percentage of power consumption. Power consumption is distributed as in the following table.

Iteration / Activity	[1, 2] 0a.m - 4a.m	[3, 8] 4a.m - 4p.m	[9, 12] 4p.m - 12p.m
Residential Consumption			
Appliances	10%-15%	35%-40%	25%-35%
Lighting	30%-40%	35%-40%	35%-45%
Other devices	The rest	The rest	The rest
Industrial Consumption			
Machine drive	5%-10%	50%-80%	25%-35%
Maintenance	40%-50%	5%-10%	25%-35%
Other processes	The rest	The rest	The rest

Power distribution table was based on our thoughts. For instance, from 0a.m to 4a.m, most people go to sleep and do not use appliances. However, percentages increase from 4a.m to 4p.m as that is the working hours. Generally, for residential activities, percentages are higher than usual from iteration 3 to iteration 12 simply because people work in that period of time.
For the power distribution of Industrial consumer, we thought that factories reduce their productivity at night (0a.m to 4a.m) due to the early morning shift. Thus, more power is for maintaining machines. However, from 4a.m till 4p.m, workers and engineers go to work, which increases the productivity of factories and decreases power for maintenance. The rest of the day, percentages are equal for both activities.

3.4 Generator Module [6]

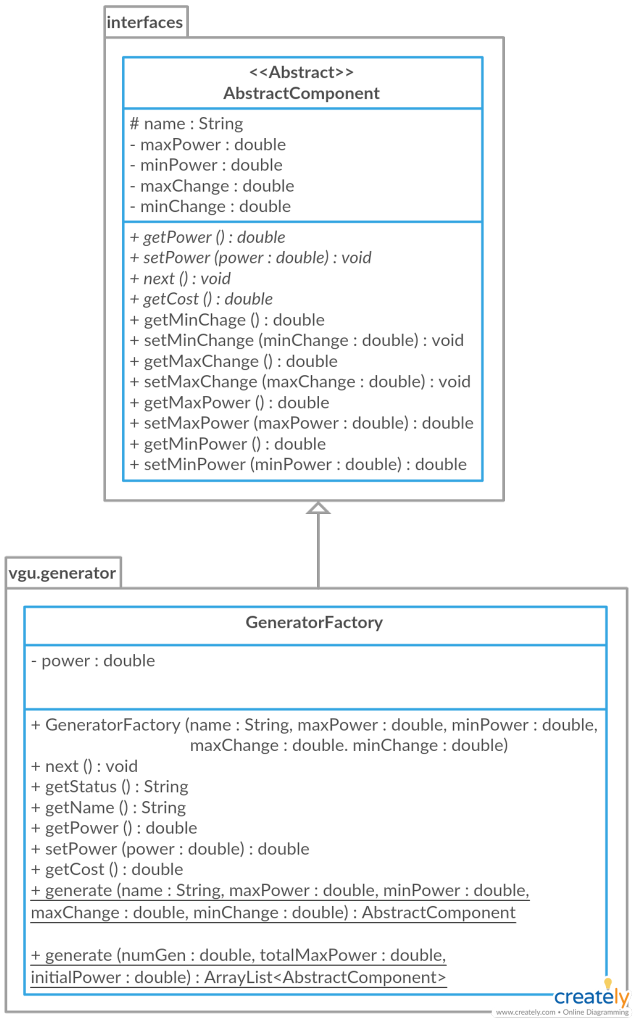

interfaces
<<Abstract>>
AbstractComponent
# name : String
- maxPower : double
- minPower : double
- maxChange : double
- minChange : double
+ getPower () : double
+ setPower (power : double) : void
+ next () : void
+ getCost () : double
+ getMinChage () : double
+ setMinChange (minChange : double) : void
+ getMaxChange () : double
+ setMaxChange (maxChange : double) : void
+ getMaxPower () : double
+ setMaxPower (maxPower : double) : double
+ getMinPower () : double
+ setMinPower (minPower : double) : double
vgu.generator
GeneratorFactory
- power : double
+ GeneratorFactory (name : String, maxPower : double, minPower : double, maxChange : double. minChange : double)
+ next () : void
+ getStatus () : String
+ getName () : String
+ getPower () : double
+ setPower (power : double) : double
+ getCost () : double
+ generate (name : String, maxPower : double, minPower : double, maxChange : double, minChange : double) : AbstractComponent
+ generate (numGen : double, totalMaxPower : double, initialPower : double) : ArrayList<AbstractComponent>
creately

Figure 3.4: Class diagram of AbstractComponent and GeneratorFactory.

[6] Author: Hoang Thai Duong

The class diagram already depicted the inheritance relationship between classes AbstractComponent and GeneratorFactory. Because AbstractComponent was given as a template in this project with the aim to exchange components between different groups. For instance, the class GeneratorFactory of our group will replace the corresponding GeneratorFactory class of another group and the other way round. The most important point is to make sure the simulator runs properly without any error during the exchange process. That is the reason why GeneratorFactory inherited AbstractComponent.

The main function in this class is creating generators. This could be done by either initialize a single generator or a list of multiple generators.

Listing 3.18: Function prototype for creating a single generator

```
public static AbstractComponent
    generate(String name, double maxPower, double minPower,
                            double maxChange,double minChange) {}
```

Listing 3.19: Function prototype for creating a list of generators

```
public static ArrayList<AbstractComponent>
    generate(int numGen, double totalMaxPower, double initialPower)
        {}
```

Electricity generated by a power plant has a producing cost which is calculated by multiply its generated power with a coefficient which has a specified formula.

Listing 3.20: Function returning producing cost of a power plant

```
public double getCost() {
  double coefficient = .5,
       maxPower = getMaxPower(),
       maxChange = getMaxChange(),
       minChange = getMinChange();

  coefficient += maxPower<2500 ? .25 : 0;
  coefficient += maxChange>.5 * maxPower ? .25 : 0;
  coefficient += minChange<.5 * maxPower ? .25 : 0;

  return coefficient * getPower();
}
```

3.5 Graphical User Interface

This chapter describes the Graphical User Interface (GUI) of the Simulator and tools used for implementation. The GUI fulfills all initial requirements of the project. Because all team members had no experience in designing Graphical User Interface using Java language and its library, it was quite a challenge at first for me to get acquaintance with JavaFX and how to control the FXML files. Eventually, I managed to finish the project in time with some new additional features for the simulation: turn consumers off (reduce the power to zero), remove certain number of consumers or generators, compare power distribution of two new types of consumers (Industrial and Residential) during each iteration. After the simulation completes, the application will generate detailed reports in text format (.txt files).

3.5.1 Tool and Library[7]

The Graphical User Interface is built with:

•**Library:** JavaFX

•**Design Tool:** Java Scene Builder 2.0.

Diagram in this section:

•**Diagram Tool:** Creatly (Online Application).

•**Diagram Types:** Structure Diagram (Sketch), Use Case Diagram, Activity Diagram.

We used Java Scene Builder as visual layout tool to design FXML files, which provide the structure for building user interface separated from the logic code. Each FXML file is then controlled by logic code of a Java Class (Controller Class).

[7]Author: Nguyen The Viet - 9990

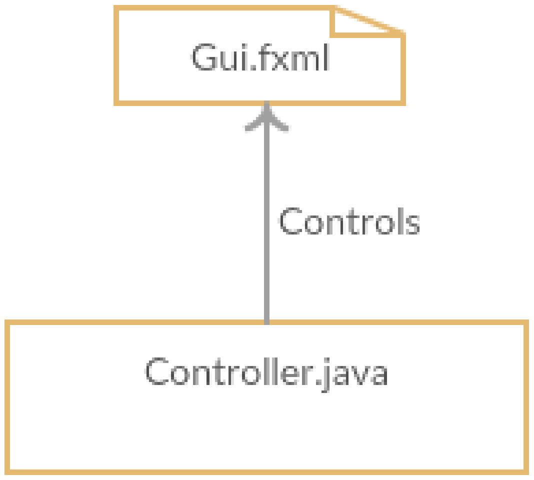


Figure 3.5: FXML and Controller relationship

In this project, we created 5 different FXML with 5 Controllers as following:

1. InputData.fxml - InputController.java.

2. LineChart.fxml - ChartController.java.

3. PieChart.fxml - PieChartController.java.

4. Table.fxml - TableController.java.

5. PopUp.fxml - PopUpController.java .

The remainder of this chapter is divided into the following sections:

•***GUI Structure Design:*** describes the structure of the GUI of the Simulator and the ways user can interact with the application.This section includes a simple diagram which illustrates the structure of the graphical user interface, a use case diagram for GUI application and an activity diagram which shows how data flows during the simulation.

•***Screen Description:*** contains detailed description of each window and functionality of its component parts. This section is the core part of the whole section.

3.5.2 Structure Design[8]

The overall structure design of the application is shown in the following diagram.

Note: each number in the first order numbering in the diagram refers to individual screens. Each screen contains several component parts which are defined as tab or button. There are five screens in total which interact with each other. The screen at the arrowhead is showed up if the button at the tail is pressed.

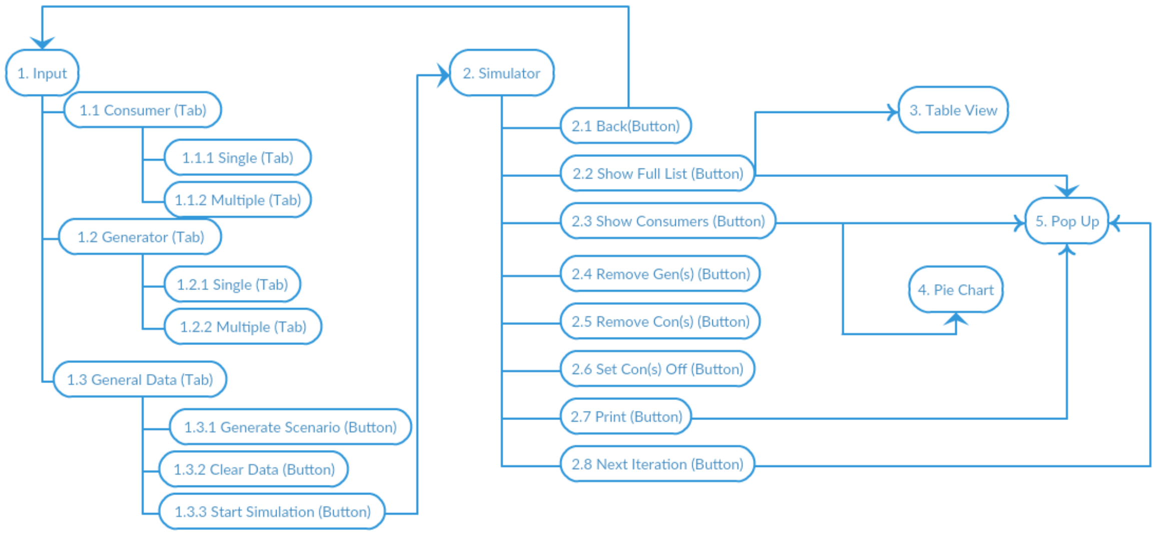

1. Input
1.1 Consumer (Tab)
1.1.1 Single (Tab)
1.1.2 Multiple (Tab)
1.2 Generator (Tab)
1.2.1 Single (Tab)
1.2.2 Multiple (Tab)
1.3 General Data (Tab)
1.3.1 Generate Scenario (Button)
1.3.2 Clear Data (Button)
1.3.3 Start Simulation (Button)
2. Simulator
2.1 Back(Button)
2.2 Show Full List (Button)
2.3 Show Consumers (Button)
2.4 Remove Gen(s) (Button)
2.5 Remove Con(s) (Button)
2.6 Set Con(s) Off (Button)
2.7 Print (Button)
2.8 Next Iteration (Button)
3. Table View
4. Pie Chart
5. Pop Up

Figure 3.6: Structure Diagram

The first screen that appears when application runs is the **Input** Screen, which allows user to generate data for simulation. Generated data will also be showed in a table. User is able to use the generated data to start simulation in the **Simulation** screen that also interacts with **Pop Up** screen whenever certain conditions are met, **Pie Chart** screen and **Table View** screen whenever user wants to have detailed information of data of simulation in the current iteration.

Use Cases of GUI Application are shown in the below diagram:

[8]Author: Nguyen The Viet - 9990

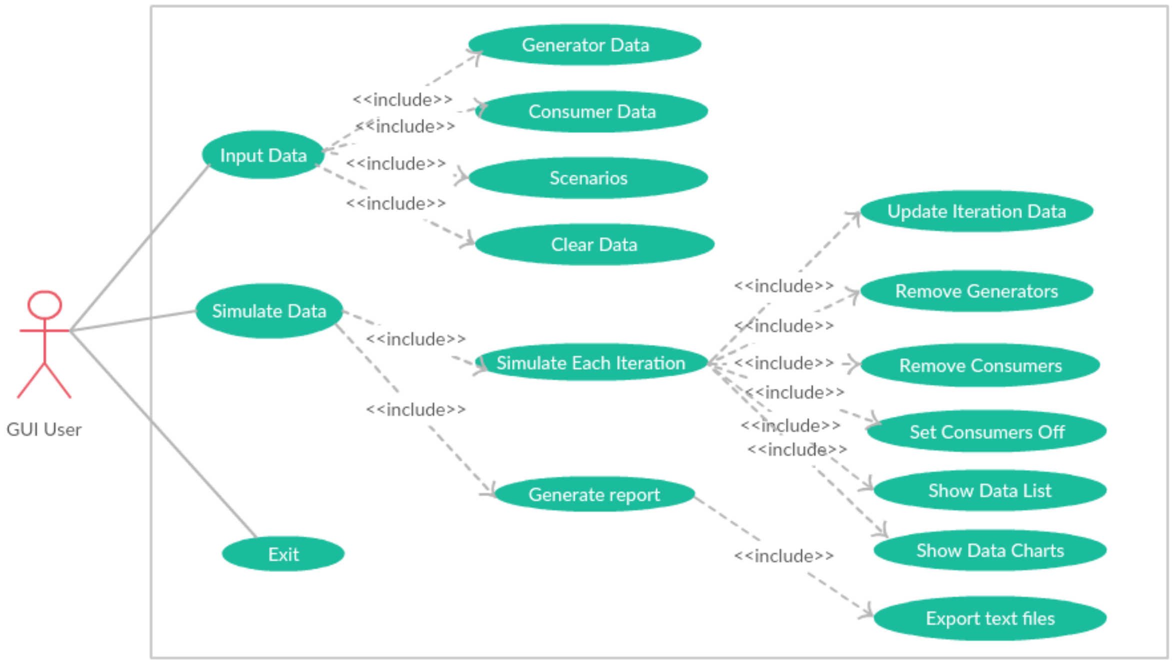


Figure 3.7: Use Case Diagram

Dataflow during simulation is further illustrated in the following activity diagram:

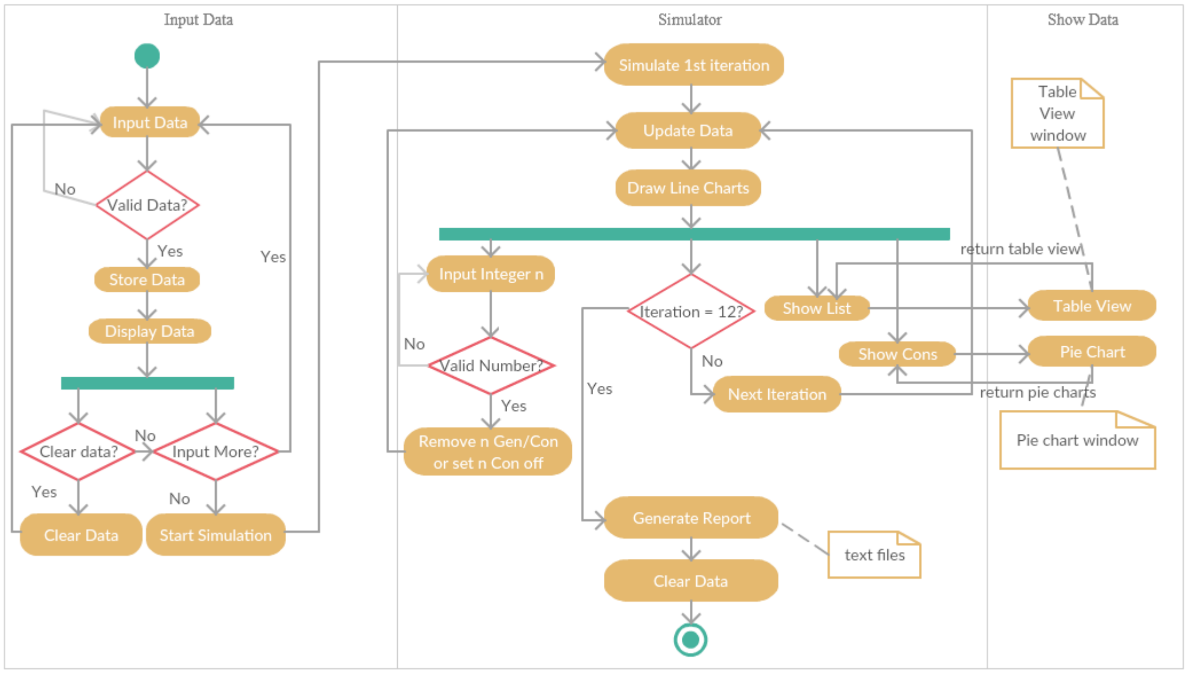


Figure 3.8: GUI Activity Diagram

- **Input Data** is where user is able to generate data multiple times or clear all data during input section. When the process is finished, the data will be used for simulation.

- **Simulator** is where data is used for simulating in twelve iterations. In each iteration, data is updated and can be showed in charts and tables. Concurrently, user is able to remove a certain number of generators or consumers depending on the requirement of scenarios. Setting off a random number of consumers function is added during the process. The algorithm will be discussed further in the later sections. Changes on data will be updated and will affect the next iterative simulations. After finishing all simulations, detailed reports of consumers and generators (numbers, total demand, total supply, total frequency, power distribution, etc.) in each iteration will be printed out, and data will be cleared.

- **Show Data** returns table view or chart view on the input data of the current iterative simulation.

Overall, the structure of the GUI and dataflow are relatively simple. Detail of individual screens and GUI use cases will be discussed in the later subsections.

3.5.3 Screen Description[9]

This section contains major part of the Graphical User Interface of the project Simulation of Power Grids, which includes detailed description of individual screen and its components. I would like to thank my colleague Hoang Thai Duong for helping me design the initial input screen with the logic code for retrieving data from user.

3.5.3.1 Reading Input[10]

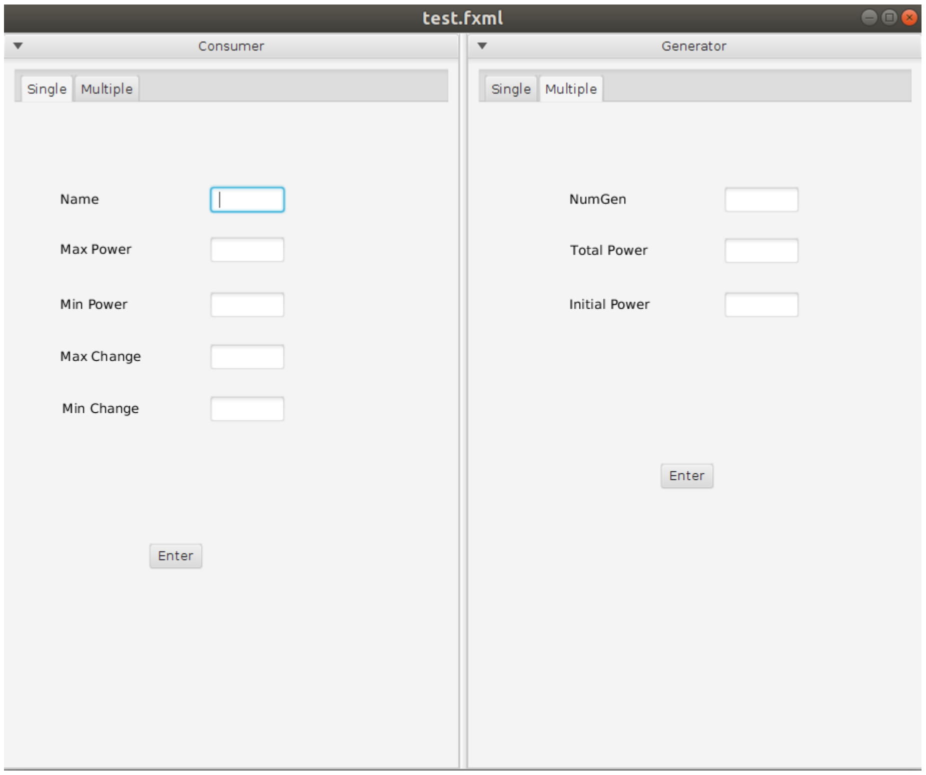


Figure 3.9: GUI for reading input.

I came up with the idea of splitting the main window into two tabs. Left tab is

[9]Author: Nguyen The Viet - 9990
[10]Author: Hoang Thai Duong

”Consumer” and right tab is ”Generator”. Each tab has its own two sub tabs ”Single” and ”Multiple”. There were logical functions implemented to check for invalid input values. If everything is valid, input values are stored and then all text fields are clear for the next input. Otherwise, a label of invalid input will be displayed and all text fields are maintained to be reviewed until they become valid. This layout is a template, it will be modified by my colleague to adapt with new changes.

3.5.3.2 Complete Input Screen [11]

The Input Window is created by Java Scene Builder as fxml file (InputData.fxml), which is controlled by logic code of java class (InputController.java)
The Input Screen is the main screen that appears to the user. It allows user to generate data used for the Simulator in two ways: manually creating input based on interest and automatically generating data based on two scenarios of the project.

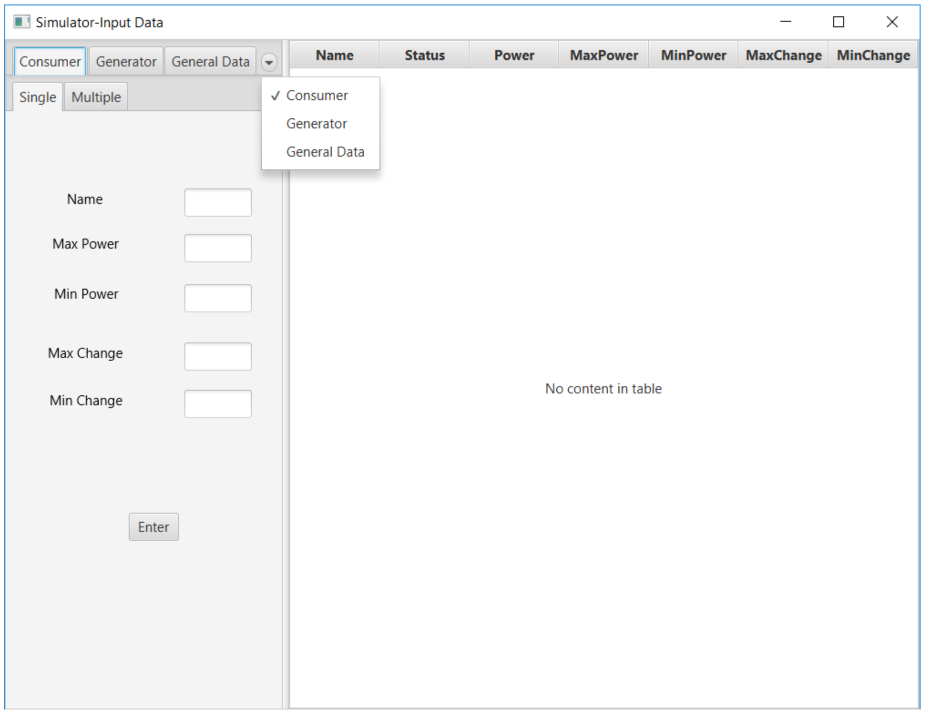


Figure 3.10: Input Screen

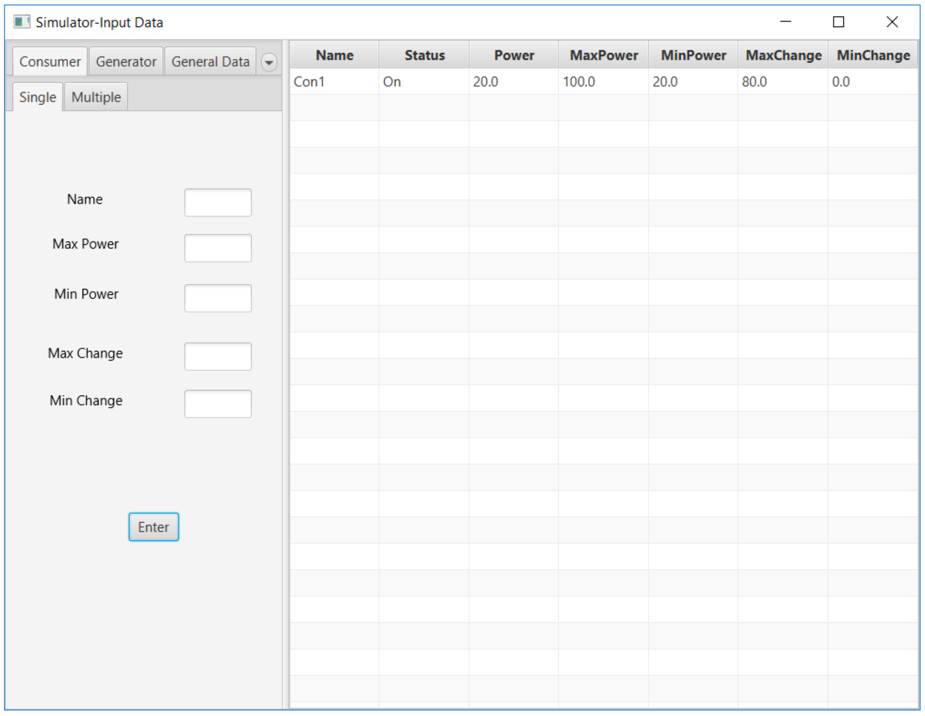
The Input screen has three main tabs: **Consumer, Generator** and **General Data**. Furthermore, both **Consumer Tab** and **Generator Tab** have two subtabs: **Single** and **Multiple**. The **General Tab** have three buttons with their corresponding functions which will be discussed in this section.

[11] Author: Nguyen The Viet - 9990

3.5.3.2.1 Consumer[12]

There are two tabs in the Consumer section. One is used for creating a single consumer with its specific characteristics, and the other is used for creating multiple consumers with total power and deviation.

Single Consumer



Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Con1	On	20.0	100.0	20.0	80.0	0.0

Figure 3.11: Single Consumer Input

Description

User is able to input data for one consumer in this tab. User must fill in all text fields with appropriate values. Otherwise application prompts ”invalid data” and requires user to re-input data again. Additionally, name of consumer cannot be an empty string. The other values which are

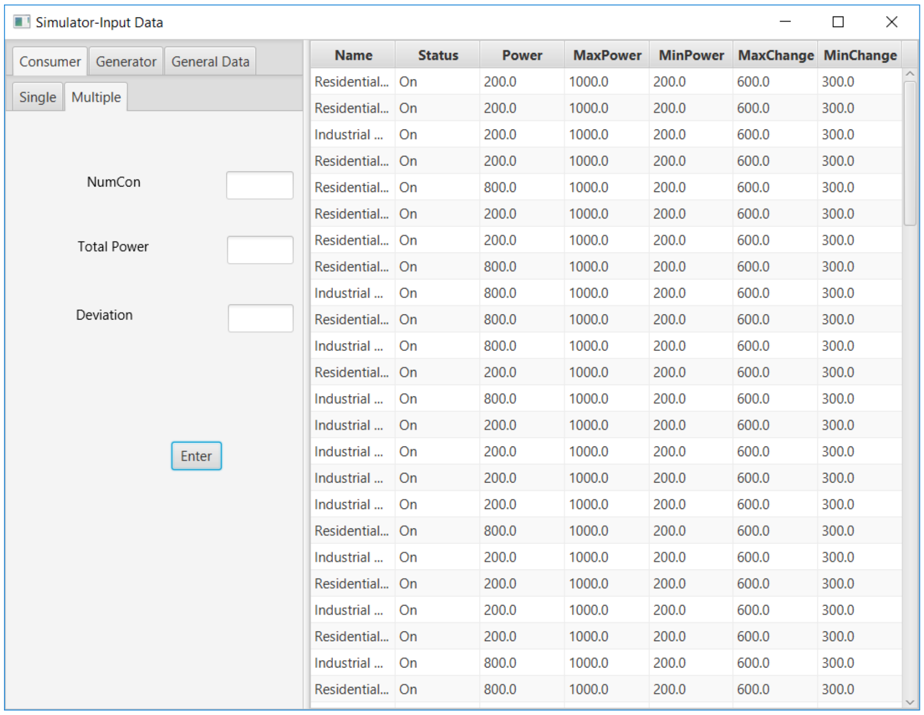
[12]Author: Nguyen The Viet - 9990

non-negative are converted into Double Type after the application receives correct inputs.

Elements

- **Texts:** *Name, Max Power, Min Power, Max Change, Min Change*: Descriptions of text fields.
- **TextFields:** *ConName, ConMaxPower, ConMinPower, ConMaxChange, ConMinChange*: User inputs data into the fields based on the text description of each field.
- **Button:** *Enter*: User submits data after filling in all text fields with appropriate values by clicking **Enter** button. Entered values are displayed on the table on the right.
- **Label:** *singleConLabel*: Prompts error messages if invalid input is submitted.

Multiple Consumers



Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	800.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	800.0	1000.0	200.0	600.0	300.0
Industrial ...	On	800.0	1000.0	200.0	600.0	300.0
Residential...	On	800.0	1000.0	200.0	600.0	300.0
Industrial ...	On	800.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	800.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	800.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	200.0	1000.0	200.0	600.0	300.0
Residential...	On	200.0	1000.0	200.0	600.0	300.0
Industrial ...	On	800.0	1000.0	200.0	600.0	300.0
Residential...	On	800.0	1000.0	200.0	600.0	300.0

Figure 3.12: Multiple Consumers Input

Description

User is able to generate multiple consumers by entering a number (integer) of consumers and their total power and deviation. Just like Single

Consumer Tab, user must fill in all appropriate values before submitting. Deviation and total power are converted into Double Type, which are non-negative. Additionally, deviation input must not exceed the total power input.

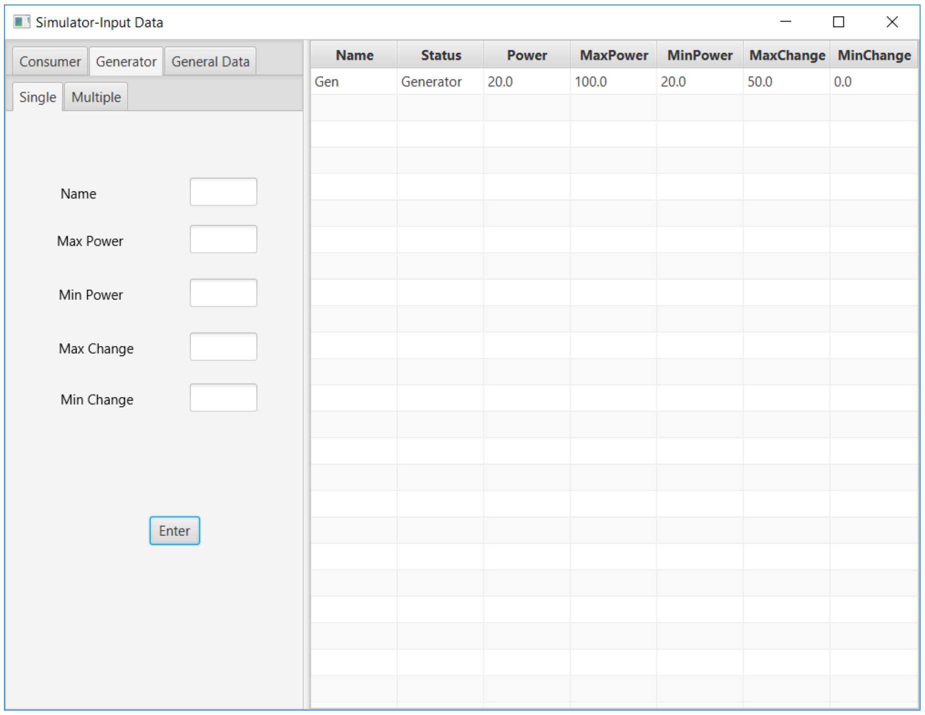
Elements

- **Texts:** *NumCon, TotalPower, Deviation*: Descriptions of text fields.
- **TextFields:** *numCon, totalConPower, deviation*: User inputs data into the fields based on the text description of each field.
- **Button:** *Enter*: User submits data after filling in all text fields with appropriate values by clicking **Enter** button. Entered values are displayed on the table on the right.
- **Label:** *multiConLabel*: Prompts error messages if invalid input is submitted.

3.5.3.2.2 Generator[13]

Likewise, there are two tabs in the Generator section. One is used for creating a single generator with its specific characteristics, and the other is used for creating multiple generators with total power and initial power.

Single Generator



Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Gen	Generator	20.0	100.0	20.0	50.0	0.0

Figure 3.13: Single Generator Input

Description

User is able to input data of a single generator just like in the Single Consumer Tab above. Name of generator cannot be empty, and other values which are non-negative are converted into Double Type after the

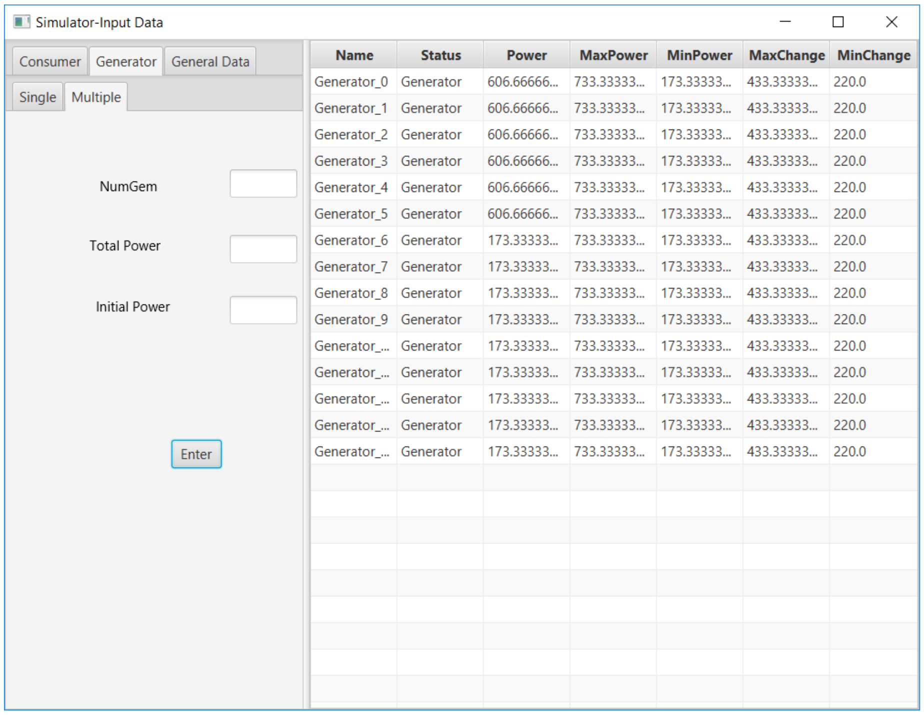
[13]Author: Nguyen The Viet - 9990

application receives correct inputs. Initial power input must be smaller than total power input.

Elements

- **Texts:** *Name, Max Power, Min Power, Max Change, Min Change*: Descriptions of text fields
- **TextFields:** *GenName, GenMaxPower, GenMinPower, GenMax-Change, GenMinChange*: User inputs data into the fields based on the text description of each field.
- **Button:** *Enter*: User submits data after filling in all text fields with appropriate values by clicking **Enter** button. Entered values are displayed on the table on the right.
- **Label:** *singleGenLabel*: Prompts error messages if invalid input is submitted.

Multiple Generators



Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Generator_0	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_1	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_2	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_3	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_4	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_5	Generator	606.66666...	733.33333...	173.33333...	433.33333...	220.0
Generator_6	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_7	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_8	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_9	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_...	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_...	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_...	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_...	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0
Generator_...	Generator	173.33333...	733.33333...	173.33333...	433.33333...	220.0

Figure 3.14: Multiple Generators Input

Description

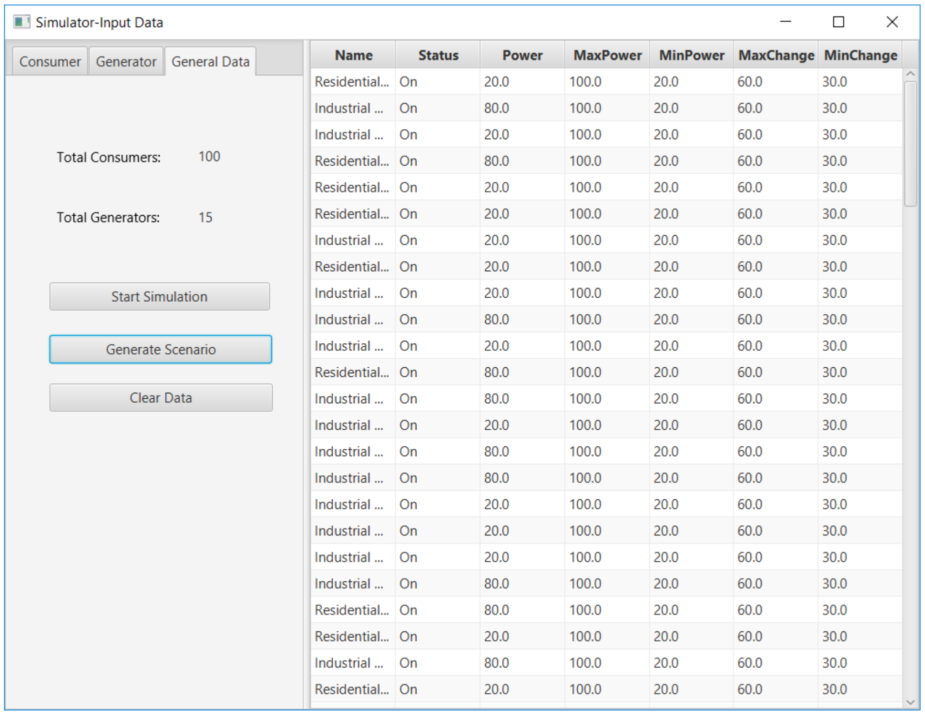
User is able to generate multiple generators by entering a number (integer) of generators and their total power and initial power. Just like

Single Generator Tab, user must fill in all appropriate values before submitting. Deviation and total power are converted into Double Type, which are non-negative. Additionally, initial power input must not exceed the total power input.

Elements

- **Texts:** *NumGen, TotalPower, Initial Power*: Descriptions of text fields.
- **TextFields:** *numGen, totalGenPower, initialPower*: User inputs data into the fields based on the text description of each field.
- **Button:** *Enter*: User submits data after filling in all text fields with appropriate values by clicking **Enter** button. Entered values are displayed on the table on the right.
- **Label:** *multipleGenLabel*: Prompts error messages if invalid input is submitted.

3.5.3.2.3 General Data[14]



Total Consumers: 100

Total Generators: 15

Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Residential...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Residential...	On	80.0	100.0	20.0	60.0	30.0
Residential...	On	20.0	100.0	20.0	60.0	30.0
Residential...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Residential...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Residential...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Residential...	On	80.0	100.0	20.0	60.0	30.0
Residential...	On	20.0	100.0	20.0	60.0	30.0
Industrial ...	On	80.0	100.0	20.0	60.0	30.0
Residential...	On	20.0	100.0	20.0	60.0	30.0

Figure 3.15: General Data Tab

Description

The Input Window contains General Data Tab which enables user to either clear all generated data, create data based on scenarios of the requirements or start simulation with the generated data. User can also see the total number of consumers and generators already added into the Control of the Simulator. The table on the right displays detailed information of each generator or consumer that has been generated.

Elements

[14] Author: Nguyen The Viet - 9990

- **Texts:** *Total Consumers, Total Generators*: Descriptions of two labels.
- **Labels:** *totalConsumer, totalGenerator*: Display number of consumers and generators respectively if input is created.
- **Button:**
  - *Clear Data*: Clear all data which is contained in the Control. Detailed information in the table on the right will be wiped out.

  - *Generate Scenario*: If the button is clicked, data based on scenarios of the requirement of the project will be generated: 15 generators with total power of 11000, initial power of 5000; 100 consumers with average power consumption of 100W.

  - Start Simulate: If the button is clicked, Input Window is closed and the Simulator Window is opened. User begins the simulation process.

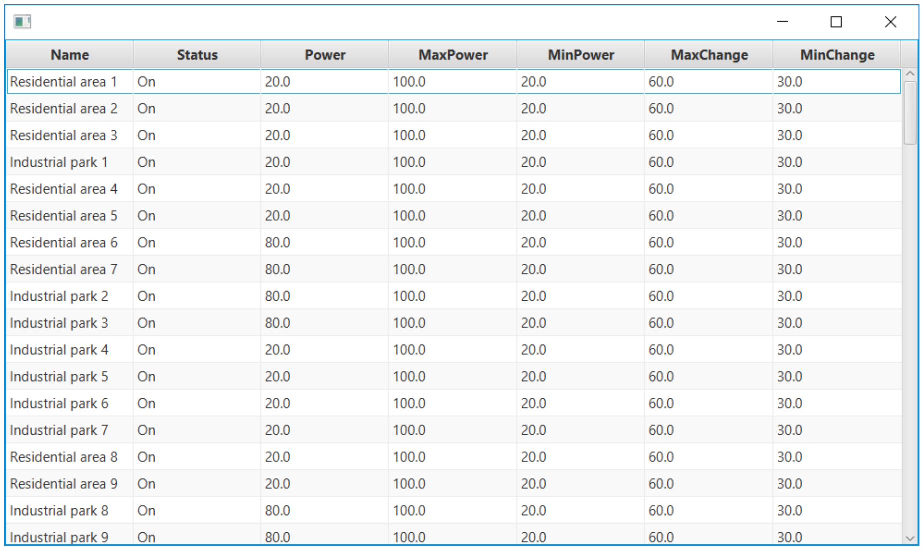
Note: If the *Start Simulate* button is clicked, The Input Controller (InputController.java) which controls the Input Window will call an instance of the Simulator Controller (ChartController.java) which controls the Simulator Window. The code implementation for this is shown below:

Listing 3.21: Calling Controller of Simulator Window

```
/*close the Input window*/
Stage stageStart = (Stage) startButton.getScene().getWindow();
stageStart.close();
/*attempt to open Simulator Window*/
try {
    FXMLLoader loader = new
        FXMLLoader(getClass().getResource("LineChart.fxml"));
    Parent root = (Parent) loader.load();
    /*create and loads instance of Simulator Controller*/
    ChartController chartController = new ChartController();
    chartController = loader.getController();
    /*Retrieve data from user inputs*/
    chartController.addConsumerList(getConsumerList());
    chartController.addGeneratorList(getGeneratorList());
    /*begins first iteration of simulation*/
    chartController.drawGraph();
    /*Open Simulator Window*/
    Stage stage = new Stage();
    stage.setScene(new Scene(root));
    stage.setTitle("Simulator");
    stage.show();
}catch(IOException e) {
    e.printStackTrace();
}
```

3.5.3.3 Table View[15]

The Table View Window (InitialTable.fxml) is controlled by logic code of java class (ShowTable.java). The screen is called whenever the button *Show Full List* in the Simulator Window is clicked.



Name	Status	Power	MaxPower	MinPower	MaxChange	MinChange
Residential area 1	On	20.0	100.0	20.0	60.0	30.0
Residential area 2	On	20.0	100.0	20.0	60.0	30.0
Residential area 3	On	20.0	100.0	20.0	60.0	30.0
Industrial park 1	On	20.0	100.0	20.0	60.0	30.0
Residential area 4	On	20.0	100.0	20.0	60.0	30.0
Residential area 5	On	20.0	100.0	20.0	60.0	30.0
Residential area 6	On	80.0	100.0	20.0	60.0	30.0
Residential area 7	On	80.0	100.0	20.0	60.0	30.0
Industrial park 2	On	80.0	100.0	20.0	60.0	30.0
Industrial park 3	On	80.0	100.0	20.0	60.0	30.0
Industrial park 4	On	20.0	100.0	20.0	60.0	30.0
Industrial park 5	On	20.0	100.0	20.0	60.0	30.0
Industrial park 6	On	20.0	100.0	20.0	60.0	30.0
Industrial park 7	On	20.0	100.0	20.0	60.0	30.0
Residential area 8	On	20.0	100.0	20.0	60.0	30.0
Residential area 9	On	20.0	100.0	20.0	60.0	30.0
Industrial park 8	On	80.0	100.0	20.0	60.0	30.0
Industrial park 9	On	80.0	100.0	20.0	60.0	30.0

Figure 3.16: Table View

Description

Table Window displays detailed information of each consumer or generator that is in the Simulator.

Elements

- **Table View** *table*: Contains columns that show individual information of each generator or consumer.
- **Table Columns:** *name, power, status, maxPower, minPower, maxChange, minChange*: Displays values of each generator or consumer.

Code Implementation

[15] Author: Nguyen The Viet - 9990

Firstly, columns of the table must be initialized. The controller has one global variable **control** of type vgu.control.Control.

Listing 3.22: Initializing Table

```
name.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,String>("Name"));
power.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,Double>("Power"));
maxPower.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,Double>("MaxPower"));
minPower.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,Double>("MinPower"));
maxChange.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,Double>("MaxChange"));
minChange.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,Double>("MinChange"));
status.setCellValueFactory(new
    PropertyValueFactory<AbstractComponent,String>("Status"));
```

Secondly, a method is created to retrieve data for the **control** variable:

Listing 3.23: Retrieve Data

```
public void retrieveData(Control control) {
    this.control = control;
}
```

Thirdly, a method is created to pass **consumers** and **generators** lists from the **control** to an ObservableList. The reason for doing this is because in order to display data onto the table, method **tableView.getItems().addAll(list)** is called, where the parameter **list** must be an ObservableList instance.

Listing 3.24: Create Observable List

```
private ObservableList<AbstractComponent> getUserList(){
    for(AbstractComponent i: control.getConsumers()) {
        myList.add(i);
    }
    for(AbstractComponent i: control.getGenerators()) {
        myList.add(i);
    }
    ObservableList<AbstractComponent> list =
        FXCollections.observableArrayList(myList);
    return list;
}
```

Finally, data can be displayed onto the table by the following logic code:

Listing 3.25: Create Observable List

```
ObservableList<AbstractComponent> list = getUserList();
tableView.getItems().addAll(list);
```

How the Table View Window is called during simulation will be discussed later in the Simulator Window.

3.5.3.4 Pie Chart[16]

As we have implemented two new types of consumers, namely **Residential** consumer and **Industrial** consumer, the GUI application is extended to adapt changes by implementing the Pie Chart Window (PieChart.fxml), which is controlled by logic code of java class (PieChartController.java).

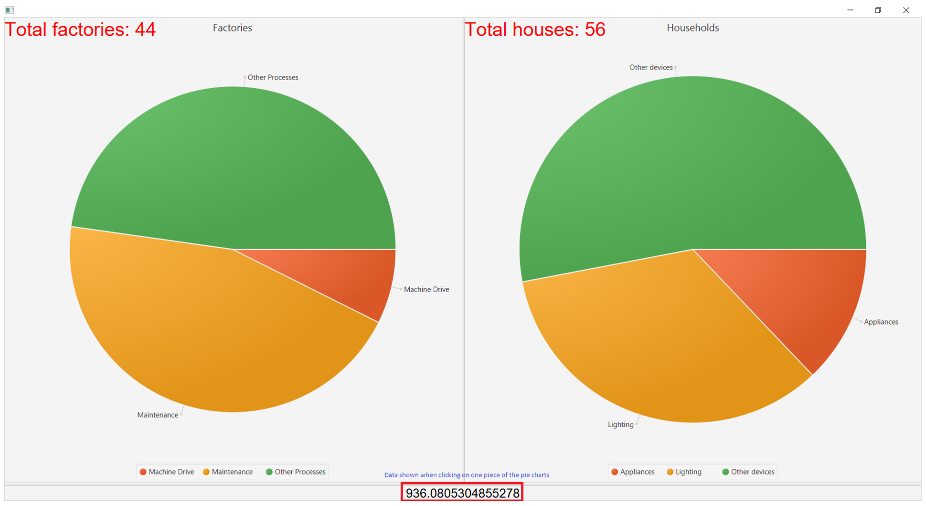

Total factories: 44
Factories
Other Processes
Machine Drive
Maintenance
● Machine Drive ● Maintenance ● Other Processes
Total houses: 56
Households
Other devices
Appliances
Lighting
● Appliances ● Lighting ● Other devices
Data shown when clicking on one piece of the pie charts
936.0805304855278

Figure 3.17: Pie Chart Of Consumers

Description

The Pie Chart Window shows two different pie charts of Residential consumers and Industrial consumers. The main purpose of this screen is to show the comparison of power distribution in several activities between two type of consumers in each iteration of simulation.

Elements

- **Label** *factoryLabel*: Shows number of Industrial consumers.
- **Pie Chart** *factoryChart*: Shows power distribution of Industrial consumers. The three main activities of this consumer type are *Machine Drive, Maintenance and Other Processes.*
- **Label** *householdLabel*: Shows number of Residential consumers.

[16]Author: Nguyen The Viet - 9990

- **Pie Chart** *houseChart*: Shows power distribution of Residential consumers. The three main activities of this consumer type are *Appliances, Lighting and Other Devices*.
- **Label** *caption*: Shows power value when user clicks onto a particular piece of the pie charts at the bottom of the screen.

Code Implementation

Just like in **Table View Window**, the controller has a global variable **control** of type vgu.control.Control and requires lists of ObservableList type in order to show data on the pie charts.
Firstly, data must be retrieve for the **control**, then two lists of Residential Consumers and Industrial Consumers will be created:

Listing 3.26: Get Observable Lists

```
/*Get Industrial Consumers*/
private ObservableList<Data> getIndustries() {
    ObservableList<Data> inList =
        FXCollections.observableArrayList();
    inList.add(new PieChart.Data("Machine
        Drive",control.getTotalMachinDrive()));
    inList.add(new
        PieChart.Data("Maintenance",control.getTotalMaintenance()));
    inList.add(new PieChart.Data("Other
        Processes",control.getTotalOtherProcesses()));
    return inList;
}
/*Get Residential Consumers*/
private ObservableList<Data> getResidents() {
    ObservableList<Data> reList =
        FXCollections.observableArrayList();
    reList.add(new
        PieChart.Data("Appliances",control.getTotalAppliances()));
    reList.add(new
        PieChart.Data("Lighting",control.getTotalLighting()));
    reList.add(new PieChart.Data("Other
        devices",control.getTotalOtherUses()));
    return reList;
}
```

Another method is created to handle Mouse Event from user. Whenever user clicks on a piece of a pie chart, its value will be displayed on the bottom of the screen:

Listing 3.27: Display Data On Mouse Pressed

```
public void noteOnData() {
    /*Handle mouse pressed for industrial chart*/
    for (final PieChart.Data data1 : factoryChart.getData()) {
        data1.getNode().addEventHandler(MouseEvent.MOUSE_PRESSED,
             new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent e) {
                caption.setText(String.valueOf(data1.getPieValue()));
            }
        });
    }
/*Handle mouse pressed for residential chart*/
    for (final PieChart.Data data2 : houseChart.getData()) {
        caption.toFront();
        caption.setTextFill(Color.BLACK);
        caption.setStyle("-fx-font: 26 arial;");
        data2.getNode().addEventHandler(MouseEvent.MOUSE_PRESSED,
             new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent e) {
                caption.setText(String.valueOf(data2.getPieValue()));
            }
        });
    }
}
```

Finally, data is displayed by calling the following logic code:

Listing 3.28: Display Data

```
      factoryChart.setData(getIndustries());
      houseChart.setData(getResidents());
      /*handle pressed mouse event*/
      noteOnData();
```

How the Pie Chart Window is called during simulation will also be discussed later in the Simulator Window.

Note: Power is distributed according to the percentage table in section **3.3.2. Additional Implementation** of my colleague.

3.5.4 Pop Up[17]

The Pop Up Window (PopUp.fxml) is controlled by logic code of java class (PopUpController.java).

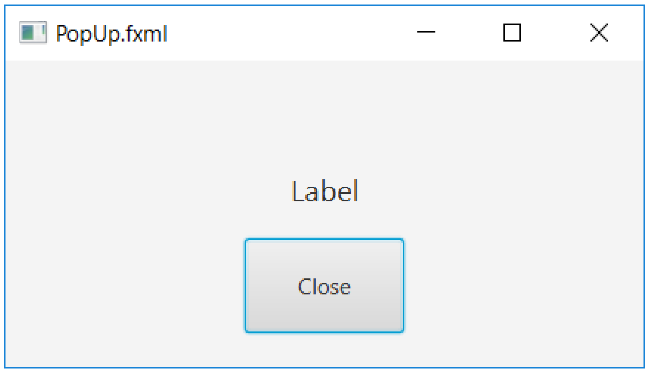


Figure 3.18: PopUp Window

Description

Pop Up Window shows a message when a condition of an event of a particular button is met.

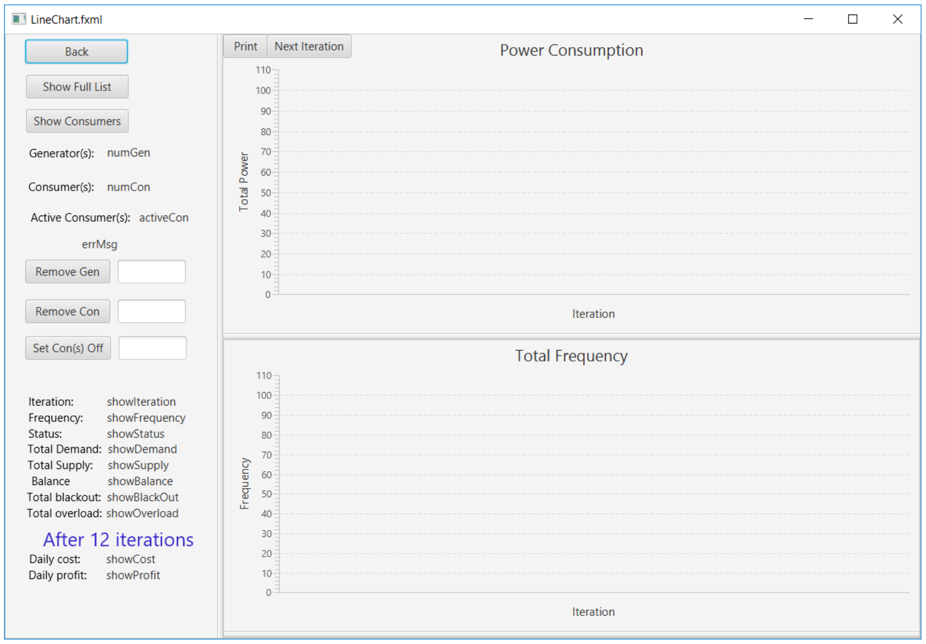
Elements

- **Label:** *text*: Displays a message in String format.
- **Button:** *Close*: Window is closed if the button is clicked

[17] Author: Nguyen The Viet - 9990

3.5.5 Simulator[18]

The Simulator Window (LineChart.fxml) is controlled by logic code of java class (LineChartController.java).



Generator(s):	numGen
Consumer(s):	numCon
Active Consumer(s):	activeCon
Iteration:	showIteration
Frequency:	showFrequency
Status:	showStatus
Total Demand:	showDemand
Total Supply:	showSupply
Balance	showBalance
Total blackout:	showBlackOut
Total overload:	showOverload
Daily cost:	showCost
Daily profit:	showProfit

Figure 3.19: Simulator

Description

The Simulator Window appears after the user clicks the button **Start Simulation** in the **Input Window**. This is the main screen that performs simulation of twelve iterations according to the requirements of the project. User is able to control each iteration by removing generators or consumers, turning a number of consumers off, showing data in table view and pie charts and seeing all the detailed information of the simulation. Additionally, user can generate reports after finishing the simulation.

Elements

[18] Author: Nguyen The Viet - 9990

- **Text - Label**
  - ***Generator(s) - numGen***: Shows number of generators in the current iteration.
  - ***Consumer(s) - numCon***: Shows number of consumers in the current iteration.
  - ***Actice Consumer(s) - activeCon***: Shows number of active consumers in the current iteration.
  - ***Iteration - showIteration***: Shows current iteration of the simulation.
  - ***Frequency - showFrequency***: Shows total frequency in current iteration.
  - ***Status - showStatus***: Shows the status of the simulation in the current iteration.
  - ***Total Demand - showDemand***: Shows the total demand of consumers in the current iteration.
  - ***Total Supply - totalSupply***: Shows the total supply of generators in the current iteration.
  - ***Balance - showBalance***: Shows total balance in the current iteration.
  - ***Daily cost - showCost***: Shows total cost after twelve iterations of simulation
  - ***Daily profit - showProfit***: Shows total profit after twelve iterations of simulation
  - ***Total blackout - showBlackOut***: Shows number of blackouts. If blackout occurs in three consecutive iterations, the status is changed to **defect** and all generators and consumers are removed from the simulator.
  - ***Total overload - showOverload***: Shows number of overloads. If overload occurs in three consecutive iterations, the status is also changed to **defect** and all generators and consumers are removed from the simulator.
- **Button**
  - ***Back***: Simulator Window is closed and the Input Window is opened again if the button is clicked. The Simulator Controller closes the window and calls an instance of Input Controller.
  - ***Show Full List***: Shows detailed information of consumers and generators in the current iteration by calling an instance of Table View Controller.
  - ***Show Consumers***: Shows power distributions of two types of consumer in two pie charts by calling an instance of Pie Chart Controller.
  - ***Next Iteration***: Simulates the next iteration (increasing by one for each click). Every time the button is clicked, values of the two

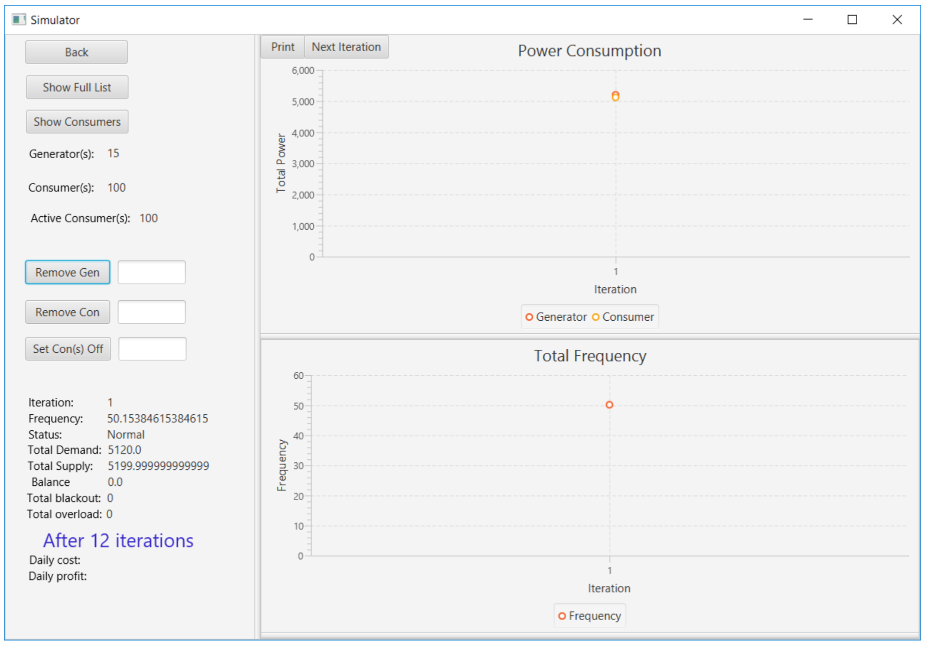
line charts of Power Consumption and Total Frequency will be updated and shown onto the GUI. Furthermore, all data which is displayed by the Labels will also be updated according to the iteration. If the button is clicked without data in the lists or the Simulator has completed all twelve iterations, a pop up screen will be shown with an error message.

  - ***Print***: Exports reports and clears all the data after simulation completes. If the button is clicked without data in the lists or during simulation, a pop up screen will be shown with an error message. User should click the button **Back** to generate new input before starting simulation again.

- **Button - Text Field**
  - ***Remove Gen - iRemGen***: User is able to input a non-negative integer n which is smaller than the size of generators list. After the button is clicked, n generators will be removed from the simulator starting from the beginning of the list.
  - ***Remove Con - iRemCon***: Same as ***Remove Gen - iRemGen***, but for consumers.
  - ***Set Con(s) Off - iSetOff***: User is able to input a non-negative integer n which is smaller than the size of consumer list. After the button is clicked, n random consumers in the list will be turned off, which means their power is set to zero and status is set to Off

1. Normal Scenario Demonstration

Assume there are 100 Consumers with average power consumption of 100W and 15 Generators with total power of 11000 and initial power of 5000 according to the scenario of the project in the **Input Window**, user clicks the button **Start Simulate** to jump to the **Simulator Window**. The first iteration is described in the following picture:



Generator(s): 15

Consumer(s): 100

Active Consumer(s): 100

```
Iteration:      1
Frequency:      50.15384615384615
Status:         Normal
Total Demand:   5120.0
Total Supply:   5199.999999999999
Balance         0.0
Total blackout: 0
Total overload: 0
```

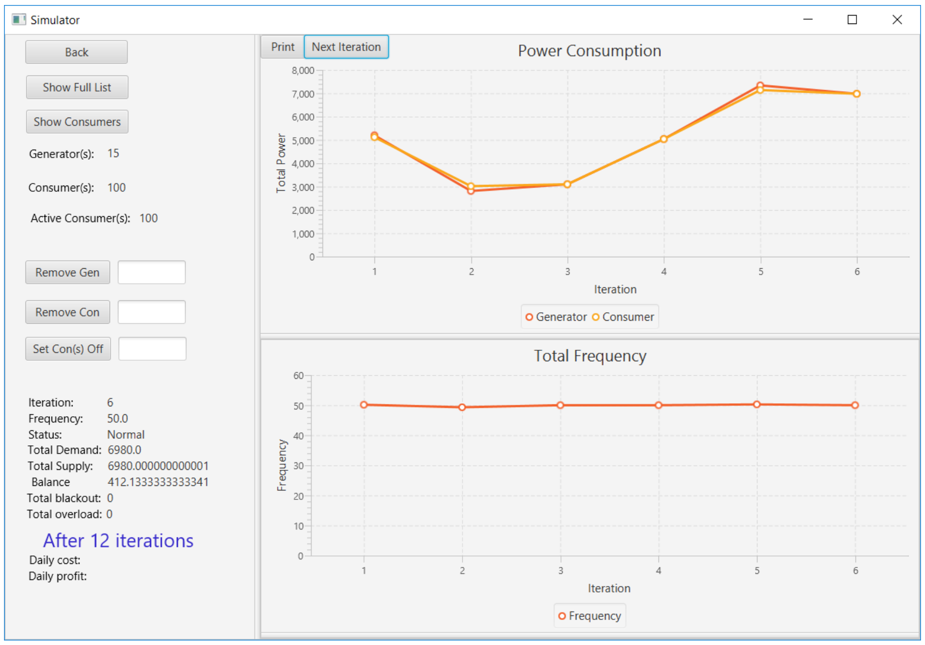
After 12 iterations

Daily cost:

Daily profit:

Figure 3.20: First iteration

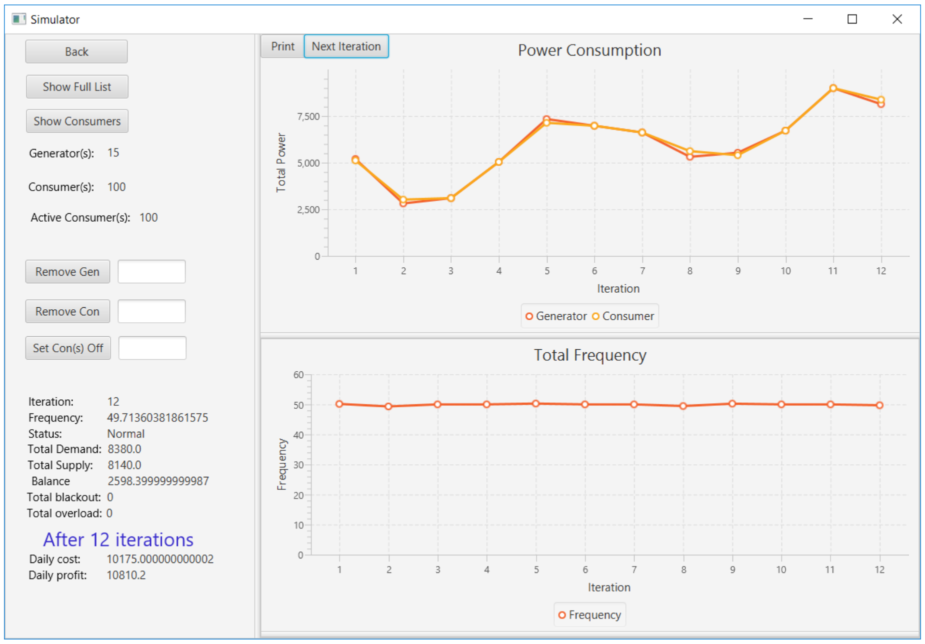
In each iteration, user can click **Show Full List** or **Show Consumers** to view detailed information of consumers and generators which are currently in the Simulator. After clicking **Next Iteration** button for five more times (sixth iteration), the data is updated:



```
Generator(s):   15

Consumer(s):    100

Active Consumer(s):   100

Iteration:        6
Frequency:        50.0
Status:           Normal
Total Demand:     6980.0
Total Supply:     6980.000000000001
 Balance          412.1333333333341
Total blackout:   0
Total overload:   0

    After 12 iterations
Daily cost:
Daily profit:
```

Figure 3.21: Sixth iteration

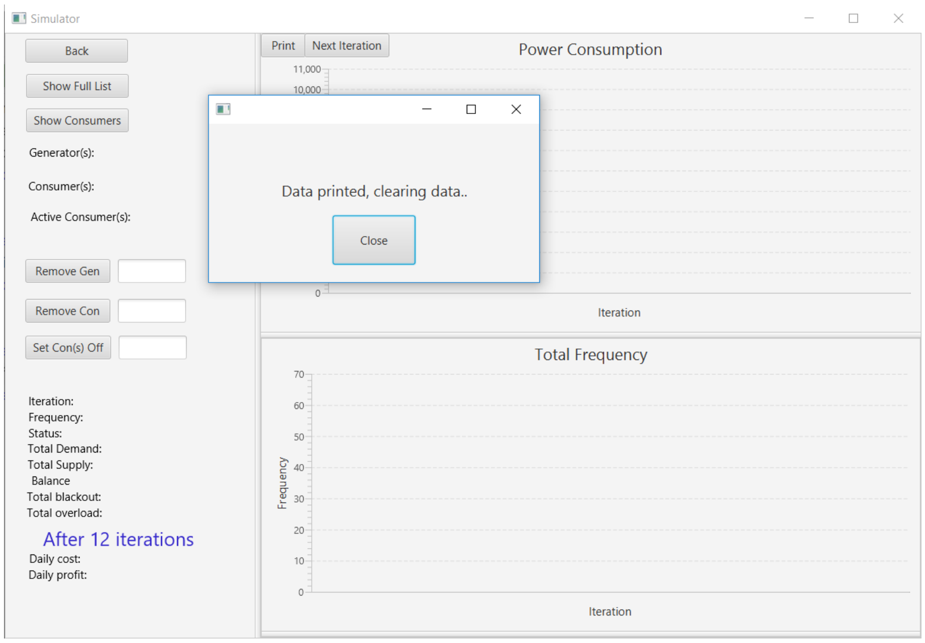
Finally at twelfth iteration:



```
Generator(s):   15

Consumer(s):    100

Active Consumer(s):   100

Iteration:        12
Frequency:        49.71360381861575
Status:           Normal
Total Demand:     8380.0
Total Supply:     8140.0
 Balance          2598.399999999987
Total blackout:   0
Total overload:   0

    After 12 iterations
Daily cost:       10175.000000000002
Daily profit:     10810.2
```

Figure 3.22: Twelfth iteration

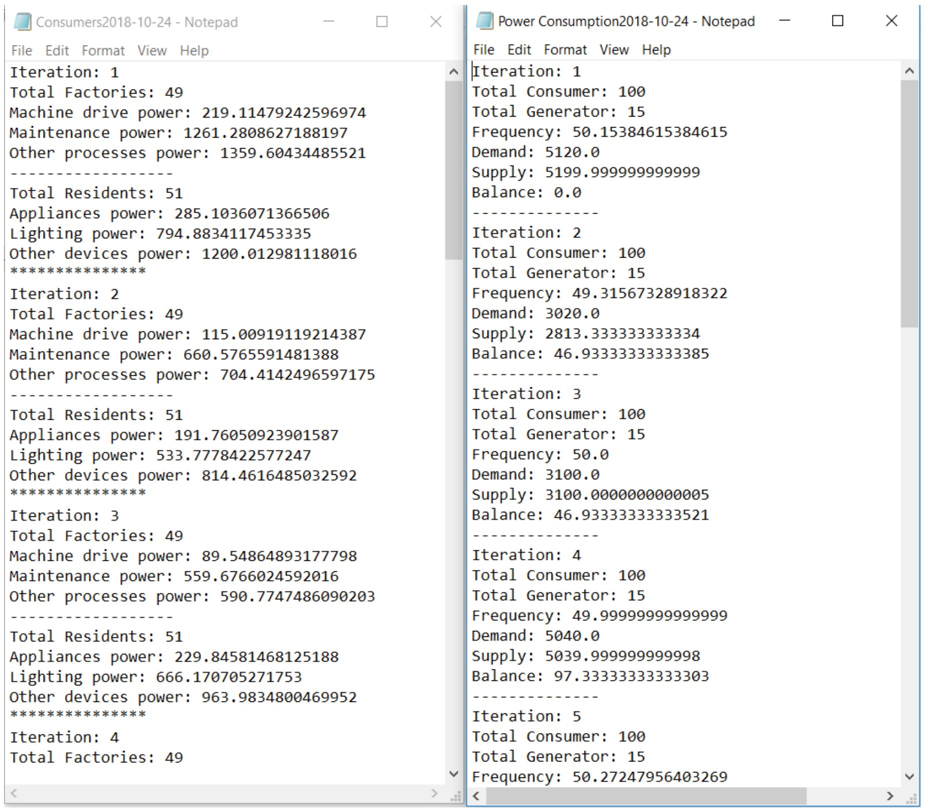
If user clicks on the button **Print**, data will be cleared and text files will be exported.



Data printed, clearing data..

Figure 3.23: Data Printed And Cleared

Two reports are generated: Power consumption of consumers & power distribution of generators report and Power distribution between two types of consumer report.



```
Iteration: 1
Total Factories: 49
Machine drive power: 219.11479242596974
Maintenance power: 1261.2808627188197
Other processes power: 1359.60434485521
------------------
Total Residents: 51
Appliances power: 285.1036071366506
Lighting power: 794.8834117453335
Other devices power: 1200.012981118016
***************
Iteration: 2
Total Factories: 49
Machine drive power: 115.00919119214387
Maintenance power: 660.5765591481388
Other processes power: 704.4142496597175
------------------
Total Residents: 51
Appliances power: 191.76050923901587
Lighting power: 533.7778422577247
Other devices power: 814.4616485032592
***************
Iteration: 3
Total Factories: 49
Machine drive power: 89.54864893177798
Maintenance power: 559.6766024592016
Other processes power: 590.7747486090203
------------------
Total Residents: 51
Appliances power: 229.84581468125188
Lighting power: 666.170705271753
Other devices power: 963.9834800469952
***************
Iteration: 4
Total Factories: 49
```

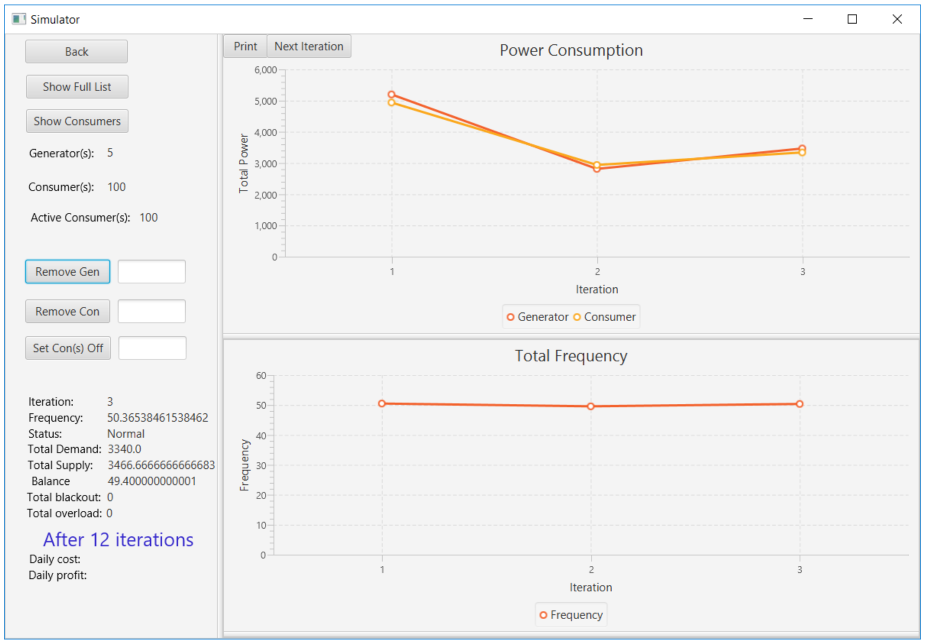
Power Consumption2018-10-24 - Notepad
File Edit Format View Help

```
Iteration: 1
Total Consumer: 100
Total Generator: 15
Frequency: 50.15384615384615
Demand: 5120.0
Supply: 5199.999999999999
Balance: 0.0
--------------
Iteration: 2
Total Consumer: 100
Total Generator: 15
Frequency: 49.31567328918322
Demand: 3020.0
Supply: 2813.333333333334
Balance: 46.93333333333385
--------------
Iteration: 3
Total Consumer: 100
Total Generator: 15
Frequency: 50.0
Demand: 3100.0
Supply: 3100.0000000000005
Balance: 46.93333333333521
--------------
Iteration: 4
Total Consumer: 100
Total Generator: 15
Frequency: 49.99999999999999
Demand: 5040.0
Supply: 5039.999999999998
Balance: 97.33333333333303
--------------
Iteration: 5
Total Consumer: 100
Total Generator: 15
Frequency: 50.27247956403269
```

Figure 3.24: Reports

2. Defect Scenario Demonstration

Assume we have the same data of the normal scenario. At iteration three, user removes 10 generators:



Generator(s): 5

Consumer(s): 100

Active Consumer(s): 100

```
Iteration:      3
Frequency:      50.36538461538462
Status:         Normal
Total Demand:   3340.0
Total Supply:   3466.6666666666683
 Balance        49.400000000001
Total blackout: 0
Total overload: 0
```

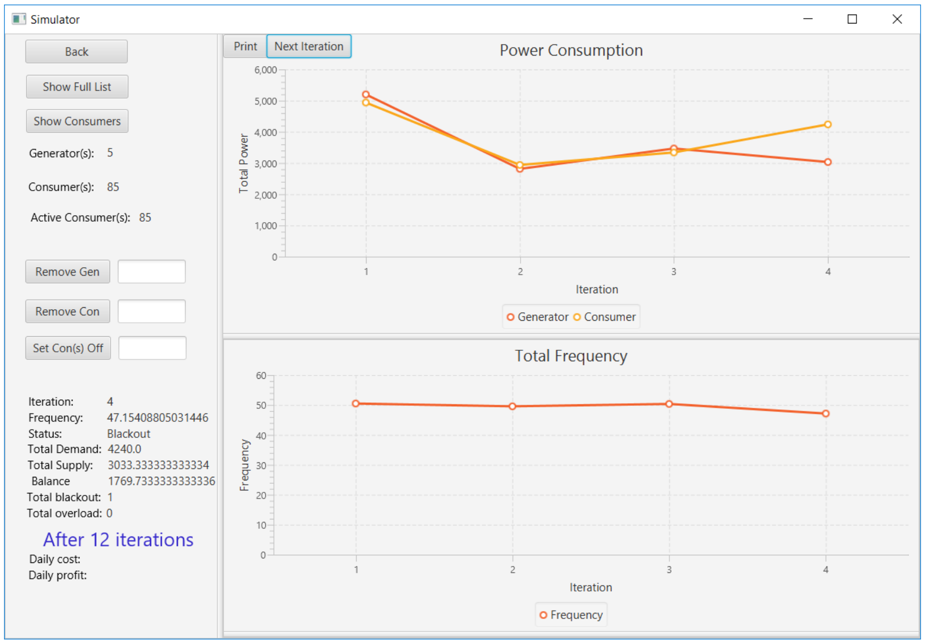
After 12 iterations

Daily cost:

Daily profit:

Figure 3.25: Remove Generators At Iteration 3

In the next iteration, 15% Consumers are removed due to the change of number of generators (blackout).



Generator(s): 5

Consumer(s): 85

Active Consumer(s): 85

```
Iteration:      4
Frequency:      47.15408805031446
Status:         Blackout
Total Demand:   4240.0
Total Supply:   3033.333333333334
 Balance        1769.7333333333336
Total blackout: 1
Total overload: 0
```

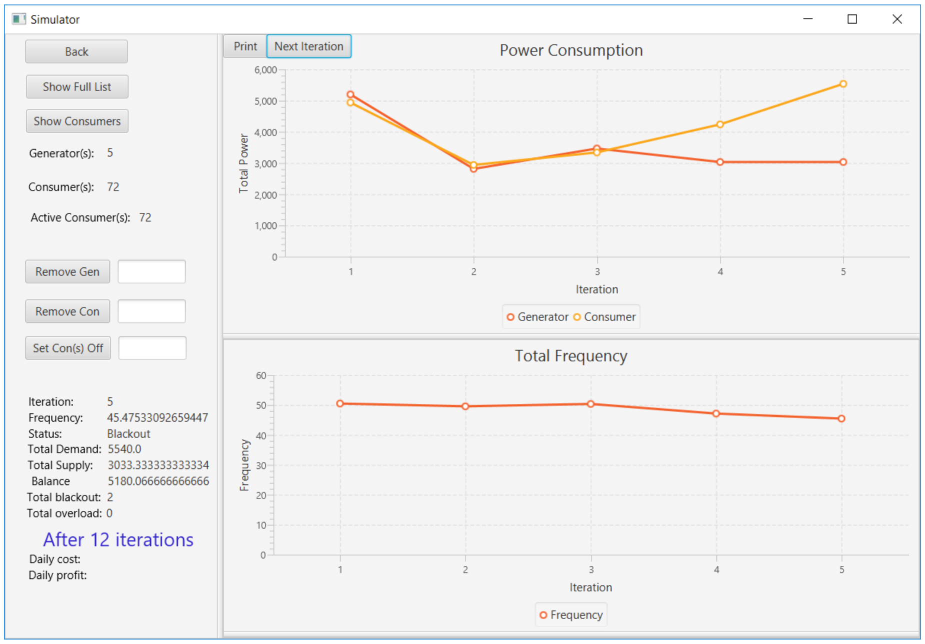
After 12 iterations

Daily cost:

Daily profit:

Figure 3.26: Iteration 4

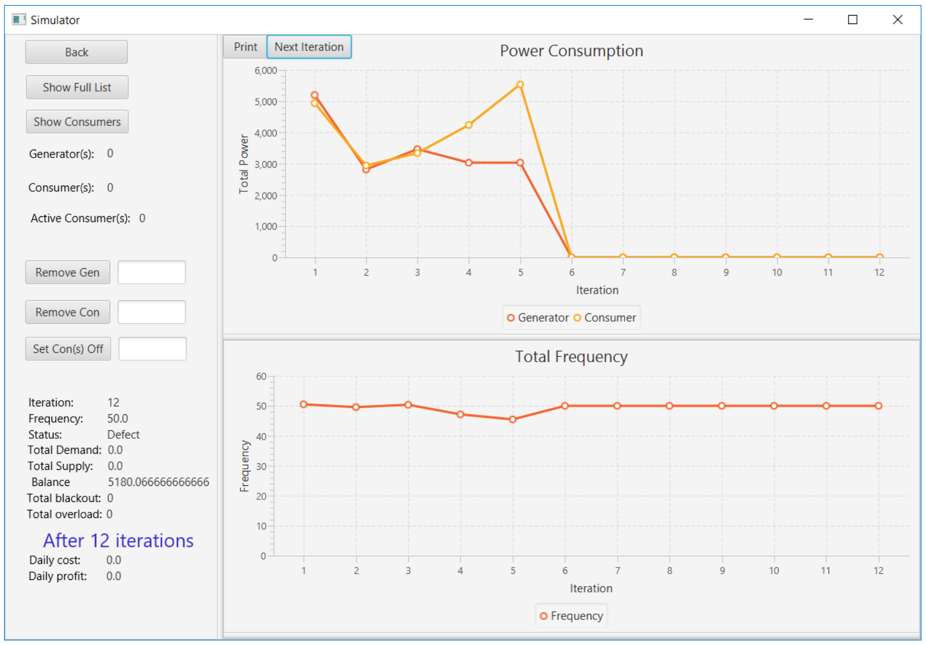
Because the power supply is not enough in the next iteration again, 15% more consumers are removed (blackout).



```
Generator(s):   5
Consumer(s):    72
Active Consumer(s):  72

Iteration:      5
Frequency:      45.47533092659447
Status:         Blackout
Total Demand:   5540.0
Total Supply:   3033.333333333334
Balance         5180.066666666666
Total blackout: 2
Total overload: 0
After 12 iterations
Daily cost:
Daily profit:
```

Figure 3.27: Iteration 5

In the sixth iteration, the power supply is also not enough again, thus causes another blackout. Simulator finally comes to defect state due to three consecutive blackouts.



```
Generator(s):    0
Consumer(s):     0
Active Consumer(s):  0

Iteration:       12
Frequency:       50.0
Status:          Defect
Total Demand:    0.0
Total Supply:    0.0
 Balance         5180.066666666666
Total blackout:  0
Total overload:  0
After 12 iterations
Daily cost:      0.0
Daily profit:    0.0
```

Figure 3.28: Defect State

Code Implementation

Since code is submitted along with the project, this document only illustrates some important logic of the implementation.

1. **BACK BUTTON**

Listing 3.29: Open Input Window

```
/*Get fxml file*/
FXMLLoader loader = new
    FXMLLoader(getClass().getResource("InputData.fxml"));
Parent root = (Parent) loader.load();
/*Create input controller instance*/
InputController inputController = new InputController();
/*Open new window*/
Stage Inputstage = new Stage();
Inputstage.setScene(new Scene(root));
Inputstage.show();
```

2. **SHOW FULL LIST BUTTON**

Listing 3.30: Open Table View

```
/*If consumers and generators lists are empty, call an instance of
    Pop-up controller*/
PopUpController popUp = new PopUpController();
popup.setPopUp("No data to show!");
popUp.setText();
/*If lists are not empty, call an instance of Table view
    controller*/
ShowTable show = new ShowTable();
/*Retrieve data for control variable*/
show.retrieveData(control);
/*Call function to display data*/
show.viewData();
/*Open a new screen implementation is the same as that of calling
    Input Window*/
```

3. **SHOW CONSUMERS BUTTON**

Listing 3.31: Open Table View

```
/*If consumers and generators lists are empty, call an instance of
    Pop-up controler*/
/*This part is the same as calling Pop-up controller in Show Full
    List Button above*/
/*If lists are not empty, Call an instance of Table view
    controller*/
PieChartController pieChart = new PieChartController();
/*Retrieve data for control variable*/
pieChart.retrieveControl(control);
/*Data will be displayed when opening the Pie Chart Window*/
/*Open a new screen implementation is the same as that of calling
    Input Window*/
```

4. **Remove Consumers Button and Remove Generators Button**

Listing 3.32: Remove Generator and Consumer

```
/*Assume user inputs an integer n, with 0<n<consumers.size() or
    generators.size()*/
/*Remove consumers*/
for(int i=0;i<n;j++) {
   control.getConsumers().remove(0);
}
/*Remove generators*/
for(int i=0;i<n;j++) {
   control.getGenerators().remove(0);
}
```

5. **SET CONSUMERS OFF BUTTON**

This is an additional function that I have implemented for the GUI application. The idea is to let user inputs an integer n, with 0 <n <consumers.size(). We create an integer array that stores n unique random numbers, which means there are no duplicates in this array.

Listing 3.33: Generate Random Unique Numbers

```java
/**
 * This method is used to generate random UNIQUE numbers given in
     a certain range
 * @param times number of random UNIQUE numbers needed
 * @param size upper bound for randomizing (total number of
     consumers in this case)
 * @return list of random UNIQUE numbers
 */
public int[] randomize(int times, int size) {
   int[] result = new int[times];
   int add;
   /*A hash map is used to keep track of duplicate values*/
   HashSet<Integer> used = new HashSet<Integer>();
   for(int i=0; i<times; i++) {
      add = (int)(Math.random()*size);
      /*If the value is unique, we need to check the status of
           consumer at this position must be ON*/
      while(used.contains(add)||control.getConsumers().get(add).getPower()==0)
           {
         add = (int) (Math.random()*size);
      }
      used.add(add);
      result[i] = add;
   }
   return result;
}
```

Now what we need to do is to get the array, then perform the following logic:

Listing 3.34: Set N Consumers Off

```java
int randomNumbers[] = randomize(amount,
     control.getConsumers().size());
for(int i=0; i<amount; i++) {
    /*setting power to -1 means to turn off the consumer*/
    control.getConsumers().get(randomNumbers[i]).setPower(-1);
}
```

6. **NEXT ITERATION BUTTON**

Listing 3.35: Next Iteration

```
/*If no data, Pop Up screen with message "No data to simulate"*/
/*If iteration has reached 12, Pop Up screen with message
    "Simulation is already finished"*/
/*Else simulate data*/
for (AbstractComponent c : consumer) {
        c.next();
}
control.nextIteration();
/*Update and display detailed information of the next iteration
    onto the GUI*/
/*Store data into an 2D array for displaying line chart, itr is
    the global variable which shows the current iteration*/
graphInfo[itr+1][0] = control.getTotalDemand();
graphInfo[itr+1][1] = control.getTotalSupply();
graphInfo[itr+1][2] = control.getFrequency();
/*Clear line charts*/
lineChart.getData().clear();
lineChart2.getData().clear();
/*Set up new line charts*/
generatorGUI = new XYChart.Series<String,Number>();
consumerGUI = new XYChart.Series<String,Number>();
frequencyGUI = new XYChart.Series<String,Number>();
generatorGUI.setName("Generator");
consumerGUI.setName("Consumer");
frequencyGUI.setName("Frequency");
/*Add data into line charts*/
for(int i = 0; i < itr+2; i++) {
    consumerGUI.getData().add(new
        XYChart.Data(iter.valueOf(i+1),graphInfo[i][0]));
    generatorGUI.getData().add(new
        XYChart.Data(iter.valueOf(i+1),graphInfo[i][1]));
    frequencyGUI.getData().add(new
        XYChart.Data(iter.valueOf(i+1),graphInfo[i][2]));
}
/*Draws line charts*/
lineChart.getData().addAll(generatorGUI,consumerGUI);
lineChart2.getData().addAll(frequencyGUI);
/*Simple function display new values on Labels*/
showIterationData();
```

7. **PRINT BUTTON**

Listing 3.36: Generate reports

```
/*Generate real time and date*/
LocalDate reportData = java.time.LocalDate.now();
LocalTime time = java.time.LocalTime.now();
/*Create and open text files*/
```

```java
PrintWriter writer = new PrintWriter("Power Consumption" +
    reportData +".txt", "UTF-8");
PrintWriter writerCon = new PrintWriter("Consumers"
    +reportData+".txt","UTF-8");
/*list for storing information of power consumption of consumers
    and power distribution of generators*/
for(String i: report) {
   writer.println(i);
}
/*list for storing information of power distribution between two
    types of consumer*/
for(String i: reportCon) {
   writerCon.println(i);
}
/*Add date and time created for the reports*/
writer.println("-----------------\n Generated at "+ reportData + "
    at " + time);
writerCon.println("Generated at "+ reportData + " at " + time);
/*Close text files*/
writer.close();
writerCon.close();
```

3.6 Conclusion[19]

In this chapter, we have dissected and explain the variables and method of multiple module in this program. These module, when use together can form the complete program. In addition, because all modules base on the **AbstractComponent** and **IControl** interface, these modules can be interchange and work without error. (Nguyen)

We have also discussed in detail about the Graphical User Interface of the Simulator. Overall, this chapter **(section 3.5)** describes individual screens with their elements and how they cooperate with each other. The Graphical User Interface application fulfills all requirements of the project and has some additional features which were discussed in the sections above. However, the layouts do not look good and balanced because this is the first time I have developed a GUI application using JavaFX and Java Scene Builder. Possible improvements and future implementation for the GUI will be discussed later in chapter five of the report. (Viet)

[19] Author: Tra Ngoc Nguyen & Nguyen The Viet

Chapter 4

Problems and solutions

From the requirements, for the test scenario 1, the frequency should at no time be less than 48Hz or greater than 52Hz. When we finished the simulator, the mains frequency was often out of the range. Eventually, our group came up with two different solutions which we present in this chapter.

4.1 Solution 1: Finding power coefficients for generators[1]

A object of type ***GeneratorFactory*** has 5 fields:

- **maxPower**: maximum power a generator can reach.
- **minPower**: minimum power a generator can reach.
- **maxChange**: maximum amount of power a generator can change.
- **minChange**: minimum amount of power a generator can change.
- **power**: the power that a generator is generating.

The main task is to find out ratios $\frac{minpower}{maxpower}$, $\frac{maxchange}{maxpower}$ and then the coefficient for min change is from 0 to the ratio of $\frac{maxchange}{maxpower}$.

[1]Author: Hoang Thai Duong

Table to summarize input for 2 test scenarios.

Parameter \ Test scenario	1	2
Number of generators	15	6
Total max power(Watt)	11000	10000
Initial power(Watt)	5000	5000

First of all, we divide the total max power by the number of generators to get the average max power for each generator.

Listing 4.1: Calculate average max power

```
double avgMaxPower = totalMaxPower / numGen;
```

By default, every generator created has min power assigned. After assigning the min power as the generating power of a generator, we set its generating power to max power if the max power is lower than the provided initial power. The process is presented like this.

```
if (initialPower >= Math.floor(avgMaxPower)) {
  generator.get(i).setPower(avgMaxPower);
  initialPower -= avgMaxPower;
}
```

Table to find initial max power and min power

Parameter \ Test scenario	1	2
Total number of generators	15	6
Generators run max power	6	3
Generators run min power	9	3
Average max power(Watt)	11000/15 ≈ 733.33	10000/6 ≈ 1666.67
Sum of max power from running generators(Watt)	6x11000/15 = 4400	3x10000/6 = 5000
Power leftover(Watt)	5000-4400 = 600	5000-5000 = 0
Average min power(Watt)	600/9 ≈ 66.67	0/3 = 0
Difference between max power and min power(Watt)	733.33-66.67 ≈ 666.66	1666.67-0 ≈ 1666.67

As we can see from the table, the difference between max power and min power of a generator is very high. Especially, the min power for each generator in both scenarios are much too low. If power demand from consumers suddenly increases or decreases significantly, generators cannot change that quickly due to the limited range for changing supplying power in each generator. This leads to overload or outage. To solve the problem, we find the min power and the max change such that max power equals min power plus max change. This means the min power can reach the max power in the available change and the other way round. A system of linear equations to describe the concept.

$$\Rightarrow \begin{cases} 6\times(minPower_1 + maxChange_1) + 9\times minPower_1 & = 5000 \\ 3\times(minPower_2 + maxChange_2) + 3\times minPower_2 & = 5000 \end{cases}$$
$$\Leftrightarrow \begin{cases} 15\times minPower_1 + 6\times maxChange_1 & = 5000 \\ 6\times minPower_2 + 3\times maxChange_2 & = 5000 \end{cases}$$

We do not want to get the exact results of minPower and maxChange because they are only true for a specific case. Thus, we want to know the ratios of minPower/maxPower and maxChange/maxPower to apply in general cases.

We assume a, b are real numbers and $0 \leq$ a, b ≤ 1
minPower = a$\times$ maxPower
maxChange = b$\times$ maxPower

$$\Rightarrow \begin{cases} 15\text{a}\times maxPower_1 + 6\text{b}\times maxPower_1 & = 5000 \\ 6\text{a}\times maxPower_2 + 3\text{b}\times maxPower_2 & = 5000 \end{cases}$$
$$\Leftrightarrow \begin{cases} 15\text{a} + 6\text{b} & = \frac{5000}{maxPower_1}(maxPower_1 = \frac{11000}{15}) \\ 6\text{a} + 3\text{b} & = \frac{5000}{maxPower_2}(maxPower_2 = \frac{10000}{6}) \end{cases}$$
$$\Leftrightarrow \begin{cases} 15\text{a} + 6\text{b} & = 75/11 \\ 6\text{a} + 3\text{b} & = 3 \end{cases}$$
$$\Leftrightarrow \begin{cases} \text{a} & = 3/11 \\ \text{b} & = 5/11 \end{cases}$$
$$\Rightarrow \begin{cases} \text{minPower} & = (3/11)\text{maxPower} \\ \text{maxChange} & = (5/11)\text{maxPower} \end{cases}$$

The percentage for minChange is now in the range [0, $\frac{5}{11}$].
With those coefficients found and after many tests, the mains frequency was always in the range [48, 52].

4.2 Solution 2: Backup generators [2]

Due to the limit in max changes of generators, the power supply sometimes can not keep pace with the customer's demand causing the frequency to drop below 48Hz. Our solution for this case is the backup generators, which will start whenever the frequency fall below 48Hz and have their power adjusted to make up for the amount of supply needed to balance the frequency.

An array named backups and a boolean variable backup is used in this solution. The value of this array will also be declared along with the value of the main generators when the program initializes.

Listing 4.2: solution implementation

```
ArrayList<AbstractComponent> backups = new
    ArrayList<AbstractComponent>();
boolean backup = false;
```

The following methods are adjusted to implement this solution.

nextIteration() : void
In the nextIteration() method, after adjusting the generator's power, if the frequency is still below 48Hz, it will start the backup generators and adjust backup generator's power in the same way it does with the main generators. After that, the method does the frequency check and balance check as usual. The backup generators will be turned off at the next iteration.

Listing 4.3: code for starting and adjusting backup power

```
if (getFrequency() < 48) {
        backup = true;
        double bchangeAmount = 0;
        double bdiff = getTotalSupply() - getTotalDemand();
        double bsign = bdiff/Math.abs(bdiff);
        for (AbstractComponent b : backups) {
           changeAmount = changeAmount(bdiff,b)*bsign;
           b.setPower(b.getPower() - bchangeAmount);
           bdiff -= bchangeAmount;
        }
      }
```

[2] Author: Nguyen Hai Duc

getTotalSupply() : double
The power of the backup generators are included in the calculation of this method whenever it works.

Listing 4.4: code for calculating total supply

```
if(generators.size() > 0) {
         for (AbstractComponent g : generators) {
            supply += g.getPower();
         }
         if (backup) {
            for (AbstractComponent b : backups) {
               supply += b.getPower();
            }
         }
      }
```

getCost() : double
The cost of operating backup generators is also counter. Here I assume that every 1 Watt power provided by the backup generator is 25 Cents more expensive than of the main generators.

Listing 4.5: code for calculating total cost

```
for (AbstractComponent g : generators) {
         cost += g.getCost();
      }
      if (backup) {
         for (AbstractComponent b : backups) {
            cost += (b.getCost() + 0.25*b.getPower());
         }
      }
```

Chapter 5

Conclusion and Future work[1]

5.1 Conclusion

In the end, we finished this project and fulfilled all the minimum requirements. Most efforts were devoted for the graphical user interface because we would like to make our users find it comfortable and straightforward when using. In addition, the GUI also supports users and makes them understand better by allowing new manual input values during the simulating process. Another reason for a good GUI is that all attempts of the team was undertaken implicitly, it needs an evidence to demonstrate our contributions.
So far, there are several points we have learned from the project. Firstly, it is absolutely crucial that the entire team understands clearly all requirements before programming. Otherwise, the whole project may fail due to unclear objective. Secondly, the project manager plays an important role in the success of the project. Thirdly, communication between members helps increase effectiveness and speeds up the project. Moreover, not only did the team learn to use LATEX, improve teamwork skills and apply software engineering knowledge but also the awareness about power issue in reality was perceived.

[1]Author: Hoang Thai Duong

5.2 Future work

Our objective of the project is to develop a mains frequency simulator for a small metropolitan power grid. Although we already finished, it is not simple to develop the application fully as the real world is far more complicated. There are additional features to be continued implementing.

5.2.1 Extend new types of energy

So far we have implemented only one source of energy which is electricity generated by power plants. We would like to make it more practical by defining new energy types because different areas have different terrains and geographical characteristics. Those new types could be thermal power, nuclear power, etc.

5.2.2 Classify consumers into specific categories

In this project we just categorized consumers into industrial zone and residential zone. More types such as commercial (businesses that have storefronts) and municipal(street light, water plants, sewer plants, pumping stations etc) can be added. Later, the suitable price is set for each type of customers.

5.2.3 Add more consumers[2]

In practice, consumer type is divided mainly into three categories: Commercial, Residential and Industrial types. Due to lack of time, we only implemented two types that have significant differences of power consumption for demonstration purpose. It would be complete if Commercial type is implemented in the future. (Reference: `https://www.epa.gov/energy/electricity-customers`)

5.2.4 Apply statistics to make the simulation more precise

Due to the lack of time, we mainly focused on simulating with simple logic. In order to make the program more reliable, mathematics has to be concerned. The more accurate the program is, the more users show their confidence to the simulator.

[2]Nguyen The Viet

5.2.5 Improve the graphical user interface[3]

On developing new functions, we will try to provide users with the most easy-to-use interface. A good GUI can greatly improve the efficiency of the working progress. (Duong)

Potential features that can be improved or added for the GUI in the future:

1. Setting a specific consumer off in the simulator
2. Setting consumers on
3. Calculate Mean value on iterations to compare the difference of power usage between two type of consumers
4. Improving GUI layouts
5. Improving Pie Charts to show statistics of the third type of consumer (Commercial)

(Viet)

[3]Hoang Thai Duong, Nguyen The Viet

Chapter 6

Work done

6.1 Tra Ngoc Nguyen - 10248

- **Programming**: Consumer Core
- **Writing report**:
  - **Chapter 1**: Introduction
    - Requirements
    - Outline
  - **Chapter 2**: Software analysis
    - Introduction
    - Use-cases Diagram
    - Conclusion
  - **Chapter 3**: Modules
    - Introduction
    - Consumer Module - Initial implementation

6.2 Nguyen Hai Duc - 9701

- **Programming**: Control module
- **Writing report**
  - **Chapter 2**: Software analysis
    - Class Diagram
  - **Chapter 3**: Module
    - Control Module - Core implementation
  - **Chapter 4**: Problems and Solutions
    - Solution 2 : Backup generators

6.3 Nguyen The Viet - 9990

- Initializing the idea of two new types of consumer and power distribution table
- **Programming**
  - Designing and implementing entire GUI application of the Simulator
  - Modifying ConsumerFactory to adapt changes to the GUI application
  - Debugging the program
- **Writing report**
  - Chapter 3.5 - Graphical User Interface (Except for the part 3.5.3.1)
  - Chapter 3.6 - Conclusion for Graphical User Interface
  - Chapter 5 - Add Consumers; Improvement For GUI

6.4 Hoang Thai Duong - 11052

- **Programming**
  - Implementing entire Generator module
  - Extending and implementing Residential and Industrial
  - Modifying and correcting Consumer module
  - Modifying Control module to adapt with changes from Consumer module
  - Coding data input for GUI
- **Writing report**
  - **Chapter 1**: Introduction
    - Motivation
    - Objective
  - **Chapter 3**: Modules
    - Consumer Module - Additional implementation and modification
    - Control Module - Additional implementation
    - Generator Module
    - Graphical User Interface - Reading input
  - **Chapter 4**: Problems and solutions - Solution 1: Finding power coefficients for generators
  - **Chapter 5**: Conclusion and future work

Chapter 7

Acknowledgement

We would like to express our thanks to the teaching assistant of the project, Mr. Denis Hock, who gave us useful advice for our project and maintained individual talks every week for problem solving. We usually received helpful feedback when we were stuck with design problems and understanding requirements. Besides, suggestions from Mr. Hock for the structure of this final report made us feel confident to do our work.