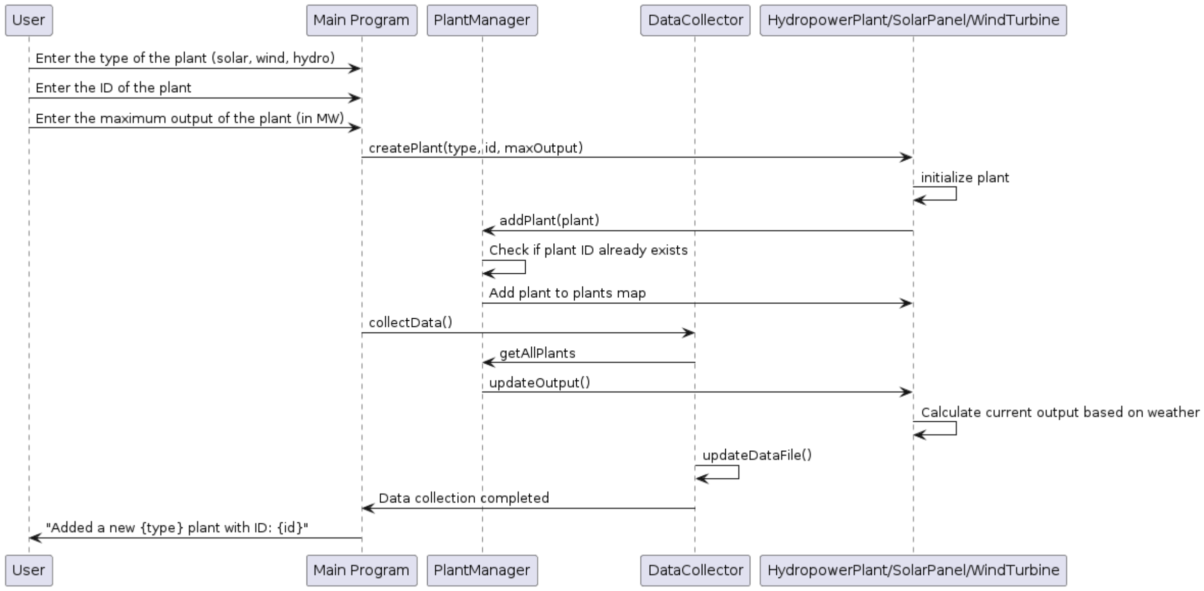
Sequence Diagram

Use Case 1: Add a new plant

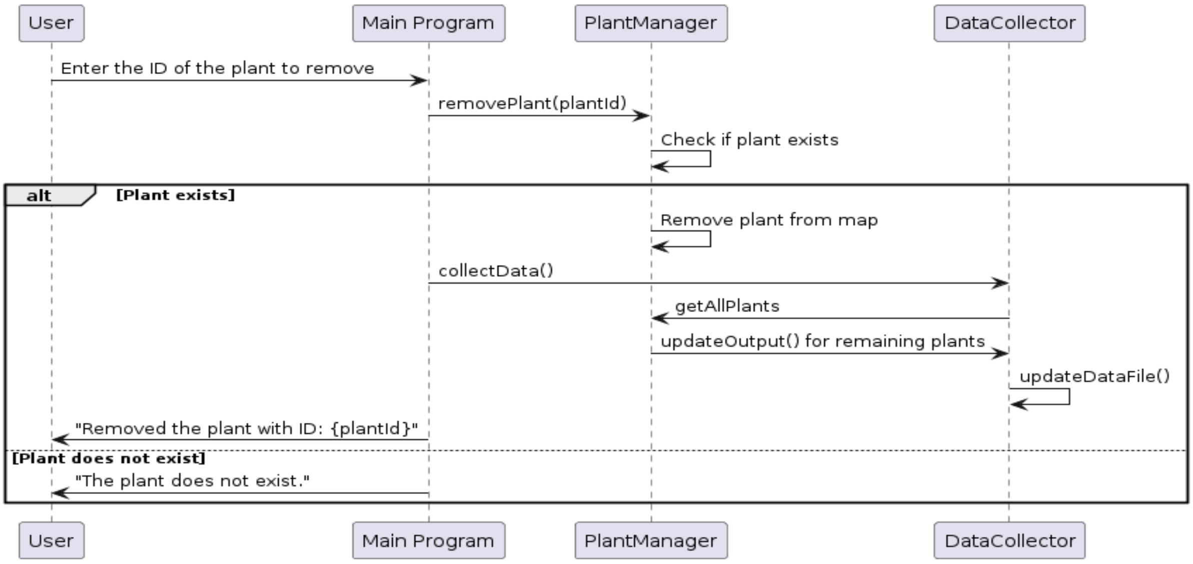
The main function of use case 1 is to allow the user to add new generation facilities to the renewable energy power plant management system.


User
Main Program
PlantManager
DataCollector
HydropowerPlant/SolarPanel/WindTurbine
Enter the type of the plant (solar, wind, hydro)
Enter the ID of the plant
Enter the maximum output of the plant (in MW)
createPlant(type, id, maxOutput)
initialize plant
addPlant(plant)
Check if plant ID already exists
Add plant to plants map
collectData()
getAllPlants
updateOutput()
Calculate current output based on weather
updateDataFile()
Data collection completed
"Added a new {type} plant with ID: {id}"

1. User Input - The user enters the type, ID and maximum output value of the generating station.
2. Create Power Station - The main program calls the appropriate constructor to create a Power Station object of a specific type.
3. Add to Manager - The newly created Power Station object is passed to the addPlant method of the PlantManager.
4. Check for Duplicate IDs - PlantManager checks to see if a Power Station with the same ID already exists.
5. Update Data Collection - The collectData method of the DataCollector is called to update the data of all the generating stations in the system and write it to a CSV file.
6. Feedback to User - The main program feeds the creation results back to the user.

Use Case 2:Remove a plant

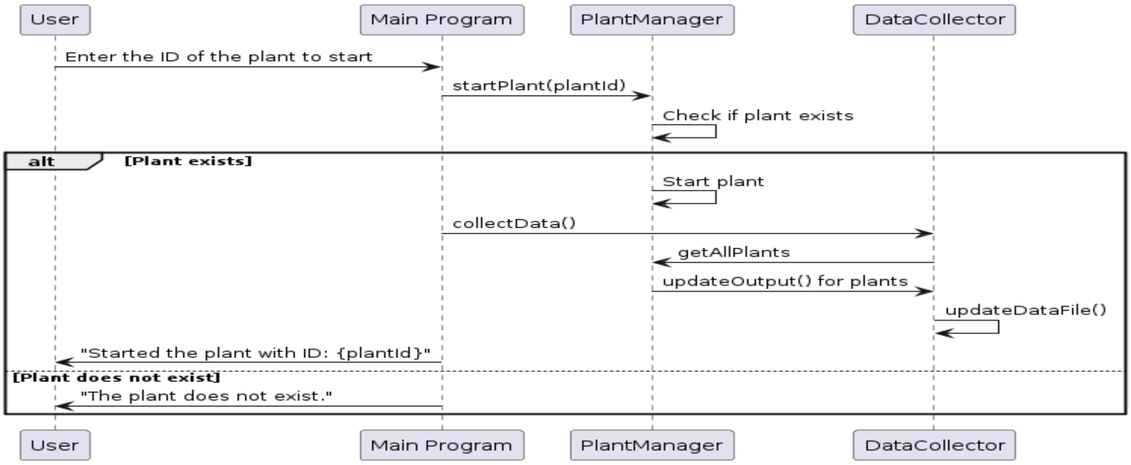
Case 2 is an operation in which a user removes a specified generating facility from the system.



1. User Input - The user enters the ID of the generating facility they wish to remove via the command line.
2. Perform Removal - The main program calls the PlantManager's removePlant method to attempt to remove the generating facility with the specified ID.
3. Check and Remove - PlantManager checks to see if a plant with this ID exists and removes it if it does.
4. Update Data - The DataCollector's collectData method is called to update the data in the system and save the updated data to a CSV file.
5. User Feedback - The system confirms to the user that the generating facility has been successfully removed or notifies the user if the facility with the specified ID is not found.

Case 3:Start a plant

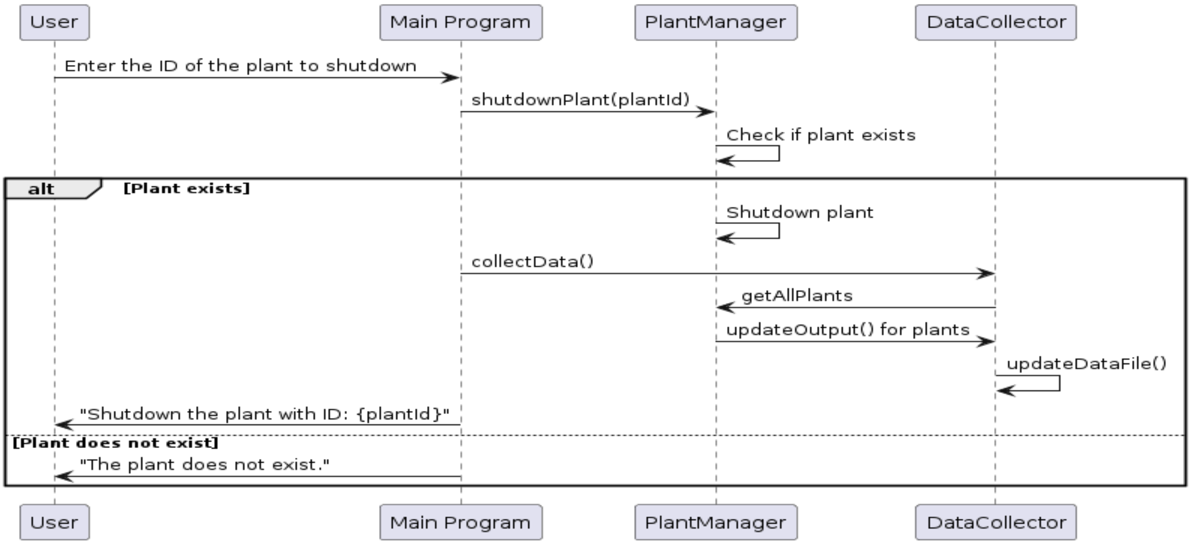
The role of Case 3 in the main program is to handle the “Start a Generation Facility” operation. It allows the user to start the facility by entering a specific facility ID.



1. User Input - The user enters the ID of the plant to be started via the command line.
2. Perform startup operation - The main program calls the startPlant method of PlantManager.
3. Check and start - PlantManager checks to see if a generating facility with this ID exists and starts it.
4. Update Data - The collectData method of the DataCollector is called to update the system data.
5. User Feedback - The system provides feedback to the user on the results of the operation.

Case 4:Shutdown a plant

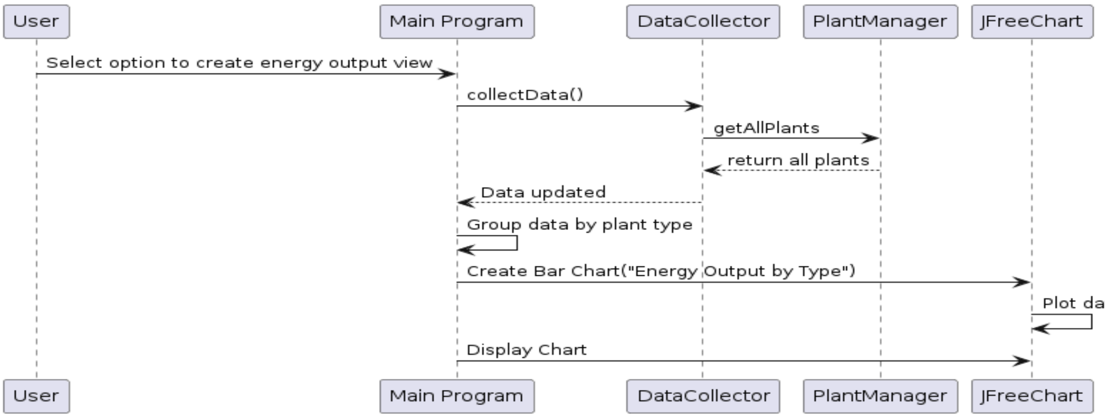
Case 4 is an operation to shut down a generating facility. This allows the system operator to shut down a facility for maintenance by entering a specific generation facility ID.



1. User Input - The user enters the ID of the plant to be shut down via the command line.
2. Perform Shutdown - The main program calls PlantManager's shutdownPlant method.
3. Check and Shutdown - PlantManager checks to see if a generating facility with this ID exists and shuts it down.
4. Data Update - The collectData method of the DataCollector is called to update the system data.
5. User Feedback - The system provides feedback to the user on the results of the operation.

Case 5:Create a view for energy output

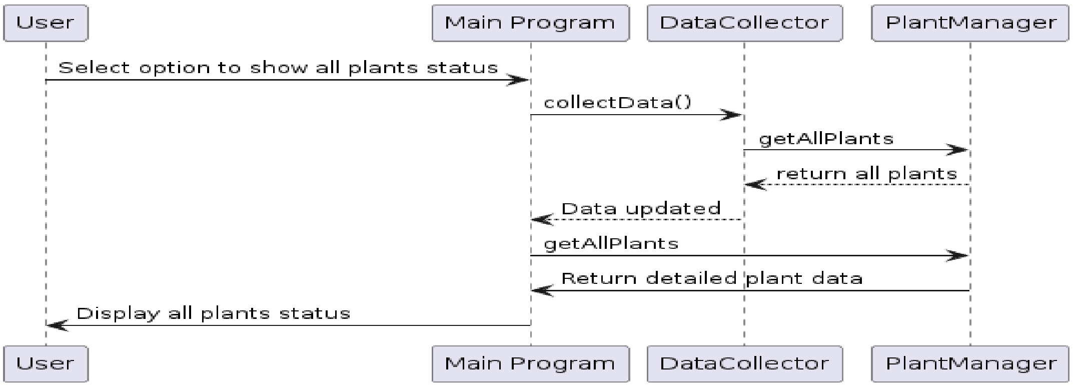
This use case deals with "creating a view of energy output". This mainly involves collecting output data from the system for different types of power stations and visualizing this data as a graph.



1. User-triggered data view generation - the user selects the option to create an energy output view via the command line.
2. Data Collection - The main program calls the collectData method of the DataCollector to update the system data.
3. Data Sorting - The main program gets the current output of all plants through PlantManager and sorts the statistics according to plant type.
4. Generate Chart - Use the JFreeChart library to create a bar chart showing the total output of each plant type.
5. Display Charts - Charts are displayed in a graphical interface to provide the user with an intuitive visualization of the data.

Case 6 Show all plants status:

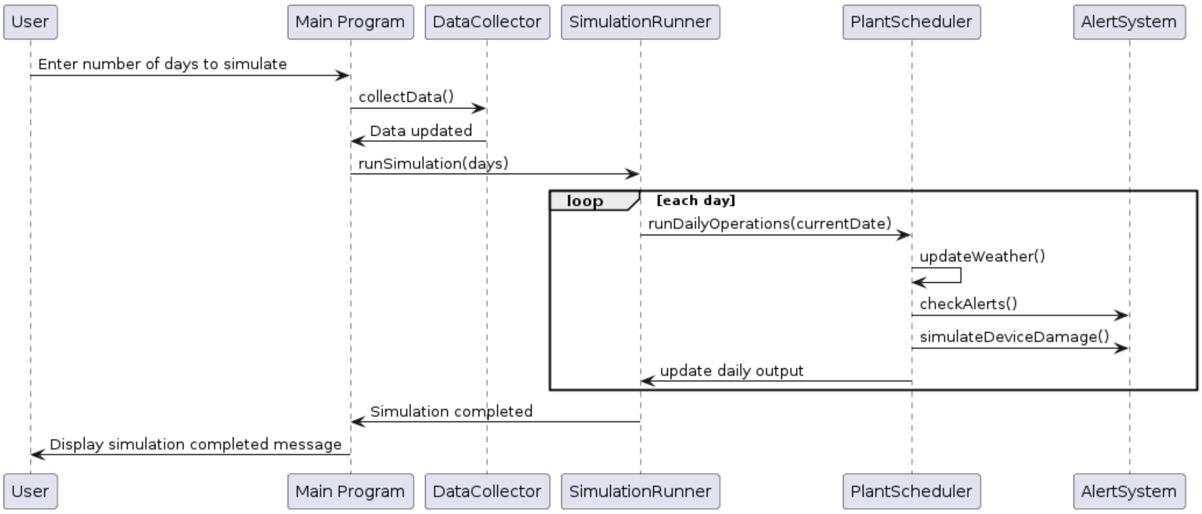
The purpose of Case 6 is to allow the user to quickly view and assess the current operational status of all power generation facilities.



1. User Triggered Status Display - The user selects the option to display the status of all facilities via the command line.
2. Data Update - The main program calls the collectData method of the DataCollector to ensure that the data in the system is up to date.
3. Get Facility Information - The main program gets detailed information about all plants through PlantManager.
4. Display Information - The system outputs the current status of all plants and other relevant information on the console.

Case 7 Simulate running in a given period :

The role of Case 7 in the system is to allow the user to perform long-term simulations to evaluate the performance of the generating facility under different environmental conditions and possible equipment failure scenarios.

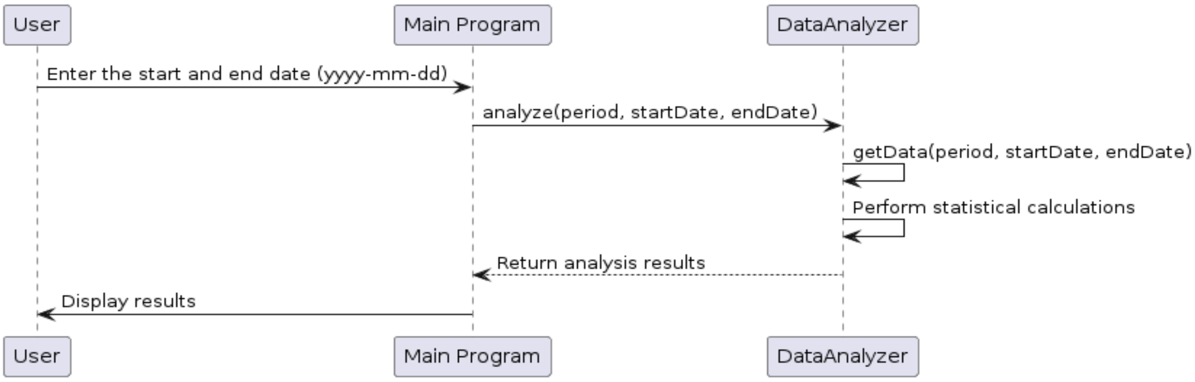


1. User Input Simulation Days - The user inputs the number of days they want the simulation to run via the command line.
2. Initialize Data Collection - The DataCollector may be used to update the status of all generation facilities before the simulation begins.
3. Simulation Run - SimulationRunner performs a daily simulation run based on the number of days entered by the user.

4. Daily Operation - PlantScheduler updates the operational status of the facility on a daily basis based on environmental data, while AlertSystem checks for alarms and simulated equipment damages and handles them automatically (alarms).
5. Update and Record Output - At the end of each day's simulation, statistics are recorded to a historical data file.
6. Feedback Completion Message - At the end of the simulation, a message is displayed to the user that the simulation has been completed.

Case 8:

Case 8 serves to allow the user to specify a date range and perform detailed statistical analysis of the data within that period, including the calculation of statistical quantities such as mean, median, plurality, data range, and median distance.


User
Main Program
DataAnalyzer
Enter the start and end date (yyyy-mm-dd)
analyze(period, startDate, endDate)
getData(period, startDate, endDate)
Perform statistical calculations
Return analysis results
Display results
User
Main Program
DataAnalyzer

1. User Input Time Range - The user inputs the start and end dates of the analyzed data via the command line.
2. Parsing Date and Calling Analyze Functions - The main program parses the entered date and calls the analyze method of DataAnalyzer to analyze the data.
3. Get and group data - DataAnalyzer internally calls the private method getData to get and group the data according to the date.
4. Perform Data Analysis - After the data is grouped according to the user-selected time period (days, weeks, months), the mean, median, plural, range, and median are calculated for each period.
5. Output Analysis Results - The analysis results are printed out via the console for the user to view.

Class Diagram:

The project consists of three main modules: Data, Facility and Utils. The system manages different types of power generation facilities, such as solar panels, wind turbines, and hydroelectric plants, through `PlantManager`, and utilizes `DataCollector` and `WeatherData` to monitor and adjust the efficiency of power generation in real time. The system also includes `DataAnalyzer` for data analysis, as well as `AlertSystem` and `PlantScheduler` to ensure the efficient operation and maintenance of power generation facilities.

Data Module

DataAnalyzer Class

private val dateFormat: DateTimeFormatter

fileName: String

Functions: getData(period: String, startDate: LocalDate, endDate)

getData(period: String, startDate: LocalDate, endDate: LocalDate): Map[LocalDate, List[Double]] - Reads and organizes data from historydata.csv into time units.
analyze(period: String, startDate: LocalDate, endDate: LocalDate): Unit - Processes the data and calculates the mean, median, plurality, range and median distance.

DataCollector Class
Attributes: fileName: String
fileName: String
plantManager: PlantManager
weatherData: WeatherData
Functions.
loadData(): Unit - Load data from a file to get the status of the plant.
updateDataFile(): Unit - Writes the updated plant information to historydata.csv.
collectData(): Unit - update the energy output of the power plant.

WeatherData Class
Attributes.
private var sunlightIntensity: Double
private var windSpeed: Double
private var waterFlow: Double
private val timer: Timer
Functions.
getSunlightIntensity: Double - Get the sunlight intensity.
getWindSpeed: Double - Get the wind speed.
getWaterFlow: Double - Get the water flow.
updateWeather(): Unit - Timer to update the weather data.
Facility Module
EnergyPlant Class (abstract)
Attributes: id: String
id: String
maxOutput: Double
protected var currentOutput: Double Double
private var deviceStatus: DeviceStatus. private var deviceStatus: DeviceStatus
private var status: String String
private var deviceStatus.
getMaxOutput. GetMaxOutput: Double
getCurrentOutput: Double Double
getDeviceStatus. DeviceStatus
getStatus. String
setStatus(newStatus: String). String
setDeviceStatus(newStatus: DeviceStatus). Unit
setDeviceStatus(newStatus: DeviceStatus): Unit Unit start(): Unit
Unit setDeviceStatus(newStatus: DeviceStatus): Unit start(): Unit Unit start(): Unit shutdown(): Unit
updateOutput(): Unit (abstract) Unit (abstract)

SolarPanel Class
Inherits: EnergyPlant SolarPanel Class Inherits: EnergyPlant
Class Inherits: EnergyPlant

updateOutput(). Unit - Adjusts the energy output according to the sunlight intensity.

WindTurbine Class
Inherits: EnergyPlant
Functions.
updateOutput(): Unit - Adjusts the energy output according to the wind speed.
HydropowerPlant Class
Inherits: EnergyPlant
Functions: updateOutput(): Unit - Adjusts the energy output according to the wind speed.
updateOutput(): Unit - Adjusts the energy output according to the water flow.

Utils Module
PlantManager Class
Attributes.
private val plants: mutable.Map[String, EnergyPlant]
Functions.
addPlant(plant: EnergyPlant): Unit
removePlant(id: String): Unit
getPlant(id: String): Option[EnergyPlant]
getAllPlants: Map[String, EnergyPlant]
shutdownPlant(id: String): Unit
startPlant(id: String): Unit
getTotalOutput: Double

AlertSystem Class
Attributes: plantManager: PlantManager
PlantManager: PlantManager
Functions: plantManager: PlantManager
checkAlerts(): Unit - Checks if all running plants output less than 10% of their maximum output.
simulateDeviceDamage(): Unit - Simulates device damage.
PlantScheduler Class
Attributes.
plantManager: PlantManager
alertSystem: AlertSystem
weatherData: WeatherData
private var runningDays: mutable.Map[String, Int]
Functions: initialize(startFraction: Double)
initialize(startFraction: Double): Unit
runDailyOperations(currentDate: LocalDate): Unit - Simulates daily operations.
FacilityUtils Object
Functions.
generateId(prefix: String, length: Int): String - Generate a facility id.

Relationship:

EnergyPlant:
Inheritance: SolarPanel, WindTurbine, HydropowerPlant are inherited from EnergyPlant.
Relationship: Not directly related, but managed by PlantManager.
SolarPanel, WindTurbine, HydropowerPlant:

Inheritance: These three classes inherit from EnergyPlant.
Dependency: Depend on WeatherData to get the corresponding environmental data for output calculation.

PlantManager:

Relationship: Manages multiple EnergyPlant, which includes SolarPanel, WindTurbine, HydropowerPlant.
Dependencies: Used by DataCollector, PlantScheduler, AlertSystem, SimulationRunner to manage plants.

DataCollector:

Dependency:
Depends on PlantManager to access and modify plant data.
Depends on WeatherData to get weather conditions.
May depend on FacilityUtils to create new plant IDs (if the CreatePlant function is implemented in this category).

WeatherData:

Dependency: Used by subclasses of EnergyPlant and PlantScheduler to get real-time weather data that affects plant output.

DataAnalyzer:

Dependency: mainly called by Main to analyze data.

AlertSystem:

Dependency: Dependency on PlantManager to get plant status for alert management.

PlantScheduler:

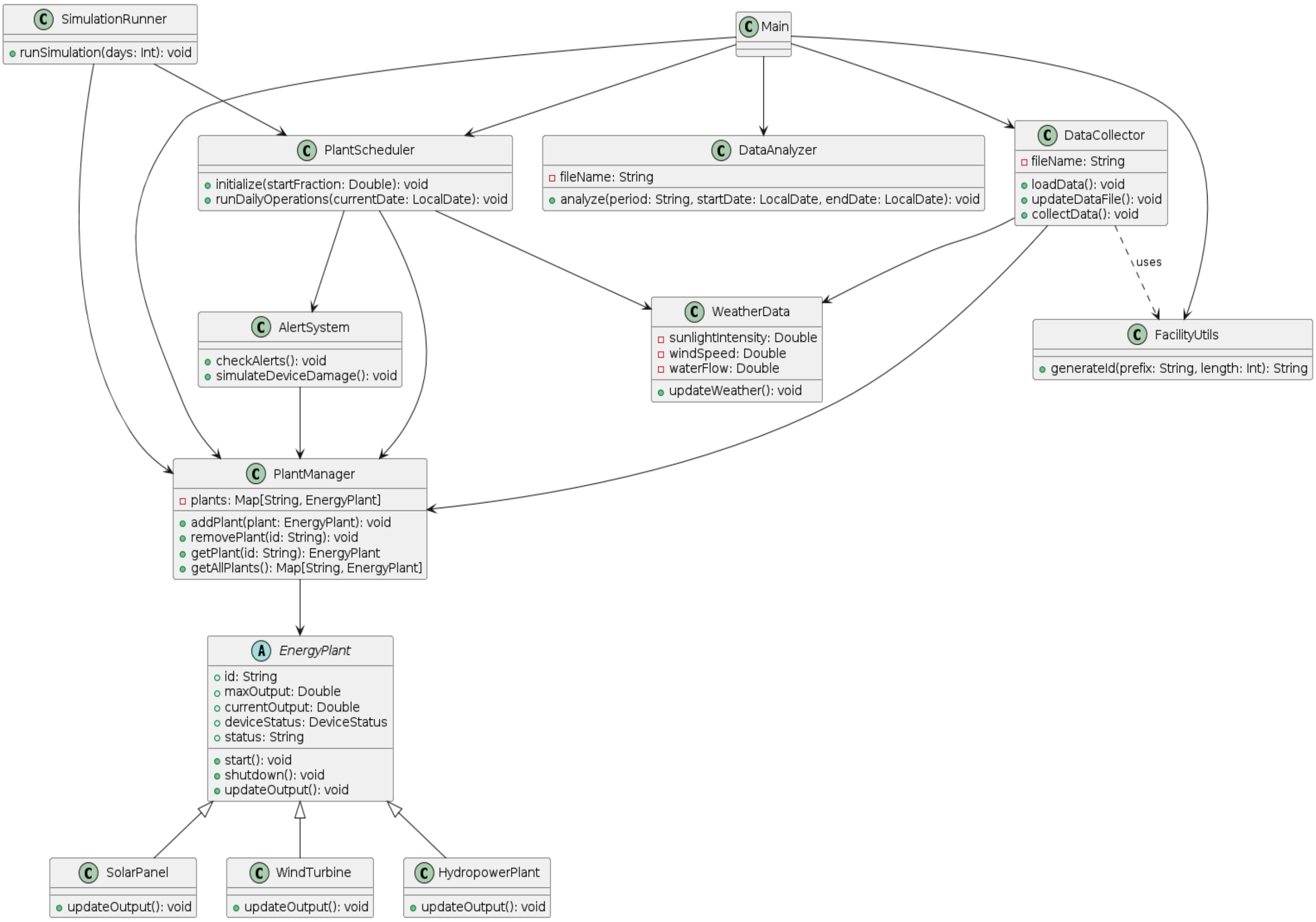
Dependency.
Dependent on PlantManager to access and control the operation of plants.
Depends on AlertSystem and WeatherData to adjust the plant schedule based on weather conditions and plant status.

SimulationRunner:

Dependency: Dependency on PlantManager and PlantScheduler to simulate plant operation and management.

FacilityUtils:

Dependency: Used by Main and possibly DataCollector to generate plant IDs.



Part B

FUNTOR

Theoretical explanation of concept

A Functor is a foundational concept in functional programming, serving as a structure that enables users to operate on a given container with a defined mapping operation. This mapping operation facilitates the conversion of a value from one type to another within the container. Fundamentally, a Functor consists of two essential components: the container, representing the data structure holding the values, and the function, which acts upon these values [1].

The container provides the context for the Functor, which enables structured transformations. The Functor's mapping operation allows users to apply a function to each element within the container, transforming it according to the function's logic [1]. This operation links functions, ranging from simple to complex, to the container elements.

Functors adhere to two core laws: identity and composition. The identity law stipulates that mapping the identity function over a Functor should leave the structure of the container unchanged [2]. The composition law ensures that the sequence of function applications does not alter the final result, thereby supporting predictable outcomes in functional programming. Collectively, these laws allow Functors to offer a consistent framework for data transformations, crucial for reliable and maintainable software.

A code base with a minimalistic but meaningful implementation of the concept

A functor usually consists of two parts, including a container and a function acting on it. The container can be a List, while functions can act on the container through a map[1]. Here is a simple example[1,2]:

```
object Tryy extends App {
  def double(x: Int) = x * 2
  def mod(x: Int) = x % 2
  val list = List(7, 8, 9)
  val doubleList = list.map(double)
  println(doubleList)
  val option: Option[Int] = Some(5)
  val modOption: Option[Int] = option.map(mod)
  println(modOption)
}
```

DoubleList is one of the simplest implementation method of functors, which applies the double function to the list through a map to achieve the double effect of all elements in the list(output is "List(14, 16, 18)"). Map can also be applied to options, such as modation where the mod function is executed on the option(output is" Some(1)").

A functor also works according to the identity and composition law, so here is an example code to show these two laws

```
object Functor {
   def apply[F[_]](implicit f: Functor[F]) = f
}
val expectedList = List(1, 2, 3)
val outputList = Functor[List].map(expectedList)(x => x)

expectedList == outputList
// true
```

The 'Functor' object in Scala, a type class, allows the application of the identity law to functors by generating instances for various containers like 'List'. Mapping the identity function over 'expectedList' with 'Functor[List].map' produces 'outputList', identical to 'expectedList', demonstrating the preservation of structure as required by the identity law.

```
val f1 = (x: Int) => x + 1
val f2 = (x: Int) => x * 2
val inputList = List(1, 2, 3)
val compRes = Functor[List].map(inputList)(f2 compose f1)
val mapRes = Functor[List].map(Functor[List].map(inputList)(f1))(f2)
compRes == mapRes
// true
```

The Scala code demonstrates the composition law for functors by comparing two methods on an `inputList`: composing functions `f1` and `f2` into a single function and then mapping it over the list, and mapping each function separately over the list followed by composing the results. An equality check confirms that both approaches yield the same result, validating the composition law for functors.

Reference

1. Functors in Functional Programming, Eugen Paraschiv, September 30, 2021 Available: https://www.baeldung.com/scala/functors-functional-programming

2. Functor | Scala Tour, Scala Tour, 2014, Available: https://dcapwell.github.io/scala-tour/Functor.html