A Software documentation guide for the Smart Grid Simulator

M. Pöchacker

Feb. 2014

Abstract

This report gives a short introduction to the Smart Grid Simulator. It explains how to use the implemented models. And gives help in programming your own energy distribution algorithm.

Contents

1 Introduction

The Smart Grid Simulator (SGS) is a Java software project developed at the Smart Grid Group. The main goal is to provide a simple to use simulation environment for smart grid scenarios.

In the section 2 we describe how to use the SGS with one of the implemented algorithms. The following section 3 contains information about the software architecture.

2 Getting Started

Import the SmartGridSimulator-package and the JGridMap to Eclipse. Run the program to get the main window of SGS.

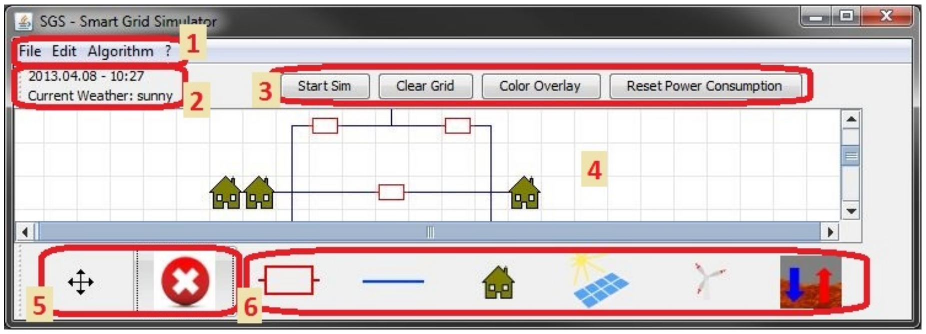


Figure 1: The SGS main window showing a random test case scenario.

The main window contains following elements:

1. The menu with the functions: `File`, `Edit`, `Algorithm` and `?` (= info).
2. Weather and time information.
3. Four buttons to control the simulation at the top button bar.
4. The main grid-map which is zoom- and moveable. It opens automatically the scenario which was stored last.
5. Two grid edit buttons.
6. Six buttons for the objects of the simulation in the bottom button bar.

2.1 How does it work?

The SGS allows to define objects and create scenarios by placing them in a lattice structure. All the objects have properties that can be modified manually and scenarios which can be saved. By choosing a predefined algorithm, or implementing a new one, it is possible to perform analysis on that scenario,

i. e., run the simulation. The algorithm modifies some of the objects properties and gives feedback by showing a Color Overlay or saving a result file.

2.2 Placing and editing objects

The bottom button bar shows two object edit functions and seven different objects. The two edit-buttons are for

1. Moving an object and
2. Deleting an object.

The predefined objects are:

1. Power Line Resistor is the object that accumulates all the losses of a power line for the model.
2. Super conductors are not modeling physical super conductors, they are lossless connectors for the different objects.
3. House stands for a power consumer specified by its power demand.
4. Solar Power Plant produces power depending on the sun.
5. Wind Power Plant produces power depending on the windspeed.
6. Geothermal Power Plant is preparing load inedpendant of climate data.
7. PowerGrid represants a connection to the main power grid.

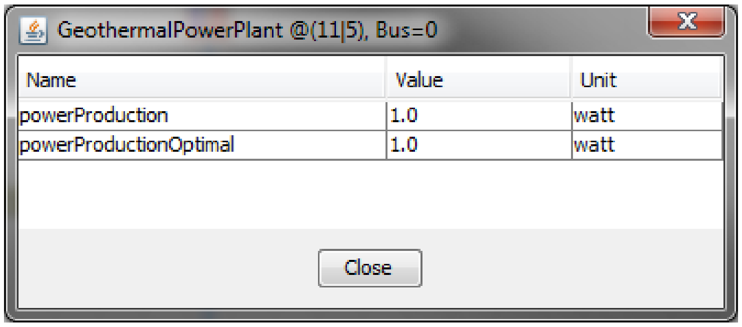
In general it is possible to add objects or replace them by a different one. See more about object definition in section 3.2. The connectors and power line objects align automatically with neighboring objects. Incorrect positioning will give a warning.

To access the properties of an object do a right click on it. This opens the properties window of the object. Which properties are visible and/or editable is defined in the selected algorithm.

Each property has a name, a value and a unit. The value could be of any numerical type. Complex values are shown with real and imaginary part separated by comma. The property window can also be closed by pressing $\boxed{\text{Esc}}$.

2.3 Choose an Algorithm and Load/Save a File

These section adresses all functions that are available through the menue.



Name	Value	Unit
powerProduction	1.0	watt
powerProductionOptimal	1.0	watt

Figure 2: The properties window for a GeothermalPowerPlant at the position 11/5. The position is counted from the top left corner, starting with zero. The two properties are defined by the Simple Power Distribution algorithm.

2.3.1 The File Button

The menu appearing after clicking the File Button contains following functions:

- `New`: Clears the current scenario (after confirmation of the warning) and generates a file for the new scenario.
- `Open`: Opens a browser window for selection of a saved scenario (a .zip file) to open.
- `Save`: Stores the current scenario as the default one, that means it opens automatically when starting SGS.
- `Save as`: Stores the current scenario in a file. The used format is zip.
- `Option`: It is not used.
- `Exit`: Closes the SGS.

2.3.2 The Edit Button

Currently no function.

2.3.3 Choose the Algorithm

After clicking on Algorithm a selection menu for the implemented Energy Distribution Algorithms opens. To get information about the selected algorithm click the info button `?` that makes a description window poping up for the selected algorithm. Which properties of an object are visible and/or editable depends on the selected algorithm. Different algorithms might use the same variables in different ways.

Simple Power Distribution

Each object in the scenario states its desired consumption or production, respectivelly. The algorithm sums up all the available power production in the connected part of the grid and allocates it to the consumers without physical order. The under provided consumers move to red in the Color Overlay, as well do generators working below their optimal capacity. Losses in the powerlines are not considered at all. Dynamics come from production changes due to climate data.

AC Powerflow Calculator

This algorithm solves the AC Powerflow problem for the grid scenario. It uses a list of Buses and Paths generated by the Network Analyzer. Each Bus, which includes all object that are lossless connected, is characterized by four parameters:

i) active power P,
ii) reactive power Q,
iii) voltage magnitude U and
iv) voltage angle.

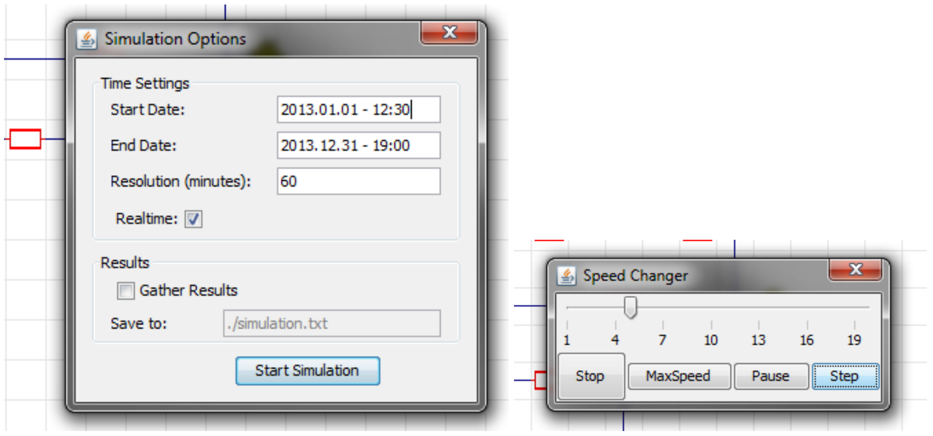
Two of them are given, the other two are estimated by the algorithm. The four values are summarized in two complex variables. One bus is defined as `REFERENCE_BUS` with $Volt\_ang = 0$ and $Volt\_Mag = 1$ or an other given fixed value (use of relative voltage values help a lot for solver convergence). The other buses are either classified as `LOAD_BUS` (P and Q are given) or as `GENERATOR_BUS` (P and $Volt\_Mag$ are given). For each bus the two missing parameters are determined by iteratively solving the set of differential equations $S = Y * U^2$ (for instance with the Gauss-Seidl method). This equation is in n dimensions (n=number of buses) where S is the appearant power (S=P+jQ), U is the complex voltage vector and Y is the addmitance matrix (Y_ij contains the addmittance between bus i and bus j, the admittance is the inverse of the resistance). The admittance matrix considers each power line by its π-equivalent circuit (the resistance between two buses and a `powerLineCharge` to ground). The `REFERENCE_BUS` is automatically set to the bus specified with no P and/or Q value. Buses with positive `NettoPower` are `GENERATOR_BUS`.

An example of power flow calculation could be found at `http://ecourses.vtu.ac.in/nptel/courses/Webcourse-contents/IIT-KANPUR/power-system/chapter_4/4_4.html`.

2.4 Running the Simulation

The `Start Sim` button opens the Simulation Options Window. It allows to insert controll data for the Time Thread, in the **Time Settings** fields, and for saving options in the **Results** field, see Figure 3. Pressing `Start Simulation` switches the `Start Sim` in the main window to `Stop Sim`. It also opens the

Speed Changer Window which allows to modify the Time Thread directly. Closing the Speed Changer Window does not influence the time thread at all. The functions of its buttons are self explaining. The used speed scale is arbitrary.


Simulation Options

Time Settings
Start Date: 2013.01.01 - 12:30
End Date: 2013.12.31 - 19:00
Resolution (minutes): 60
Realtime: [✓]

Results
Gather Results [ ]
Save to: ./simulation.txt

Figure 3: Two windows control the Time Thread. The Simulation Options Window sets basic values, i. e., start-, stop time and time increment, as well as result saving options. The Speed Changer controlls the Time Thread on the fly.

The **Clear Grid** button requires confirmation before executing the expected function.

The **Color Overlay** is a function that is supported by the JGridMap package. It gives additional feedback about simulation results. Which parameters are involved in that feedback is defined in the Object classes. Its default function is the fraction between required and supplied power.

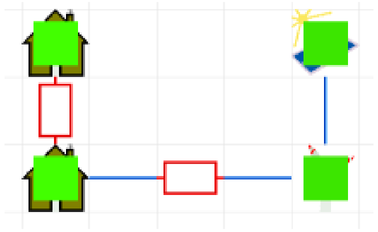


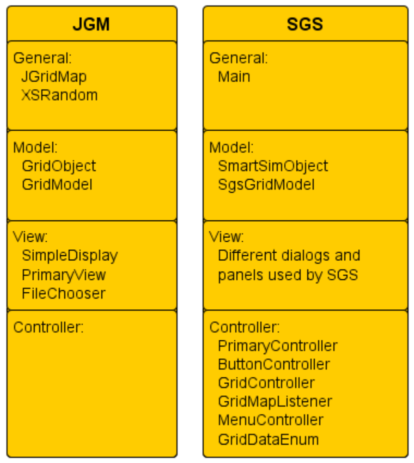
Figure 4: The Color Overlay gives direct feedback about simulation results.

The **Reset Power Consumption** is used to reset power consumption values of all objects to zero.

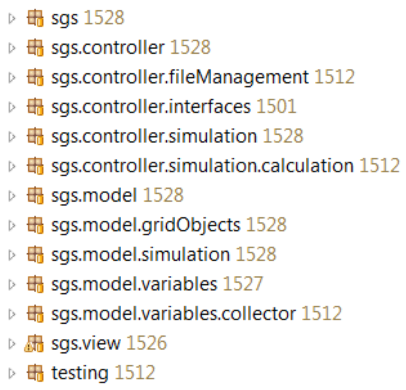
3 Software modification

3.1 The Software Architecture

The software project Smart Grid Simulator (SGS) is written in Java by mainly use of Eclipse and, for some parts of the GUI, Jigloo. Be sure to use an up-to-date version of Eclipse to extend it. The general architecture of the SGS follows the Model-View-Controller concept for software engineering. The idea is to keep the simulator flexible and extendable what reasons in a more complex software structure. Ideally the user can focus on implementation of simulation algorithms and on the objects including their properties.



(a) Structure of the JGridMap and the SmarGridSimulator project.



```
sgs 1528
sgs.controller 1528
sgs.controller.fileManagement 1512
sgs.controller.interfaces 1501
sgs.controller.simulation 1528
sgs.controller.simulation.calculation 1512
sgs.model 1528
sgs.model.gridObjects 1528
sgs.model.simulation 1528
sgs.model.variables 1527
sgs.model.variables.collector 1512
sgs.view 1526
testing 1512
```

(b) Packages of the SGS-project.

Figure 5: The software design is based on the Model View Controller architecture.

3.1.1 The JGridMap (JGM)

The SGS software uses functions and objects of the JGridMap which manages the main Grid Map (see Figure 1) and is a separate project. Its basic structure is visible in Figure 5a. The listed parts are briefly explained in the following. All the functions for displaying and managing the grid are located in the `JGridMap` class. `XSRandom` provides an improved random function compared to the one from Java. The `GridObject` is a class, which instances represent any object in the graphical lattice. Displayable objects, of a project using JGM, need to extend this class. The `GridModel` contains all parameters which are necessary during runtime. This includes all the Grid Objects, the used instance of JGridMap and also some static settings. The `PrimaryView` provides an entire user surface including the menue and button bars.

3.1.2 The SGS packages

In Figure 5b are listed all packages of the SGS project. The `Main`-class, which is in the sgs package, is initializing the program. It first starts the model, then the view and finally it starts the controller. The `SmartSimObject` and `SgsGridModel` classes are extending the classes of the JGM, e. g., GridObject and GridModel, respectively. The instances of `SmartSimObjects` are explaind more detailed in Section 3.2 and Figure 9. The controller of the SGS is splitted into various areas. The `PrimaryController` handles the coordination between the other controllers. The GridMapListener implements an interface of the JGM and manages the grid lattice control.

3.2 The SGS Objects and their Properties

All objects are captured by an ID in the Grid Model. And there is a list of all the defined object types. This list is the `GridDataEnum` which is located in the sgs.model-package. It is used for identification of the objects in the grid, for loading objects from a saved file and for handling the graphics used to display the objects. For each item in the `GridDataEnum` a button in the GUI will be generated automatically. `GridDataEnums` are refering to not-Java files, that is why we highly recommend not rename them. For more detailed information please see the rich comments in the code.

Figure 9 gives an UML-diagram of all the currently defined object types. All of them are extendig the abstract SmartGridObject (which extends the GridObject of JGM) class which has all the methods implemented to set and get any variables. The concrete objects only need to implement a few methods directly. Each instance of the SmartGridObject refers to a GridDataEnum which is the return value of the `getEnum()` method. Each GridDataEnum knows its associated object class and can generated instances of it with the factory method `SmartGridObject.factory(GridDataEnum)`.

GridObject is the basic type of any object in JGM. The main parameter is the position in the grid.

SmartGridSimObject is the basic object of SGS, holds all important functions for grid objects and extends GridObject.

ProSumer is a general term for energy consuming and/or energy producing facilities.

- **House** is the object for any type of energy consumer.
- **Powerplant** is subsumming all the energy producing objects. Production may depend on climate conditions.
  - **SolarPowerPlant** produces energy according to daytime and weather conditions.
  - **WindTurbinePowerPlant** produces energy depending on the wind speed.

GeoThermalPowerPlant produces energy independent of weather and time.

PowerTransport is the superclass for all grid objects used for power transportation.

SuperConductor works as a lossless connector between the other grid objects.

PowerLine represents a realistic connection between grid objects. It follows the π-equivalent circuit for power lines.

3.2.1 A set of variables for each object

All the variables of an object, which are all the physical properties needed in the simulation, are packed into the Variable Set (see the sgs.model.variables-package). `VariableSet` is a class, basically a Linked List, that manages all the `SingleVariables` of an object. That allows to use different variables for the same object according to the deployed algorithm. An UML diagram for VariableSet and SingleVariable classes is shown in figure 6. For a more complete version see figure 10.

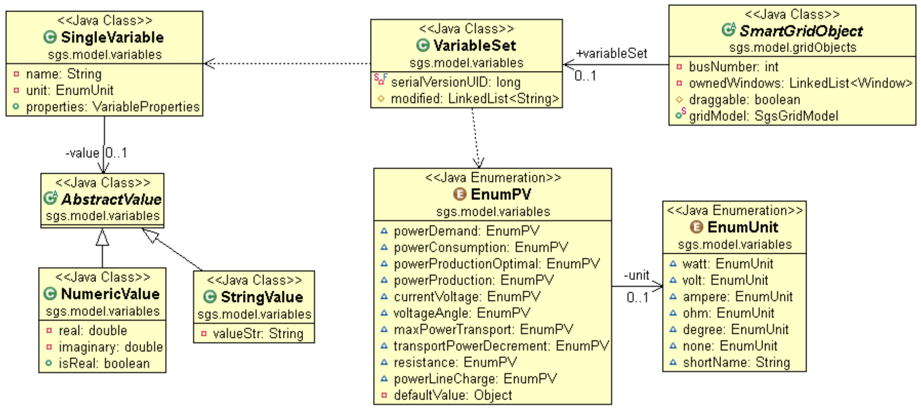


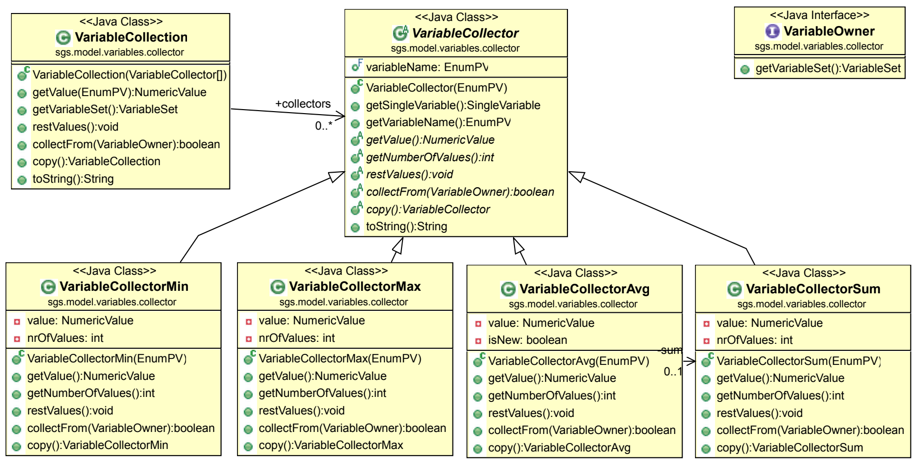
Figure 6: UML diagramm of the VariableSet, SingleVariables and their relation to SmartGridObjects.

Each variable has a name, a value, a unit (according ISO) and some properties. All possible names for variables are content of the `EnumPV` (Enumerator for Predefined Variables), i. e., what is not in the enum-list cannot be set as a variable. The possible units, listed in EnumUnit, are correlated to the names, e. g., for the variable with name currentVoltage the unit is volt. The properties of a variable define if it is visible and/or editable, e. g., in the property window. The values of a variable are instances of `AbstractValue`. The extending classes are `StringValue` and `NumericValue`. The `NumericValue` class implicates complex double precision numbers. The class internally distinguishes if it is a single or a complex value and is printed accordingly. It further provides a copy()-method and methods for calculations, e. g., mulitply(), divide() and conjugate().

3.2.2 Buses, Paths and Collectors

Several SmartGridObjects that are neighbours in the lattice or are connected lossless by supra conductors are part of the same `Bus` (in sgs.model.simulation). The connection between different buses is a `Path`, which is more or less equal to the involved PowerLine object and has the same resistance. The Network-Analyzer (in sgs.controller.simulation) is automatically generated all the Bus and Path objects, which are LinkedLists of Grid Objects. Those object lists are accessible via the GridModel instance. The NetworkAnalyzer is using the `resitanceAttributes` defined in the SgsGridModel. It currently cannot treat multiple resistances in a path (parallel connections) correctly.

Collectors and Collections are getting values from a list of objects, like Buses and Paths. That could be the average consumption of energy, the total load or the overall resistance. Available collectors are defined for the maximum, minimum, average and the sum of all contained values. An UML diagramm is shown in Figure 7. Collectors are in the sgs.model.variables.collectors package. To get details about usage of collectors see the code of the example algorithm.



<<Java Class>>
VariableCollection
sgs.model.variables.collector
VariableCollection(VariableCollector[])
getValue(EnumPV):NumericValue
getVariableSet():VariableSet
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollection
toString():String
+collectors
0..*
<<Java Class>>
VariableCollector
sgs.model.variables.collector
variableName: EnumPV
VariableCollector(EnumPV)
getSingleVariable():SingleVariable
getVariableName():EnumPV
getValue():NumericValue
getNumberOfValues():int
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollector
toString():String
<<Java Interface>>
VariableOwner
sgs.model.variables.collector
getVariableSet():VariableSet
<<Java Class>>
VariableCollectorMin
sgs.model.variables.collector
value: NumericValue
nrOfValues: int
VariableCollectorMin(EnumPV)
getValue():NumericValue
getNumberOfValues():int
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollectorMin
<<Java Class>>
VariableCollectorMax
sgs.model.variables.collector
value: NumericValue
nrOfValues: int
VariableCollectorMax(EnumPV)
getValue():NumericValue
getNumberOfValues():int
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollectorMax
<<Java Class>>
VariableCollectorAvg
sgs.model.variables.collector
value: NumericValue
isNew: boolean
VariableCollectorAvg(EnumPV)
getValue():NumericValue
getNumberOfValues():int
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollectorAvg
-sum
0..1
<<Java Class>>
VariableCollectorSum
sgs.model.variables.collector
value: NumericValue
nrOfValues: int
VariableCollectorSum(EnumPV)
getValue():NumericValue
getNumberOfValues():int
restValues():void
collectFrom(VariableOwner):boolean
copy():VariableCollectorSum

Figure 7: UML diagramm about the collectors.

3.2.3 Define the Color Overlay

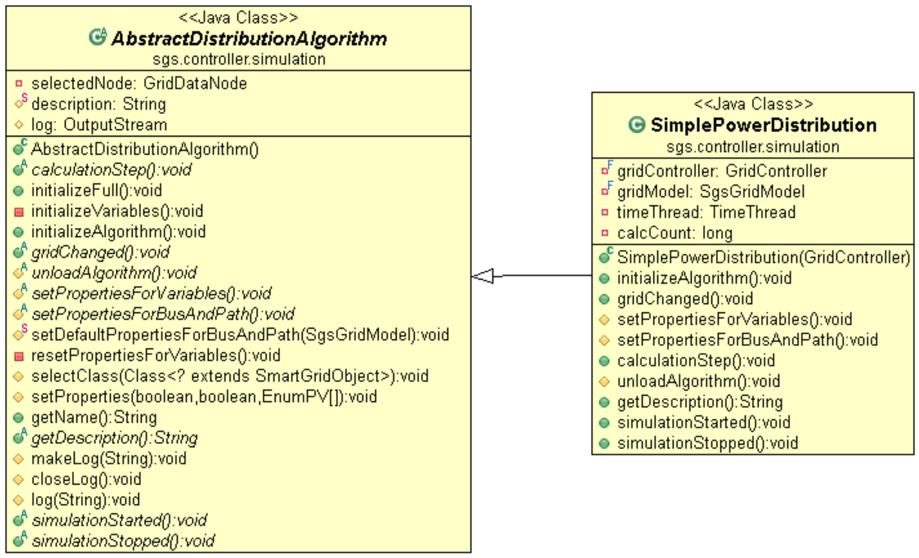
The Color Overlay is currently defined in the GridController.repaintData() method. For Houses is the overlay value defined as the ratio of (real part of) CurrentPowerConsumption and (real part of) PowerDemand. For all instances of PowerPlant it is the ratio of (real part of) CurrentPowerConsumption and (real part of) PowerProduction.

3.3 The Time Thread

The Time Thread (in sgs.controller.simulation) is managing the time progress of all the simulation. This includes the execution of the time steps, adjusting weather and climate data and termination of the simulation run. The the executed algorithm is also instanciated by the Time Thread.

3.4 Definition of an Algorithm

All the implemented algorithms need to extend the AbstractDistributionAlgoritm (like the ACPowerFlowCalculation in Figure 11 or the SimpleDistributionAlgorithm in Figure 8). An implemented algorithm, which is extending the AbstractDistributionAlgorithm is automatically recognized and added to the algorithm selection menu.



<<Java Class>>
AbstractDistributionAlgorithm
sgs.controller.simulation
selectedNode: GridDataNode
description: String
log: OutputStream
AbstractDistributionAlgorithm()
calculationStep():void
initializeFull():void
initializeVariables():void
initializeAlgorithm():void
gridChanged():void
unloadAlgorithm():void
setPropertiesForVariables():void
setPropertiesForBusAndPath():void
setDefaultPropertiesForBusAndPath(SgsGridModel):void
resetPropertiesForVariables():void
selectClass(Class<? extends SmartGridObject>):void
setProperties(boolean,boolean,EnumPV[]):void
getName():String
getDescription():String
makeLog(String):void
closeLog():void
log(String):void
simulationStarted():void
simulationStopped():void
<<Java Class>>
SimplePowerDistribution
sgs.controller.simulation
gridController: GridController
gridModel: SgsGridModel
timeThread: TimeThread
calcCount: long
SimplePowerDistribution(GridController)
initializeAlgorithm():void
gridChanged():void
setPropertiesForVariables():void
setPropertiesForBusAndPath():void
calculationStep():void
unloadAlgorithm():void
getDescription():String
simulationStarted():void
simulationStopped():void

Figure 8: Any algorithm needs to extend the AbstractDistributionAlgorithm.

In general an implemented algorithm holds all the calculations and also some specifications for the objects and collections (Bus and Path). In the following we state the five main tasks of the algorithm including the related methods in the AbstractDistributionAlgorithm (abstract methods are labeled with ^{A}).

1. Specify the SingleVariables and their attributes for each type of object. `resetPropertiesForVariables()`, `selectClass()`, `setProperties()`, ^{A}`setPropertiesForVariables()`

2. Specify the attributes which separates the Buses (Path objects) for the NetworkAnalyzer. ^{A}`setPropertiesForBusAndPath()`
3. Specify the variable Collections for Buses and Paths. ^{A}`setPropertiesForBusAndPath()`, `setDefaultPropertiesForBusAndPath()`
4. Distribute the power between the objects according to the definition of the algorithm. This includes:
   (a) Collection of the necesarry data from SmartGridObjects. ^{A}`initializeAlgorithm()`
   (b) Handle updates of the scenario. ^{A}`gridChanged()`
   (c) Do calculations or run a external solver. ^{A}`calculationStep()`
   (d) Write the data back to the SmartGridObjects. ^{A}`unloadAlgorithm()`
5. Create and write the Log-file for results. ^{A}`simulationStarted()`, ^{A}`simulationStopped()`, `makeLog()`, `closeLog()`, `log()`

In the following we explain the **abstract methods** in more detail. All the code examples are from the SimpleDistribution Algorithm which is described in 2.3.3.

Constructor: A special constructor is obligatory in the class as it will be called automatically (reflection) by the GridController. It requires exactly one GridController object as parameter.

calculationStep: This is the core of the algorithm - the calculation is done here.

setPropertiesForVariables: All the used variables and the user rights (Read and Edit) are set in this mehtod. A variable is "defined" as soon it is "editabel" **or** "readable" (An editable but not readable variable is considered as used (=defined, but not displayed to the user) which is important for the NetworkAnalyzer).

```
@Override
protected void setPropertiesForVariables() {
this.selectClass( PowerPlant.class );
this.setProperties(true, true, EnumPV.powerProductionOptimal);
this.setProperties(true, false, EnumPV.powerProduction);

this.selectClass( House.class );
this.setProperties(true, true, EnumPV.powerDemand);
this.setProperties(true, false, EnumPV.powerConsumption);

Out.pl("> Variables were set...");
}
```

This example uses two other methods from the parent class which are well commented there. The output is not necesary.

setPropertiesForBusAndPath: This method defines how Paths and Buses are identified by the NetworkAnalyzer and the Collectors. It is higly recommended to structure it in three parts like in the examplary code. The setDefaultPropertiesForBusAndPath method in the AbstractDistributionAlgorithm class contains more documentation.

```
@Override
protected void setPropertiesForBusAndPath() {
// --- resistanceAttributes ---
gridModel.resistanceAttributes.clear(); // no resistance attributes
// --- busVariableCollection ---
gridModel.busVariableCollection = new VariableCollection(
new VariableCollectorSum(EnumPV.powerConsumption),// result
new VariableCollectorSum(EnumPV.powerProduction),// Input 1
new VariableCollectorSum(EnumPV.powerDemand) // Input 2
);
// --- pathVariableCollection  ---
gridModel.pathVariableCollection = new VariableCollection();
}
```

initializeAlgorithm: Any initialization, necessary to run the algorithm, is done here.

unloadAlgorithm: No speciale use, can free external resources used by this algorithm.

gridChanged: Is called if the Grid is modified.

simulationStart: Called directly after the simulation was started. Use this to make a LOG file if one is used.

simulationStopped: Called directly after the simulation was stopped. Use this to flush/close a log file if one was used.

getDescription: Delivers a short explanation of the algorithm. It is the text which is shown when klicking the "Info" option within the algorithm menu of the GUI.

3.5 Definition of a Model

Models are part of Powerplant- and Consumer-Objects (Prosumer-Objects) and define the behavior of these objects. For each type of Prosumer-Object exists a model-type, like SolarPowerPlant-Models or WindTurbinePowerPlant-Models. It's also possible to define custom models which cannot be associated with a special type for example a model which depends on an external timeseries. Model types are realized as Abstract classes, which means that every Model has to extend a abstract model class. To make it easier the following enumeration contains five important steps to define custom models.

1. Take a look at the model hierachy (Fig.12) and choose the appropiate abstract model class.
2. Define the VariableSet which includes all attributes that are important for the model and should be visible and/or editable in the objects property window.
3. Specify the icon (which is visible in the property window), the name and the description for your model in the constructor.
4. Implement the updateModel procedure, which is called by the timethread to calculate the current power- production or demand and is also the interface between model and object.
5. Intitialize the variable-set of the model and set the initial values of the variables.

In the following a description of the constructor and methods, which have to be implemented in a custom model, is given.

Definition of attributes and class signature: Attributes which should be contained by the models variableSet should be declared as SingleVariable objects. Variables which are already defined in super-classes should not be definded a second time. In other words don't declare variables twice. Attributes which are only used in the model can have any type (like NumericValue).

```
public class SolarPeakPowerModel extends AbstractSolarPowerPlantModel{
private SingleVariable longitude;
private SingleVariable latitude;
private SingleVariable height;
private SingleVariable efficiency;
private SingleVariable squareMeters;
```

Constructor: The Constructor of your the gets an object of the type as input parameter. First of all, the super-constructor with the object as single argument must be called. Further the variables **modelName**, **description** and **icon**. have to be initialized.

```
public SolarPeakPowerModel(SolarPowerPlant powerPlant){
super(powerPlant);
modelName = "SolarPeakPowerModel";
description = "Das ist ein PeakPower Model";
icon = new ImageIcon("Data2/SolarPowerPlant_ICON.png");
}
```

Notice: The icon must be stored in the directory ”SmartGridSimulator/Data2” and should be about 50px high and about 50 px wide.

updateModel: The method **updateModel** has to contain the calculation of the current power production (powerplants) or power demand (consumers). This method is called from **TimeThread** in its **timestep**-method. Its important that the EnumPV-Variable (powerproduction or powerdemand) of the object is also set.

```
public void updateModel(GregorianCalendar currentTime
, Weather weather, int resolution){
squareMeters.setValue(peakPower.getValueDouble() /
(SolarCalculations.Wpeak_Intensity*efficiency.getValueDouble()));
double sunFactor = 1-weather.cloudFactor;
     double intensity;
     AzimuthZenithAngle sun = PSA.calculateSolarPosition(currentTime,
latitude.getValueDouble(), longitude.getValueDouble());
     intensity = SolarCalculations.getSolarIntensity(sun.getZenithAngle(),
height.getValueDouble());
     Out.pl("intensity="+ intensity + ",
pos="+latitude.getValueDouble()+"/"+longitude.getValueDouble()+",
height="+height.getValueDouble()+",
squareMeters="+squareMeters.getValueDouble()+", "+sun+"
,ratio="+intensity/SolarCalculations.Wpeak_Intensity);
        this.powerProduction =  new NumericValue(
squareMeters.getValueDouble() *
intensity*efficiency.getValueDouble() * sunFactor); \\
powerPlant.variableSet.get(EnumPV.powerProduction).
getValueNumeric().setReal(Math.round(this.powerProduction
.getReal()*10000)/10000.0);
         powerPlant.setPowerProductionOptimal(
this.peakPower.getValueNumeric().roundValue(4));
}
```

initVariableSet: This method is responsible for the initialization of the **SingleVariable**-objects, adding them to the models variableSet and specify whether they are visible and/or editable in the property window. The **initVariable** method is an already defined method and can be used like illustrated below.

```
protected void initVariableSet(){
this.longitude = this.initVariable("longitude", new NumericValue(14.4),
EnumUnit.degree, true, true);
this.latitude = this.initVariable("latitude", new NumericValue(46.6),
EnumUnit.degree, true, true);
this.height = this.initVariable("height", new NumericValue(0),
EnumUnit.kilometers, true, true);
this.efficiency = this.initVariable("efficiency", new NumericValue(0.2),
EnumUnit.percentDiv100, true, true);
this.squareMeters = this.initVariable("squareMeters", new NumericValue(0),
EnumUnit.squareMeters, true, true);
}
```

A Appendix

The appendix contains large figures, like UML diagramms, which would not fit in the text.

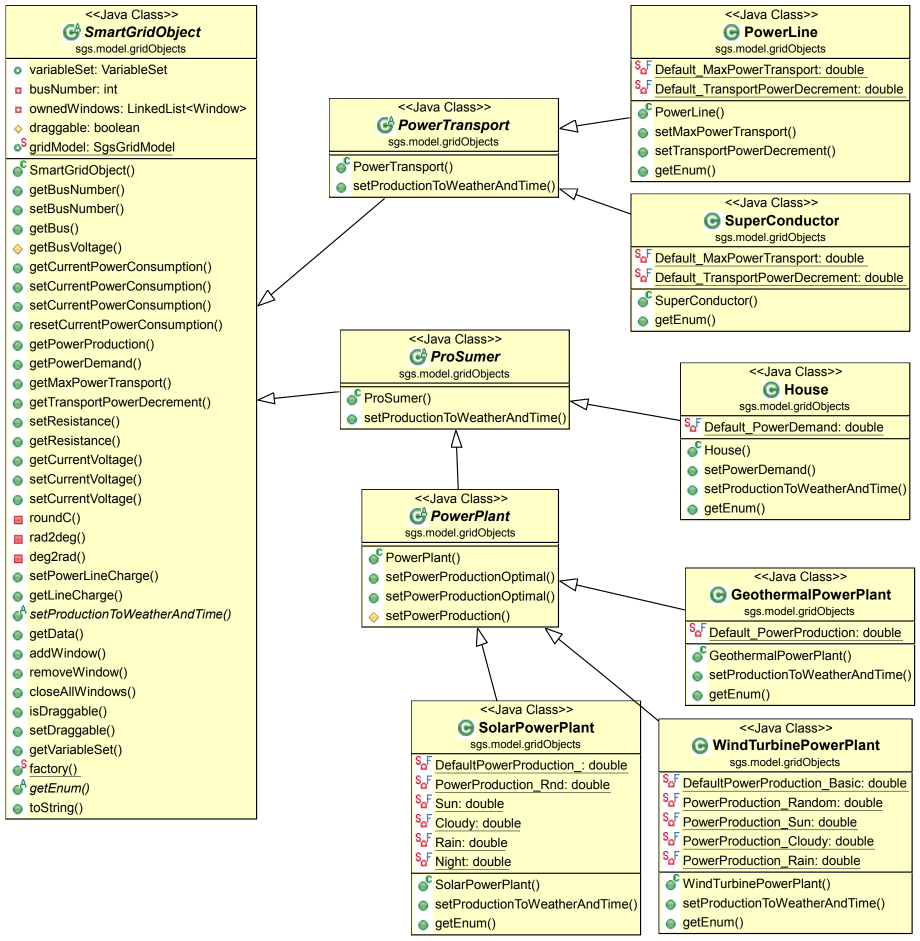


Figure 9: The UML diagram of the sgs.model.object package is a inheritance tree with the root parent SmartGridObject, which is extending GridObject of the JGM.

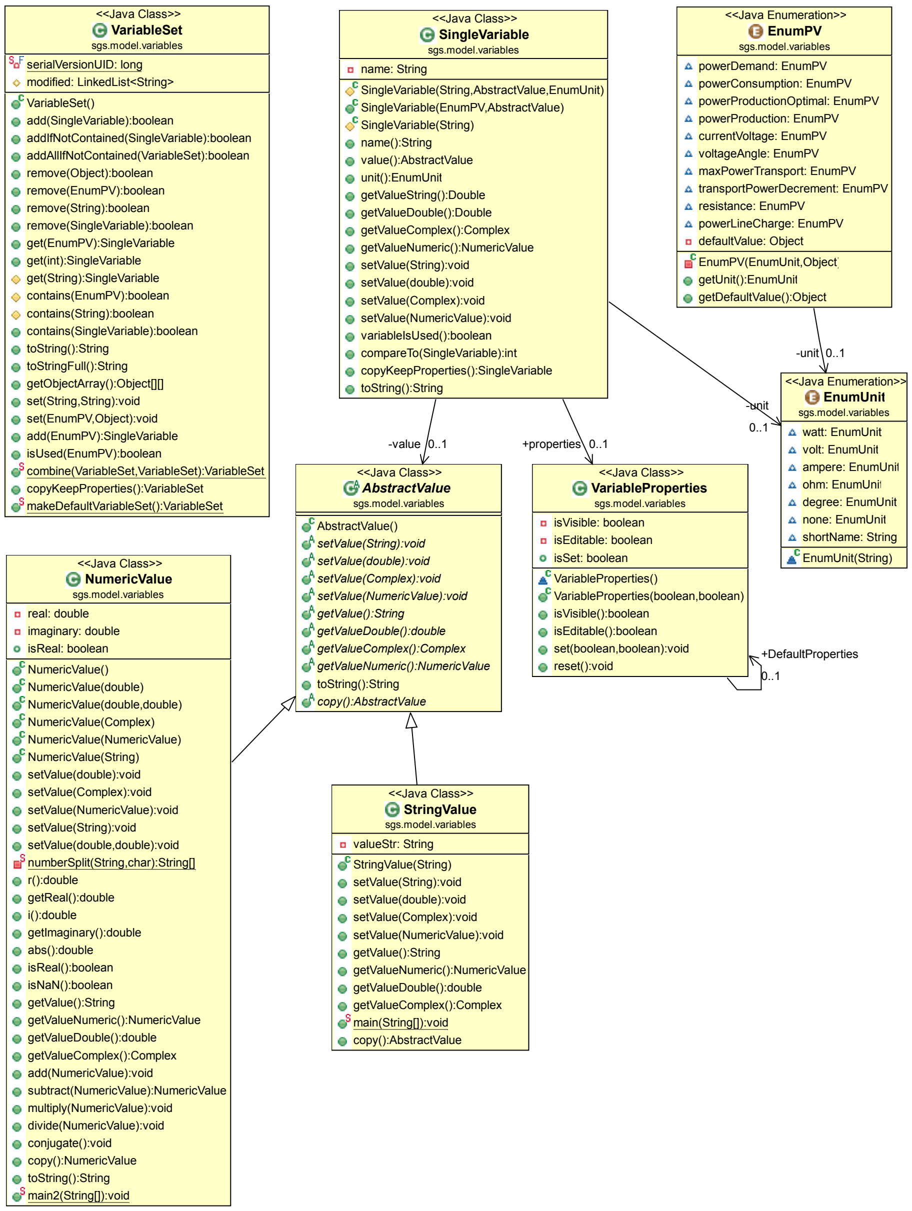


Figure 10: UML diagramm of the variable set of an object and the Single Variables.

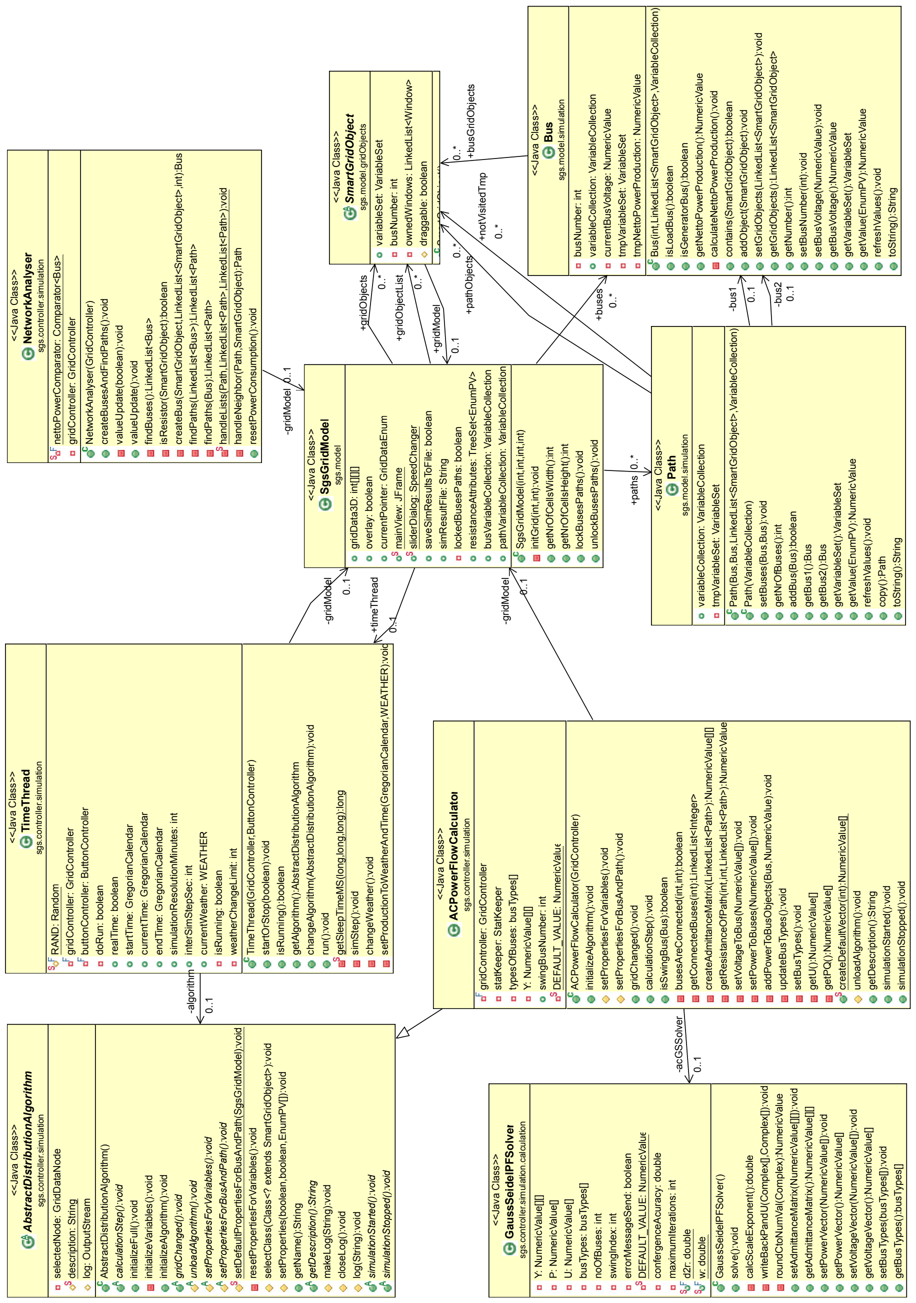


Figure 11: The SgsGridModel contains the SGObjects, runs the TimeThread as well as the NetworkAnalyzer and holds the Path and Bus lists.

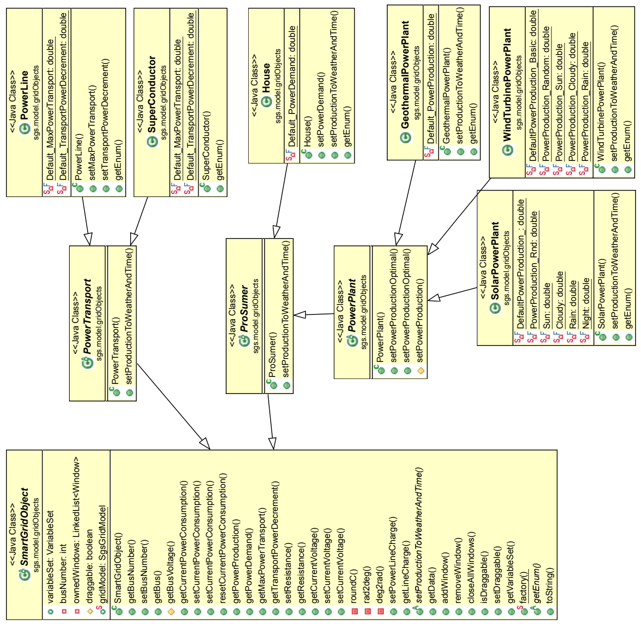


Figure 12: The Objects Model Hierarchy