

UPTEC W 02 015
ISSN 1401-5765

Examensarbete
M.Sc. Thesis work

Design of a simulator for the activated sludge process - JASS v3.0

Design av en simulator för aktivslamprocessen
- JASS v3.0

Gustav Grusell
June 2002

Abstract

Design of a simulator for the activated sludge process - JASS v3.0

Gustav Grusell, department of systems and control, Uppsala University

This thesis describes the work to improve a simulator for the activated sludge process. The simulator is denoted JASS (Java based Activated Sludge process Simulator) and was developed in a previous Master thesis work. The activated sludge process is a biological process used in wastewater treatment for degrading organic matter and also for removing nitrogen compounds. JASS uses the IWA Activated Sludge Model No. 1 to simulate the activated sludge process, and a fourth order Runge-Kutta method is used for numerical integration. A secondary settler is also included in the program, and it is simulated using a ten layer model. The main purpose of the work has been to improve the graphical user interface of the program with use of the Netbeans integrated development environment. Comments from users of the program have been used to improve the user-friendliness. Also, the flexibility of the simulator has been improved to make it easier to simulate different activated sludge process plants, and to implement new types of controllers.

Keywords: Simulator, Java, Netbeans, Activated Sludge Process, Control

Referat

Design av en simulator för aktivslamprocessen - JASS v3.0

Gustav Grusell, avdelningen för systemteknik, Uppsala universitet

Denna rapport beskriver det arbete som gjorts för att förbättra JASS (Java based Activated Sludge process Simulator). JASS är ett Java-program som simulerar aktivslamprocessen, vilken är en biologisk process som används vid avloppsvattenrening för att avlägsna organiskt material och kväveföreningar. JASS använder ASM1-modellen för att simulera aktivslamprocessen, och en fjärde ordningens Runge-Kutta-metod används för de numeriska beräkningarna. En sedimenteringsbassäng simuleras också med hjälp av en sedimenteringsmodell med tio lager. Huvudsyftet med arbetet har varit att förbättra det grafiska användargränssnittet med hjälp av Netbeans, vilket är en grafisk utvecklingsmiljö som kan användas med Java. Kommentarer från deltagare på kurser där programmet använts har utnyttjats för att förbättra användarvänligheten. Programmets flexibilitet har också förbättrats för att möjliggöra simulering av olika typer av aktivslamprocesser med olika typer av regulatorer.

Nyckelord: Simulator, Java, Netbeans, Aktivslamprocessen, styrning

Preface

Main supervisor :

Bengt Carlsson, Department of Systems and Control, Uppsala University

Co-supervisors :

Pär Samuelsson, Department of Systems and Control, Uppsala University

Mats Ekman, Department of Systems and Control, Uppsala University

Examiner :

Bengt Carlsson, Department of Systems and Control, Uppsala University

Thanks to the supervisors and Ove Ewerlid for valuable help.

This work is sponsored by Svenska Kommunförbundet and Svenskt Vatten.

Copyright ©Gustav Grusell and Department of Systems and Control,
Uppsala University UPTEC W 02 015, ISSN 1401-5765
Printed at the Department of Systems and Control, Uppsala University,
Uppsala 2002

Contents

1	Introduction	5
1.1	Objectives	5
1.2	Introduction to wastewater treatment	5
1.2.1	Using the activated sludge process for nitrogen removal	6
1.3	Previous versions of JASS	7
2	JASS 3.0 design	9
2.1	Tools	9
2.1.1	Java	9
2.1.2	NetBeans IDE	10
2.2	Design goals	11
2.3	The simulator	11
2.4	The graphical user interface	12
2.4.1	Presentation of simulation results	13
2.4.2	Presentation and input of simulation properties	13
2.4.3	Controlling the simulation	15
3	JASS 3.0 implementation in Java	16
3.1	The simulator	16
3.1.1	Controllers	16
3.1.2	FlowGenerators	19
3.1.3	Samplers	21
3.1.4	IAWQCompartment	23
3.1.5	IAWQSettler	26
3.1.6	JassModel	26
3.1.7	The Simulator class	29
3.1.8	Loading and saving models	29
3.1.9	Other classes	30
3.2	The graphical user interface	31
3.2.1	The class BarDiagram	31
3.2.2	The class GGPlot	32
3.2.3	The class RTPlotPanel	33
3.2.4	The class BarDiagramPanel	33
3.2.5	The class TreatmentResultPanel	33
3.2.6	The class ProcessImageComponent	34
3.2.7	The class GuiMain	35
3.2.8	The Dialog classes	35
3.2.9	Other classes	36
3.2.10	Multiple language support	37

3.2.11	Initialization of the GUI in class SimulationController .	37
3.3	Multi Threading	38
3.4	Communication between the GUI and the simulator	38
3.4.1	The simulation controller	39
3.4.2	The JassModel update queue	39
3.5	Jass3 : The main class	40
4	Conclusions	41
A	Models for the activated sludge process and the settler	43
A.1	The IWA activated sludge model no 1	43
A.1.1	Organic matter	43
A.1.2	Different fractions of nitrogen and other compounds . .	44
A.1.3	Biological and chemical reactions	44
A.2	ASM1 process matrix	49
A.3	Process parameters in the simulator	50
A.4	The modelling of one mixed zone	51
A.5	Modelling of the settler	53
B	Users manual	54
B.1	Introduction to JASS	54
B.1.1	ASP models used in JASS	54
B.1.2	The GUI	54
B.1.3	Running the simulator	55
B.1.4	Displaying simulation data	55
B.1.5	Changing simulator properties	56
B.1.6	The controllers	57
B.2	Working with JassModels	60
B.2.1	Introduction to JassModels	60
B.2.2	Creating new models	60
B.2.3	Implementing new controllers	61
B.3	Compiling JASS 3.0	63
C	JASS 3.0 package and classes overview	65

1 Introduction

1.1 Objectives

The purpose of this master thesis work has been to improve a simulator for the activated sludge process denoted JASS (Java based Activated Sludge process Simulator). The main idea was to improve the graphical user interface (GUI) by using the Netbeans IDE. Also the general structure of the program needed to be improved in order to be more general and easy to modify. As the work progressed, other ideas for improvement also appeared.

1.2 Introduction to wastewater treatment

This section gives a brief introduction to wastewater treatment. For more information on the subject, see (Tchobanoglous and Burton 1991).

The main purpose of a typical wastewater treatment plant is to remove suspended solids, floating particles, organic material and also phosphorus and nitrogen compounds from the water. A typical approach is to do this in three steps:

- Mechanical treatment
- Biological treatment
- Chemical treatment

In the mechanical treatment step, larger objects are removed by a grid. The grid is followed by a sand trap that removes heavy particles like sand and gravel. The last step in the mechanical treatment is a settler where suspended solids are allowed to settle.

The main purpose of the biological treatment step is to remove solved organic material from the waste water. This is accomplished by micro organisms, usually bacteria. With modifications, the biological step can also be used to remove nitrogen and phosphorus compounds. The biological step also includes a settler where microorganisms and particles can settle. One of the most common processes for biological treatment is the *activated sludge process* (ASP). In its simplest form, it is implemented by adding air to the wastewater, and thus making it possible for aerobic bacteria to decompose organic matter. The activated sludge process can also be modified for nitrogen removal, as described in section 1.2.1.

Chemical treatment can be used to remove dissolved phosphorus from the wastewater. This is done by adding chemicals that converts the solved

phosphorus to insoluble compounds and also stimulates flocculation. The flocks can then be removed by sedimentation or flotation.

The sludge that is removed from these three steps needs further treatment. The first step is thickening, in which water is removed from the sludge, which has a high water content. Thereafter the sludge from the mechanical and biological treatment, that is rich in organic material, will have to be stabilized to reduce odor and kill micro organisms. This can be done in a number of ways, for example in an organic digester.

1.2.1 Using the activated sludge process for nitrogen removal

A basic activated sludge process is shown in Figure 1. The first step is an aerated tank, in which microorganisms use dissolved oxygen to oxidize organic matter, and thus gain energy for growth. In this process, the carbon in the organic matter is either converted to carbon dioxide or incorporated in the microorganisms. The formed biomass can then be separated from the water in a settler. Some of the sludge collected in the settler is usually recirculated into the treatment basin, to keep enough biomass to decompose the incoming organic matter.

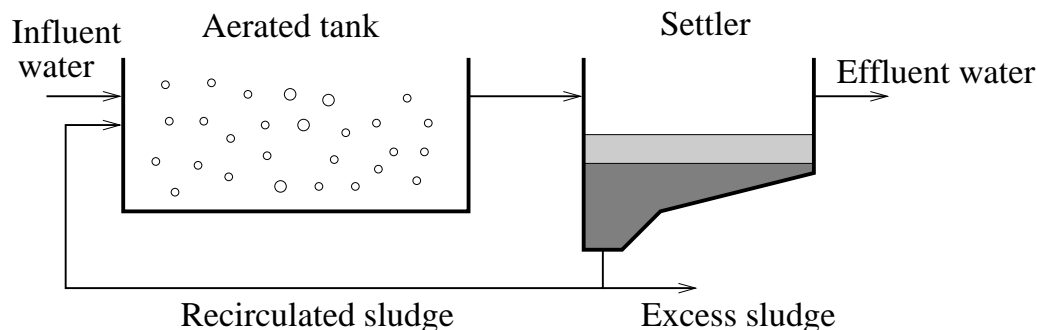


Figure 1: An activated sludge process with one aerated tank and a settler

Most of the nitrogen in wastewater is in the form of ammonium, NH_4^+ . Under aerobic conditions, NH_4^+ can be oxidized by microorganisms into nitrate, NO_3^- . This is called nitrification. It is however also desirable to minimize the concentration of nitrate in the effluent water, since this is often a limiting nutrient in the recipient. In order to reduce the nitrate concentrations, it can be converted to gaseous nitrogen, N_2 , by microorganisms under anoxic conditions. This process, is called denitrification and can be used in the activated sludge process.

In order to remove as much nitrogen as possible from the wastewater, both aerobic and anoxic conditions are needed. To use this in the activated

sludge process, two types of plant configurations are common, these are called post denitrifying and pre denitrifying.

In a postdenitrifying system, for example the one shown in Figure 2, the aerobic tanks are put before the anoxic ones. This may lead to a shortage of readily degradable organic matter in the anoxic zones, which may limit the denitrification. To get good efficiency it may therefore be necessary to add carbon from an external source to the anoxic zones to keep a high efficiency on the denitrification.

In a predenitrifying system, on the other hand, the anoxic zones are located before the aerobic, and therefore the organic matter in the influent water can be used for denitrification. The drawback with this strategy is that it is necessary to recirculate nitrate rich water from the last zone to the first in order to get good efficiency in the nitrogen removal.

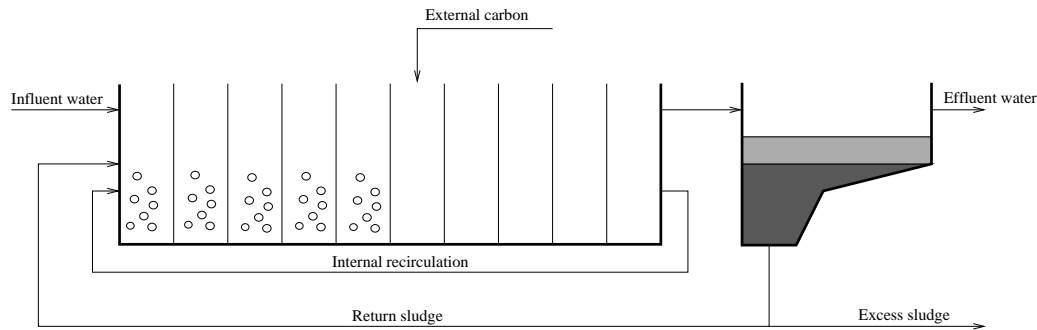


Figure 2: A postdenitrifying plant with 10 compartments and external carbon

1.3 Previous versions of JASS

JASS is a Java program that simulates the activated sludge process in a wastewater treatment plant. Simulation data is presented graphically to the user, and properties of the process can be changed interactively. The program is written as an Java applet and can therefore be run through a standard web browser. Version 1.0.0 of the Java Activated Sludge Simulator (JASS) was finished in March 1998 as a Master thesis work, see (Samuelsson 1998).

Since the completion of JASS 1.0 it has been used in courses at the Department of Systems and Control, both for students and wastewater treatment plant operators. The program has performed well and has been found easy to learn and use. Version 2.0 was finished in November 2000, see (Samuelsson *et al.* 2001). The main improvement was the introduction of a ten-layer model of the settler (in JASS 1.0 ideal settling was assumed).

Other improvements were the introduction of a supervisory DO controller and improved presentation of treatment results.

2 JASS 3.0 design

2.1 Tools

This section describes the tools used in this work: the programming language Java, and Netbeans. Netbeans is an integrated development environment that can be used for Java as well as for other programming languages.

2.1.1 Java

Java was first released by SUN in 1995. At first it was mainly used for web programming with applets, but has since then gained much popularity also for other applications. This is because it is easy to use, powerful, and, probably most important, platform independent. Platform independence comes from the fact that when a Java program is compiled, it is not compiled to machine code, but rather to an intermediate language called Java byte-code. The program is then executed by running a *Java virtual machine* that interprets the byte-code to machine code and executes it. The Java virtual machine, of course, is platform specific. It is available for all major platforms and many minor ones as well.

Java is an *object oriented* programming language. This means that a Java program works with *objects*, which is a collection of data and methods to manipulate the data. An object is defined by a *class*. A class defines a set of data (*fields*) and methods. Each field and method has an *access modifier*. The access modifier decides which classes can access the field or method. It may be *private* in which case it is only accessible inside the class, *protected*, which gives access from inside the class and its subclasses (subclasses are explained below), *package*, which gives access to all classes that are in the same *package* (packages are explained below), or *public*, which gives no restrictions on the access. This makes it possible to write robust program, since the class may decide exactly how its data can be modified by disallowing direct access to the data, and instead providing a set of methods for manipulating it. Also note that classes themselves have access modifiers.

Creating an object from a class is called *instanciating* the class, or to create an *instance* of the class. This is done by using the Java `new` keyword, or by calling `Class.getInstance` on the `Class` object of the class.

A powerful concept in object-oriented programming is *inheritance*. A class may inherit another class, in which case it is said to be a *subclass* of that class. The class that it inherits is said to be its *superclass*. A subclass inherits all of its superclass fields and methods, and may define new ones as well. In this way it is easy to write a set of classes that shares some

functionality, and it also provides an easy way to re-use code. In Java, classes and methods may be defined as *abstract*. An abstract method does not define any functionality, it only defines its *signature*, that is its name, parameters and return type. A class that has one or more abstract methods must be defined as abstract. An abstract class can not be instantiated, but may be used as a superclass. Non abstract subclasses must then define the functionality of the abstract methods.

Another Java concept that somewhat resembles inheritance is the *interface* concept. An interface is a collection of methods. Only the signatures of the methods are defined in the interface. A class may then *implement* the interface by defining all methods in the interface. Even though the functionality of the methods are not defined in the interface, for it to be useful they should have a intended functionality that should be implemented by the implementing class.

Java supports *multithreading*, which is a way to let the program perform more than one task at once. Multiple threads may execute the same code and modify the same data. In multithreaded applications, it is important to ensure that multiple threads do not modify the same data at the same time, since this can lead to unpredictable results.

More information on Java can be found in (Skansholm 1987).

2.1.2 NetBeans IDE

NetBeans IDE is an *Integrated Development Environment* that can be used for programming in Java, as well as other languages like C++ and XML. It contains a code editor and a debugger and allows for visual design of Java components. It also has many other features. As for the development of JASS 3.0, the visual design tools were the most important feature. This feature allows the user to visually create and design GUI elements. This is a very powerful feature, especially for users who have little experience of GUI programming in Java.

When graphically creating and editing GUI's, the IDE auto-generates the Java code that represents the GUI. This gives the user a good view on what's being done, and is also a very good way to learn Java GUI programming. It is not possible to directly modify the auto-generated code in the code editor, but it can be changed indirectly by using GUI design tools. It is also possible to create new user components in the form of JavaBeans. These components can then be used in the GUI in the same way as the standard GUI components.

2.2 Design goals

The goals of the design were the following:

1. To make the program more user friendly.
2. To make the program easy to maintain and modify.

To achieve the first goal it is necessary to have a graphical user interface that is clear and easy to understand. The second goal requires a good program structure. Fortunately, object oriented programming with Java gives good support for this.

2.3 The simulator

The simulator part of JASS 3.0 (from now on simply called the simulator) simulates the activated sludge process of a wastewater treatment plant. The simulator has been designed to be as flexible as possible, in order to allow simulation of plants with different properties. Figure 2 shows a typical plant layout with ten compartments and a settler. The plant shown implements post denitrification and external carbon is added to the first non-aerated compartment. The just mentioned properties are typical and may differ in plant setups that can be simulated with JASS 3.0. The only requirement from the simulator on the plant setup to be simulated is that it has at least one compartment and a settler. The plant setup can be arbitrary in the following respects:

- The number of compartments and their volumes.
- The settler area and height.
- External carbon may be added in any compartment.
- Internal recirculation flow, return sludge flow and excess sludge flow may be present.
- The parameters used in the models for simulating the ASP and the settling may be set to any physically meaningful value.
- The influent water may be fed into one or more compartments. If multiple inputs are present, the influent concentrations in all of them will be the same, although the flow rates may vary.
- All flow rates may be controlled by a controller, and controllers may be cascaded. Only PID controllers have been implemented, but it is possible to implement other types and use them with the simulator.

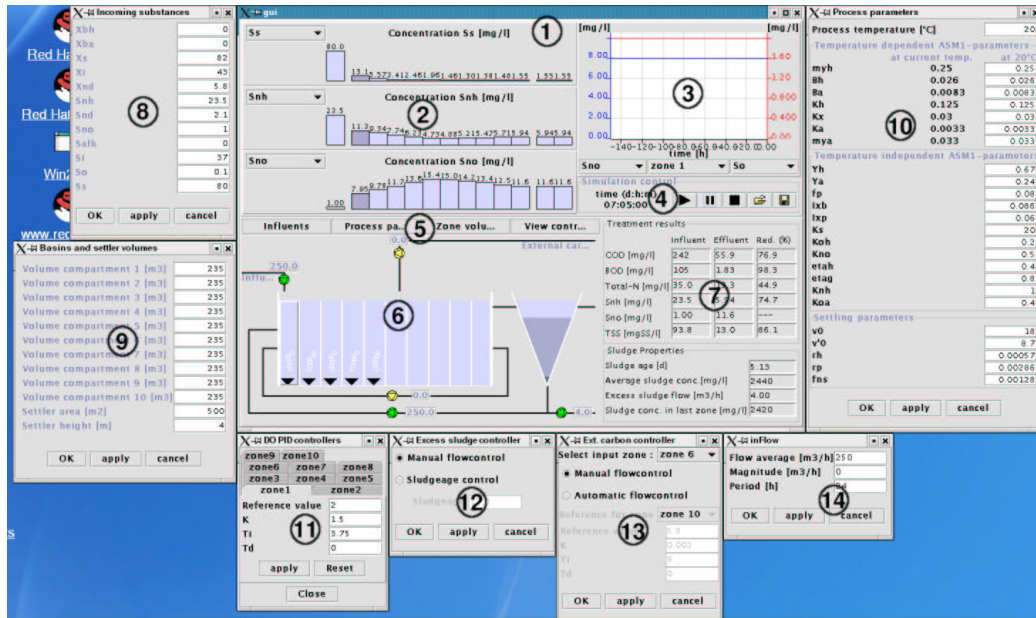
Some of the above mentioned properties of the plant setup can be changed in the GUI (see Section 2.4.2) while the others can easily be changed with the use of Java programming (see Appendix B.2).

Each compartment in the simulated ASP is regarded as a completely mixed reactor and is modelled with the IWA (International Water Association) Activated Sludge Model no. 1 (ASM1). This is described in more detail in Appendix A.1 and A.4. The model used for simulating the settler is described in Appendix A.5.

2.4 The graphical user interface

Figure 3 shows the graphical user interface with some dialog windows open. Most of the main GUI window has been designed to look like the GUI of

Figure 3: The GUI, with some dialogs



previous versions of JASS, but some changes has been done. The GUI of the previous version of JASS has been thoroughly tested and most parts found to be clear and intuitive. One improvement of the new GUI is that multiple language support has been added, which makes it easy to make the GUI available in many languages. Resources for a Swedish and an English version are included in the program. Also the GUI's for controllers have been improved as well as a number of other GUI functions.

The GUI has four functions: To present data from the simulation, to present parameters for the simulation, to allow online user input of parameters for the simulation, and to allow online user input for controlling the simulation. The GUI can be said to consist of two parts: the main GUI window and a number of dialogs. The number of dialogs may vary depending on the setup of the simulated plant. More about the GUI can be found in the users manual in Appendix A.5.

2.4.1 Presentation of simulation results

Presentation of simulation results is done in the main GUI window only, and is done in four different ways:

The bar-diagrams There are three bar diagrams, collectively numbered 2 in Figure 3. Each of these shows the concentration of a (user selectable) component in the influent water, all compartments, the return sludge and the effluent water. They may also be set to show airflow rate or carbon flow rate.

The real time plot The plot, numbered 3 in Figure 3 shows the values of two different variables (concentration of a component, carbon flow rate or air flow rate) in one location, which can be the influent water, any compartments, the return sludge or the effluent water. The plot is updated in real time and shows the values for the last 150 hours, so it is possible to study changes of a variable over time.

Treatment results and sludge properties In the upper part of the panel numbered 7 in Figure 3, the concentration of some important components in the influent and effluent water are displayed, together with the reduction in percent from influent to effluent. In the lower part some properties of the sludge in the settler are displayed.

Sludge concentration in the settler The sludge concentration in the different layers in the settler is presented as different colors in the settler image, located to the right of the process image (number 6 in Figure 3). Darker color means higher sludge concentration in the layer.

2.4.2 Presentation and input of simulation properties

The simulation properties can be divided into two groups: those for the setup and control of the simulated plant, and those for the model used for simulat-

ing the ASP and the settling in the settler. The latter groups consist of the parameters of ASM1 (see appendix A.1) and the settler model parameters (see appendix A.5). The former group is somewhat larger, and can roughly be divided in four parts: physical properties of the plant (compartment and settler volumes), properties of the influent water, configuration of the controllers in the plant, and flow rates into, out of and internally in the plant.

Process parameters With process parameters, we mean the parameters used by the models for simulating the ASP and the settling in the settler. Those parameters are displayed and can be changed in the process parameters dialog, numbered 10 in Figure 3. This dialog is displayed by clicking the “Process parameters” button, located in the panel numbered 5 in Figure 3.

Physical properties of the plant The compartment volumes, and the settler area and height, are displayed in the “Zones and settler volumes” dialog, numbered 9 in Figure 3. This dialog also allows the user to change these values. To display this dialog, the user clicks the “Zone volumes” button in the main GUI window, located in the panel numbered 5 in Figure 3.

The influent water The concentration of the components in the influent water are displayed and can be changed in the “Incoming substances” dialog, which is displayed by pressing the “Influents” button, located in the panel numbered 5 in Figure 3.

Configuration of the controllers Clicking the “Controllers” button, located in the panel numbered 5 in Figure 3, brings up a popup menu where the names of the dialogs for configuring controllers are listed. Clicking an item in the menu brings up the corresponding dialog. The number of items in the menu may vary, since this depends on the setup of the plant being simulated. The dialogs numbered 11, 12 and 13 in Figure 3 shows some common dialogs for configuring controllers. Note that the dialog numbered 11 allow configuration of multiple controllers, while the other dialogs correspond to one controller each. The parameters that can be set for each controller depends on the type of controller.

Setting the flow rates Displaying and setting the magnitude of the different flow rates is done in flow dialogs, like the one numbered 14 in Figure 3. When a pump symbol in the process image (number 6 in Figure 3) is clicked, a flow dialog corresponding to that pipe is displayed. Flow dialogs may differ

in appearance, the one in Figure 3 allows for setting a flow rate that varies in time in a sinusoidal shape. Others may allow only constant flow rates to be set.

The process image In the process image (number 6 in Figure 3), the status of aeration is presented: in aerated compartments, an air valve and some bubbles will be displayed. Also the colors of the pumps indicates the status of the different flows: a green pump means a nonzero flow rate, a yellow means a zero flow rate, and a Gray (disabled) means that the flow rate is being automatically controlled by a controller

2.4.3 Controlling the simulation

The simulation is controlled with the five buttons located in the *Simulation control* panel: *start*, *pause*, *stop*, *load* and *save*. With these buttons the simulation can be started and stopped, and different plant setups can be saved and loaded.

3 JASS 3.0 implementation in Java

This section describes how Java has been used to implement the functionality of the program, as described in the previous section. Three different parts of the program: the simulator, the GUI, and the communication between these will be described. Two lists with short descriptions of all packages and classes can be found in appendix C.

3.1 The simulator

The classes making up the simulator is found in the package `jass3.simulation` and its subpackages. In this section the different classes, their use and implementation is described.

3.1.1 Controllers

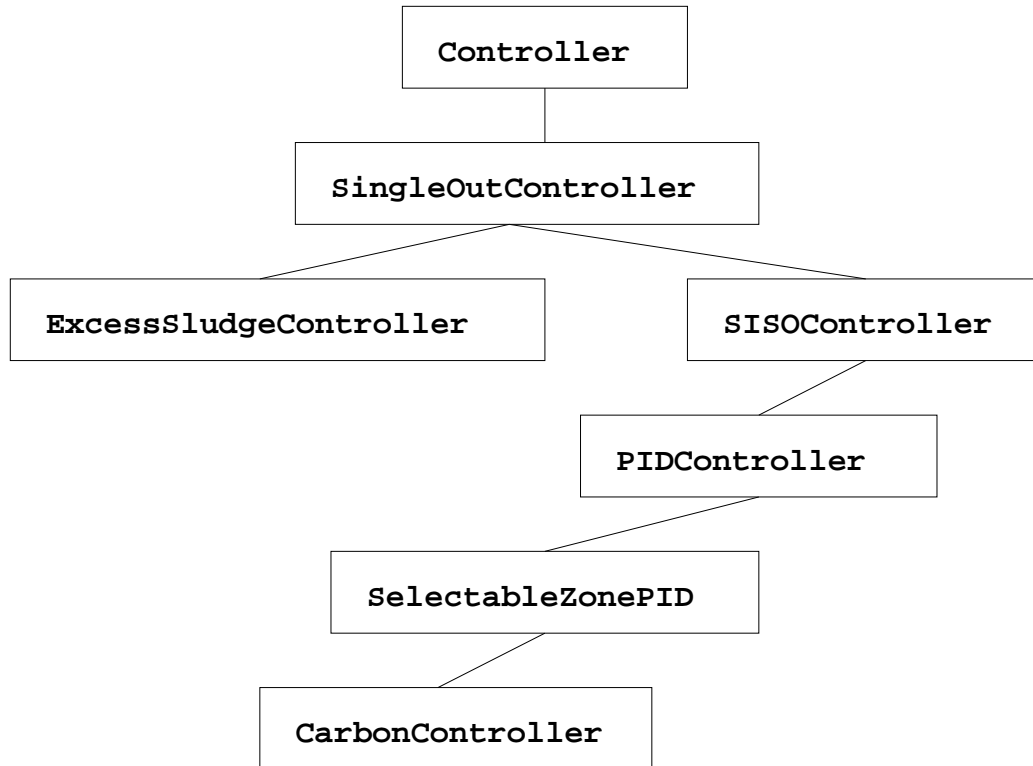


Figure 4: Class hierarchy of the controller classes

The Java classes representing controllers are located in the package `jass3.simulation.controller`. Figure 4 shows the class hierarchy of the

implemented controller classes. Three of the classes are defined as abstract. This means that they are not instanciable, they are meant to be used as superclasses. They are present to make it easier to implement new types of controllers. The abstract classes are `Controller`, `SingleOutController` and `SISOController`.

For a controller class to allow user input, it needs to provide two more classes: a gui class that should be a subclass of `ControllerGui` and a parameter class (preferably an inner class) that should be a subclass of `Controller.Parameters`. The parameter class should encapsulate the parameters of the controller that can be changed, and is used for passing the parameters to and from the controllers gui. The class `Controller.Parameters` is just an empty implementation. `ControllerGui` inherits `javax.swing.JPanel` and provides abstract methods for passing parameters to and from the gui.

Below is a description of the controller classes and the functionality they provide. Note that all of these classes except `SISOController` also has an inner parameter class, named `<classname>.Parameters`, and that those classes that are non-abstract also has a gui class, named `<classname>Gui`.

The abstract Java class `Controller` is the superclass of all controller classes. The only functionality in this class is for setting a name on a controller. The name may be an internationalized string located in a resource file, in which case the name will be reread from the resource file for the current locale if a serialized controller is deserialized.

`SingleOutController` is another abstract class and is the superclass of all controllers implemented in JASS 3.0. It is a little bit more limited than the name may imply. It is not only limited to one output, but also may control only one variable i.e. it has only one reference. However, it may have more than one input. A `SingleOutController` may work as a slave controller of another `SingleOutController`.

`SISOController` is an abstract class representing a single input, single output controller. It adds functionality for receiving a sample from a sampler.

The class `PIDController` represents a discrete time PID controller. It is implemented as a differential form PID with limits u_{max} and u_{min} on the output, which means the control signal u is calculated as shown in (1) and (2):

$$\Delta u_n = K \left(e_n - e_{n-1} + \frac{h}{T_i} e_n + \frac{T_d}{h} (y_n - 2y_{n-1} + y_{n-2}) \right) \quad (1)$$

$$u_n = \begin{cases} u_{max} & \text{if } u_{n-1} + \Delta u_n > u_{max} \\ u_{n-1} + \Delta u_n & \text{if } u_{min} \leq u_n \leq u_{max} \\ u_{min} & \text{if } u_{n-1} + \Delta u_n < u_{min} \end{cases} \quad (2)$$

where y is the value of the controlled variable, e is the control error, and h is the sampling interval. Subscript n means at sample instant n . K , T_i and T_d are the PID parameters. When $T_i = 0$, it is interpreted in JASS as $T_i = \infty$ and u is calculated as in (3)

$$u_n = K e_n - \frac{T_d}{h} (y_n - y_{n-1}) \quad (3)$$

The calculation of the control signal is implemented in the Java method `newInput`.

The class `SelectableZonePID` inherits `PIDController`. The added functionality is that when this class is used with a sampler for collecting data, the user can choose which zone to control, i.e. from which zone the sample is taken.

The class `CarbonController` inherits `SelectableZonePID`. As the name implies, it is intended for controlling external carbon, but may be useful for other things as well. When used for controlling a `FlowGenerator` (see section 3.1.2), it allows the user to select the zone where the flow of the `FlowGenerator` should be added. This controller may also be turned off in its gui, to allow the user to set the flow manually.

The class `ExcessSludgeController` represents a specialized controller for controlling the excess sludge flow. It uses the mass-balance equation (4) to calculate the excess sludge flow from the desired sludge age and four sampled values. The four sampled values are $V_a \cdot SS_m$, Q_{out} , SS_{out} and SS_e .

$$Q_e = \frac{V_a \cdot SS_m \cdot \frac{1}{\theta} - Q_{out} \cdot SS_{out}}{SS_e} \quad (4)$$

where

Q_e = excess sludge flow in m^3

V_a = total volume of the aerated compartments, in m^3

SS_{out} = SS (Suspended Solids) concentration in the effluent water, in $g\ SS/m^3$

SS_m = mean sludge concentration in the aerated compartments, in $g\ SS/m^3$

θ = sludge age set-point, in days

Q_{out} = Effluent flow, in m^3

SS_e = SS concentration in the excess sludge, in $g\ SS/m^3$

3.1.2 FlowGenerators

The class `FlowGenerator` and its subclasses are used for controlling flow rates in the ASP. It can be thought of as an abstract pipe with a valve, that can be controlled manually or automatically by a controller. Every `FlowGenerator` has an integer field named `type` that identifies the type of flowgenerator, or, to be more precise, the fields in the proper `IAWQCompartment` where the rate and state of the flow should be put (see Section 3.1.4). The possible values for the `type` field are presented in Table 1. Two fields named `inputLocation` and `outputLocation` are also present. `inputLocation` indicates the origin of the flow represented by the `FlowGenerator`, and `outputLocation` the destination. The possible values for these are the same as the possible sample locations presented in Table 2, except that the influent water can neither be the origin nor the destination of the flow, and the settler can not be the destination. A `FlowGenerator` also holds the state of the water in the represented flow. This state is only used on certain occasions, this is described below.

In JASS 3.0, `FlowGenerators` are used for four purposes:

- For generating influent water to the plant. In this case, the `FlowGenerator` not only controls the flow rate, but also holds the influent concentration of the ASM1 components. The field `outputLocation` holds the zone where the influent water enters the plant. The `type` field should be set to `TYPE_IN`. The `inputLocation` field should be set to -1, meaning the water comes from outside the plant. When used for this purpose, the `FlowGenerator` holds the state of the influent water. Multiple `FlowGenerators` of this type may be used to implement step feed.
- For return sludge flow and internal recirculation. The `type` field should then be set to `TYPE_PRE` or `TYPE_REG` respectively. The `inputLocation` field should be set to the location of the origin of the flow, which would be the last zone or the settler bottom layer, respectively. The output of the flow would in both cases be the first zone. When used in this way, the `FlowGenerator` controls the flow rate, and the state of the water is taken from the origin of the flow, so the state held by the `FlowGenerator` is not used.
- For external carbon flow. The `type` field should then be set to `TYPE_CAR`, and the `inputLocation` field to the number of the zone where the carbon is put. The `outputLocation` field should be set to -1. In this case, only the flow rate is controlled. The COD of the carbon source is set in

the constant `ETHANOL_KONC` in the class `IAWQCompartment`. The state held by the `FlowGenerator` is not used.

- For excess sludge flow. In this case, the `type` field should be set to `TYPE_REG`. The `inputLocation` should be the to the settler bottom layer, and the `outputLocation` to -1, meaning the destination is outside the plant. The state of the flow does not matter in this case, since the water goes out of the simulated plant anyway.

value	constant	meaning
0	<code>FlowGenerator.TYPE_IN</code>	Represents an input
1	<code>FlowGenerator.TYPE_PRE</code>	Represents internal recirculation
2	<code>FlowGenerator.TYPE_REG</code>	Represents return sludge or excess sludge
3	<code>FlowGenerator.TYPE_CAR</code>	Represents external carbon

Table 1: `FlowGenerator` types

In order to allow the user to change the flow, a subclass of `FlowGenerator` should, in much the same way as a subclass of `Controller`, provide a GUI class and a parameter class. The GUI class should be a subclass of `FlowGeneratorGui` and the parameter class a subclass of `FlowGenerator.Parameters`.

Three `FlowGenerator` classes has been implemented as a part of JASS 3.0. These are `FlowGenerator` and two subclasses, `SinusFlowGenerator` and `ControlledFlowGenerator`.

FlowGenerator This class is the superclass of the two others. There are also a GUI class and a parameter class for this class, named `FlowGeneratorGui` and `FlowGenerator.Parameters`, respectively. The GUI allows the user to set the magnitude of the flow, that otherwise is constant.

SinusFlowGenerator This class gives a flow with a sinus-shaped magnitude. A GUI class, named `SinusFlowGeneratorGui` allows for the user to set the mean value, the period and the amplitude of the flow. An inner parameter class is also provided.

ControlledFlowGenerator A `ControlledFlowGenerator` has a flow with a magnitude that is controlled by a controller, which should be a `SingleOutController`. This class also allows the flow to be set manually when the controller is turned off.

FlowGeneratorGui is used for this. An inner parameter class is also provided.

3.1.3 Samplers

State data is sampled from the ASP by aid of samplers. Two sampler classes are implemented in JASS 3.0, `jass3.simulation.DataSampler` and `ExcessSludgeController.ExcessSludgeSampler`. The latter is a subclass of the former, and an inner class of `ExcessSludgeController`. It has little or no use outside this class.

A `DataSampler` has three important parameters (fields): a sample period, a `DataSampler.SampleSource` and a `DataSampler.SampleTarget`. The sample period decides how often the `DataSampler` should take a sample. `DataSampler.SampleSource` and `DataSampler.SampleTarget` are inner interfaces of `DataSampler`, defined as shown below. A third inner interface is also shown, `DataSampler.MultipleSampleTarget`

```
/** Interface for classes that wish to give data to a
 * DataSampler. */
public interface SampleSource{
    public double getSample( int location, int variable);
}

/** Interface for classes that wish to recieve data from a
 * DataSampler. */
public interface SampleTarget{
    public void newSample( double value );
}

/** Interface for classes that wish to recieve data from a
 * subclass of DataSampler that gives more than one value.*/
public interface MultipleSampleTarget extends SampleTarget{
    public void newSample( double[] values );
}
```

`SampleSource` should be implemented by any class that wants to get sampled data from a `DataSampler` that samples one signal. `MultipleSampleSource` should be implemented by classes that should receive data from a subclass of `DataSampler` that samples several signals. `SampleSource` should be implemented by classes that are the source of sampled data. In JASS 3.0 `SampleTarget` is implemented by class `Controller` and therefor also by

its subclasses. `MultipleSampleTarget` is implemented by a single class: `ExcessSludgeController`. `SampleSource` is also implemented by a single class, `JassModel`.

As can be seen above, `SampleSource.getSample` takes two integer parameters to decide which value to sample: *location* and *variable*. The possible values for these parameters when getting samples from a `JassModel` representing a plant with n compartments are shown in Table 2 and 3

location	index
Influent water	0
Zone 1	1
...	...
Zone n	n
Settler bottom layer	$n + 1$
Settler top layer	$n + 2$

Table 2: Sample locations available in `JassModel`

Some explanations for Table 3:

- The index constants are defined constants that can be used for indexing. `State` refers to the class `jass3.simulation.iawqasm.State`.
- V_a is the total volume of the aerated compartments, and SS_m is the mean sludge concentration of the aerated compartments. For this sample variable, the location parameter does not matter.
- Q_{in} is the flow rate of influent water into a location. Flow of internally recycled water or sludge is not included. The settler is an exception, where Q_{in} is the flow from the last compartment.
- Q_{out} is the flow rate from one location to the next. If the location is the settler, Q_{out} denotes the effluent flow rate.
- Q_{pre} is the flow rate from the previous compartment. If the sample location is the first compartment, it is the flow of internally recirculated water. If the sample location is the settler, this flow rate is zero.
- $Q_{in,tot}$ is the total influent flow rate.
- All flow rates are in m^3/h .

variable	index	index constant
$X_{B,H}$	0	State.XBH
$X_{B,A}$	1	State.XBA
X_S	2	State.XS
X_I	3	State.XI
X_{ND}	4	State.XND
S_{NH}	5	State.SNH
S_{ND}	6	State.SND
S_{NO}	7	State.SNO
S_{ALK}	8	State.SALK
S_I	9	State.SI
S_O	10	State.SO
S_S	11	State.SS
$V_a \cdot SS_m$	12	JassModel.SAMPLE_SLUDGECONC_MULT_VOLUME
SS	13	JassModel.SAMPLE_SLUDGE_CONC
Q_{in}	14	JassModel.SAMPLE_QIN
Q_{out}	15	JassModel.SAMPLE_QOUT
Q_{pre}	16	JassModel.SAMPLE_QPRE
$Q_{in,tot}$	17	JassModel.SAMPLE_QIN_TOT

Table 3: Sample variables available in JassModel

A `DataSampler` keeps track of the time to the next sample. When its `step` method is called, the time to the next sample is decreased. If it reaches zero, a sample is taken from the `SampleSource` by calling its `getSample` method, and the value returned is passed to the `SampleTarget` by calling its `newSample` method.

3.1.4 IAWQCompartment

This class is the same as the class `Iaw` in JASS 2.0, with some minor modifications. The following description is taken from (Samuelsson 1998). Some minor changes to the text has been done when necessary due to changes in the implementation.

A proper description is that an instance of `IAWQCompartment` is simply a compartment with all its characteristics. The most important data fields needed to solve the differential equations of the IAWQ model are shown below.

```

double [] inflow;
double [] state;
double [] previous;
double [] regler;
double [] outflow;
double Qin;
double Qpre;
double Qreg;
double Qcar;
double Car;
double Air;
double V;

```

The array `inflow` contains the substance concentrations of the influent water, `state` contains the actual concentrations in a compartment, `previous` contains substance concentrations in the water coming from a previous zone, `regler` contains the concentrations in the excess sludge and `outflow` the concentrations in the water going out of the compartment. Similarly, `Qin` is the external inflow rate to the compartment, `Qpre` is the flow rate from the previous zone, `Qreg` is the flow rate of the water that is recycled from the settler and `Qcar` is the flow rate of external carbon source into a zone. Note that there is no datafield `Qout` since the outflow of the zone is calculated as the sum of the inflows in each time-step. The parameter `Car` is the COD of the water from the external carbon source (it is set to the constant value of `ETHANOL_KONC`), `Air` is the airflow into the zone and `V` is simply the volume of the zone. One thing here needs some further explanation: In the first zone the flow rate `Qpre` and concentration array `previous` is used when a predenitrifying system is simulated, that is when water is recirculated from the last zone to the first.

`IAWQCompartment` has five member functions, they will here be discussed in detail. To begin with the simplest one, the function `setNewConst(double[] newConst)`, is called when stoichiometric or biologic constants have been changed in the users interface. `setNewConst` also calls another member function in its own class, `CalcHelpConst()`, that calculates some help constant to make the solving of the differential equations a bit faster. The function `getState()` simply returns the current state in the compartment. The most important functions in the class are those for solving the differential equations, `newcalcIawqRungeKutta()` and `deltaF(double newstate[], double ystate[])`. The `newcalcIawqRungeKutta()` takes one time step (0.01h) in the differential equations for all the substances with the fourth order RungeKutta method. The code of the implemented Runge-

Kutta method given in (6) , see Appendix A.4, is shown below.

```
public void newcalcIawqRungeKutta(){
    int i = 0;
    if(upd.newProcConst.checkProcessUpdate()==true)
        setNewConst();

    deltaF(k1, state); //computes R-K parameters
    for(i=0; i<11; i++) {
        y_between[i] = state[i] + (__deltaT/2)*k1[i];
        if (y_between[i] < 0.0) y_between[i] = 0.0;
    }
    y_between[_SS] = state[_SS] + (__deltaT/2)*k1[_SS];
    if (y_between[_SS] < 0.0) y_between[_SS] = 0.0;

    deltaF(k2, y_between);
    for(i=0; i<11; i++) {
        y_between[i] = state[i] + (__deltaT/2)*k2[i];
        if (y_between[i] < 0.0) y_between[i] = 0.0;
    }
    y_between[_SS] = state[_SS] + (__deltaT/2)*k2[_SS];
    if (y_between[_SS] < 0.0) y_between[_SS] = 0.0;

    deltaF(k3, y_between);
    for(i=0; i<11; i++) {
        y_between[i] = state[i] + __deltaT*k3[i];
        if (y_between[i] < 0.0) y_between[i] = 0.0;
    }
    y_between[_SS] = state[_SS] + (__deltaT/2)*k3[_SS];
    if (y_between[_SS] < 0.0) y_between[_SS] = 0.0;

    deltaF(k4, y_between);

    for(i=0; i<11; i++) { //computes new states
        state[i] = state[i] + (__deltaT/3)*(0.5*k1[i] + k2[i]
            + k3[i] + 0.5*k4[i]);
        if (state[i] < 0.0) state[i] = 0.0;
    }
}
```

The most significant things in the code are the calls to the last member function of the class, `deltaF(double newstate[], double ystate[])`. This function is in principle equivalent to the $f_i(t, y_1, \dots, y_N)$ function described in the algorithm, i.e it computes the process rates (mass balances as well as biological and stoichiometric process rates). The function is called with the parameter `ystate[]`) which answers to y_1, \dots, y_N in the algorithm description. The values of the Runge-Kutta coefficients are stored in the array `newstate[]`.

3.1.5 IAWQSettler

This class is the same as the class `IawSetta` in JASS 2.0, written by Mats Ekman, with some minor modifications. It simulates settling using the model described in Appendix A.5, a ten-layer model with the double-exponential method for settling velocity. It resembles `IAWQCompartment` and the two classes have some fields in common: `Qin`, `Qreg`, `inflow`, `outflow` and `state`. It is important to note that in `IAWQSettler`, `Qreg` is not an inflow, but rather used for holding the outflow of sludge from the settler bottom layer. Also, `state` holds the state in the settler bottom layer, while the state in the top layer is stored in `stateout`.

3.1.6 JassModel

The class `JassModel` is probably the most important class in the simulator part of JASS 3.0. It represents a more or less arbitrary treatment plant that can be used for simulating, and uses all the above mentioned classes to hold information of the plant.

Three hash tables are used for storing the controllers, flowgenerators and samplers. Controllers and flowgenerators may be hashed by their name or by some arbitrary string, depending on what methods are called for adding them to the model. Samplers are hashed by some arbitrary string. Note that normally there is no need to access entries in these hash tables by name. This is only needed when removing a controller, flowgenerator or sampler.

There are several ways to get a `JassModel` that represents a specific plant layout. The easiest way is to instantiate a subclass of `JassModel` that represents that specific layout. Two such subclasses are available in JASS 3.0. They are located in the package `jass3.simulation.models` and are named `BenchmarkModel` and `OldJassModel`. The class `BenchmarkModel` is meant to represent a plant setup as specified in (Copp 2002), a five compartment predenitrifying plant (in the current implementation it does not completely

match the specified setup). `OldJassModel` is a little bit more general, and can have any number of compartments and can be pre- or post-denitrifying by default. Note that a plant can easily be changed from predenitrifying to post-denitrifying during simulation by changing the air flow rates. These classes comes by default with all controllers, flowgenerators and samplers necessary. Another way to get a `JassModel` representing a specific plant layout is to instantiate `JassModel` directly with the right number of compartments. This gives a basic model with no controllers, flowgenerators or samplers. This `JassModel` can then be changed to reflect the desired plant setup. This is basically what is done in the two subclasses mentioned before. A look in the source-code for `OldJassModel` might give a good idea on how it is done. It is also possible to instantiate any subclass of `JassModel` or loading a serialized one, and then remove or add controllers, flowgenerators and samplers as desired.

`JassModel` also holds an inner class called `JassModel.Update`. This class is used for sending updates to the model. How this is done will be described in more detail in section 3.4. The updates are stored in a vector appropriately named `updates`.

During simulation, the class has two important tasks: to simulate a time step, and to handle changes to the plant model due to user input. To simulate a time step, the method `SimulateStep` is used. The code is shown below.

```

/** Simulates a given step in time.
 * @param double timeStep the step in time
 */
public void simulateStep( double timeStep ){
    /* Updates the sampler and lets them sample. */
    for( Enumeration e = samplers.elements();
        e.hasMoreElements(); )
        ((DataSampler) e.nextElement() ).step( timeStep );
    /* Updates the model */
    if( !updates.isEmpty() )
        updateModel();
    calculateFlows();
    /* Calculates the new states of the compartments
     * and the settler */
    for( int i = 0; i < numberOfCompartments - 1; i++ ){
        System.arraycopy( compartments[i].state, 0,
                        compartments[i].outflow, 0, NSV );
        compartments[i].newcalcIawqRungeKutta();
        System.arraycopy( compartments[i].outflow, 0,

```

```

        compartments[i+1].previous,
        0, NSV );
    }
    System.arraycopy( compartments[numberOfCompartments-1].state,
        0,
        compartments[numberOfCompartments-1].outflow,
        0, NSV );
    compartments[numberOfCompartments-1].newcalcIawqRungeKutta();
    System.arraycopy( compartments[numberOfCompartments-1].outflow,
        0, settler.inflow, 0, NSV );
    settler.newcalcRungeKutta();
    /* increases the simulation time */
    simulationTime += timeStep;
}

```

What is done in this method is the following:

1. The program goes through all samplers and makes them count down the time to the next sampling instant, and samples by calling the `step` method if the time to the next sampling is less than zero.
2. The `updates` vector is checked for updates to the model. If the vector is non-empty, the updates are executed by calling the `updateModel` method that executes all updates in the `updates` vector.
3. The method `calculateFlows` is called. This method goes through all `FlowGenerators` and sets the magnitude and state of flows in the appropriate compartments and the settler.
4. For all compartments, the `state` is copied to `outflow` and `newcalcIawqRungeKutta` is called.
5. The `outflow` state of each compartment is copied to the `previous` field of the next compartment, except in the case of the last compartment, where it is copied to the settler.
6. The settler's `newcalcRungeKutta` is called.
7. The simulation time counter is increased.

3.1.7 The Simulator class

This class runs the simulation thread, and therefore implements the `Runnable` interface. Basically it runs a loop in which it calls the `simulateStep` method of the used `JassModel`. The time step used for solving the differential equations is defined in this class to be 0.01h. This class also controls the speed of the simulation by calling `System.currentTimeMillis` at regular intervals. If the simulation goes faster than desired, it makes small pauses by calling `Thread.sleep`. The default simulation speed is four simulation hours per real time second. This can be changed by editing the value for the field `simulationMinutesPerSecond` in the source code.

The simulation thread always runs, even when simulation is stopped. In the latter case, it sleeps in 100 milliseconds intervals until the simulation is started. Updates from the user to the model is still executed even when simulation is stopped. Two boolean fields are used to control the simulation: `runSimulation` and `simulating`. The field `runSimulation` is meant to be modified from outside the simulation thread and indicates if simulation should be started or stopped. That it is meant to be modified from outside the simulation *thread* does not mean that it is accessible outside the class. Actually it has private access, and is modified by calling the `startSimulation` and `stopSimulation` methods. The field `simulating` is set inside the simulation thread to indicate if simulation is running. It indicates when simulation has actually started or stopped after `runSimulation` has been set.

This class also provides methods for saving models. Saving has to be synchronized so that the model is not modified while saving. The field `lockModel` is used for this. If it is set to true, no changes to the model are allowed. Before a save the method `requestSimulationSave` should be called. It sets `lockModel` to true, suppressing any updates to the model. Then it sets `runSimulation` to false and waits until simulation has ceased. Then it returns, and the caller can now save the model. When the model is saved, `saveDone` should be called to unlock the model before simulation is started again. Methods for saving and loading models are implemented in the class `ModelFileHandler`.

3.1.8 Loading and saving models

JASS 3.0 contains functionality for saving and loading plant setups, as contained in the class `JassModel`, to files. The java *serialization* mechanism is used for this. Java serialization provides an easy way to save an object to file, more on this can be found in (Skansholm 1987). The only restriction on the object to be saved is that it must be an instance of a class that implements

the *serializable* interface. This interface is empty, so no method definition have to be added. `JassModel` implements this interface, as does all other classes that are used by it. Therefore the code for saving and loading models is very simple. It is implemented in the class `ModelFileHandler`. Another class called `ModelArchive` serves as a front end for loading models that may be located at different places. It holds a list of model names, and each model name is associated with a location where an instance of `JassModel` (or a subclass of it) representing the model may be found. The location may be one of the following:

- A file containing a serialized `JassModel`.
- A jar-file containing one or more serialized `JassModel`. The jar-file should not contain a manifest file.
- An URL that points to a jar-file containing one or more serialized `JassModel`.
- A `Class` object that represents a subclass of `JassModel`. In this case, a new instance of this class will be created when the model is loaded from the `ModelArchive`.

`SimulationController` holds a `ModelArchive` which entries are showed in the “load” menu, and which is used for loading those models.

3.1.9 Other classes

`jass3.simulation.ModelGenerator` Contains some methods and initial values useful for initiating/creating `JassModels`. The initial values are initial states for pre- and post-denitrifying models. Three methods for getting `SelectionSets` suited for use with a DO setpoint controller and `CarbonController` are provided, as well as a main method that creates some models and saves them to files. This method can be used to get some models for JASS 3.0.

`jass3.simulation.controller.ControllerVector` This class is used for adding controllers to a `JassModel` when the controllers GUIs are to be displayed in one dialog. More on this can be found in Section 3.2.11. A `ControllerVector` is basically a `Vector` with an internationalized name (internationalization is described in Section 3.2.10).

`jass3.simulation.iawqasm.Flow` This class holds a flow that has a magnitude and a state according to IAWQ ASM1. The state is represented by an instance of `State`.

`jass3.simulation.iawqasm.State` This class represents a state regarding the concentration of the components defined by IAWQ ASM1. It also contains some methods for calculating variables derived from the state, such as COD and BOD.

`jass3.simulation.iawqasm.ProcessParameters` Holds process parameters, as defined in IAWQ ASM1. It also holds an instance of `SettlerParameters` with parameters used for simulating the settler.

`jass3.simulation.iawqasm.SettlerParameters` Holds parameters used for simulating the settler with the double-exponential model.

3.2 The graphical user interface

In this section the most important classes that are part of the GUI are described. Most of the classes are located in the package `jass3.gui` and its subpackages. The exceptions are the GUIs of controllers and flowgenerators. See sections 3.1.1 and 3.1.2 for a little more about this. The reason for this will be discussed later.

The GUI of JASS 3.0 can be divided into two parts: a static part, that is always present, and a model dependent part, that depends on the current `JassModel` used. The latter consists of the dialogs for flowgenerators and controllers, while the former consists of pretty much everything else. This approach gives a flexible GUI, since it is easy to add new controllers. As long as they provide a GUI class, as described in section 3.1.1, their GUIs will automatically be included in the GUI of the program. The same goes for flowgenerators.

In the following subsections, most of the classes in the GUI are described, starting with those used in the main GUI window, which is represented by the class `GuiMain`.

3.2.1 The class `BarDiagram`

The class `BarDiagram` displays a bar diagram that may have any number of bars and a user defined space between each bar. The implementation is fairly simple. `BarDiagram` inherits `javax.swing.JPanel` and overrides its `paintComponent` methods to implement custom painting.

`java.awt.geom.Rectangle2D` is used for painting the bars. Each bar represents a value, and the values of the bars is set with the `setBarData` method. The bars are automatically scaled to fit into the panel. The bars are not fitted to fill the given area, but rather when the value of one bar is too big, the maximum value is set to that value multiplied by 1.33. If the maximum value of a bar is smaller than half the max value, the max value is also set to the largest value multiplied by 1.33. The value of each bar is also displayed above the bar. It is also possible to set a tool tip for each bar that is displayed when the mouse pointer is held above a bar.

3.2.2 The class `GGPlot`

This class displays a plot. The plot may have any number of data series. The plot may have two scales, one to the left and one to the right. Each data-series is scaled according to one of these. The line representing each data series may have its own color, and the scales are colored according to these. The y values of the plot may be scaled automatically in two different ways: either it is scaled so that the largest displayable value is the largest value multiplied with a constant (1.1 by default) or the plot is rescaled when the largest value is bigger than the largest displayable value. In the latter case, the plot is also rescaled when the largest value is smaller than half of the largest displayable value. The x values may be automatically scaled to fit exactly. The class also has methods for moving all data points, useful when used for real-time plots. A number of classes is used to build the plot, they are described below.

The class `PlotArea` This class displays the actual plot, excluding scales. It also may display a grid that corresponds to the ticks of the scales.

The class `PlotGrid` This class represents the grid that may be shown in the plot.

The class `PlotLine` This class represents the line in the plot that corresponds to a data series.

The class `PlotScale` This class represents a scale for a plot. It may be placed left, right, top or bottom to the plot area. It holds a reference to the plot area to calculate the scale correctly.

3.2.3 The class `RTPlotPanel`

This panel holds a plot together with three `JComboBoxes`. The boxes allows the user to select which components to plot, and in which area. The plot has two data series and each of those have it own scale, one on the left and one on the right. This class has five important methods: `addData(double timeStep)`, `getPlot`, `addActionListener`, `setLocations` and `setSelectedLocation`. `addData` lets the user add data two the plot data series. Y values for each data series should be given together with a value that indicates the (simulation) time since the last points were added. This adds the points with the x-value zero and the x-value of all other points are decreased by the given time. `setLocation` sets the locations that should be selectable in the location combo box. `setSelectedLocation` sets which location should be selected. `getPlot` gets the plot, which is an instance of `GGPlot`. This can be useful for resetting the plot and such. `addActionListener` register an action listener that will receive action events when a new value is selected in any of the combo boxes. Depending on what combo box fired the action event, the event will have its action command set to one of the constants `LOCATIONCB_ACTION_COMMAND`, `LEFTCB_ACTION_COMMAND` or `RIGHTCB_ACTION_COMMAND`, which are defined in this class. A bean info class is also presents for this class to enable it to be used as a Java bean.

3.2.4 The class `BarDiagramPanel`

This panel holds three `BarDiagrams`, together with a `JComboBox` for each, for selecting which component to display. This class has a couple of methods for setting the appearance, these can be found in the javadoc. The most important method are `setDiagramData` which sets the data of one of the bar diagrams. Other methods worth mentioning are `addActionListener` and `setSelectedBar`. `addActionListener` adds an action listener that will receive action events when a new value is selected in any of the combo boxes. The combobox firing the event is identified in the event action command by three defined constants: `JCB1_ACTION_COMMAND`, `JCB2_ACTION_COMMAND` and `JCB3_ACTION_COMMAND`. `setSelectedBar` sets the index of the selected bar. The selected bar will be painted with a different color in all of the bar diagrams. A bean info class is present for this class.

3.2.5 The class `TreatmentResultPanel`

This panel is used for displaying treatment results and sludge properties. There are six treatment results: COD, BOD, total-N, S_{NH} , S_{NO} and TSS. Each of those have three labels, intended for displaying in-, out- and reduction

values. These values are set with the method `setTreatmentResults`. It takes two parameters, an int index and an array of strings. The index decides which result should be set. The string array should contain values for in, out and reduction. Constants for indexing are defined, see the javadoc. The method `setSludgeValues` sets sludge properties. It takes an index and a string. The index indicates which properties label should be set, and the string is the value for that label. A bean info class is present for this class.

3.2.6 The class `ProcessImageComponent`

This class displays an image of the setup of the process. To build up the image the classes `BasinComponent`, `SettlerComponent`, `BasinCollectionComponent`, `Valve` and `ProcessImageComponent.Pipe`. They are described briefly below.

The class `BasinComponent` This class displays a treatment basin. The basin may be aerated, in which case an air valve and some bubbles will be displayed. Aeration is set with the `setAir` method. This class inherits `JToggleButton` and its status is shown by changing the border when it is selected.

The class `BasinComponentCollection` This class displays a number of treatment basins. It holds an array of type `BasinComponent []`. This class has only two methods: `setAir` that lets the user set the aeration in one of the basins, and `getButtons` that returns an array containing all the basins. The return type of the latter method is `JToggleButton []`.

The class `SettlerComponent` This class displays a view of the settler. The sludge concentration in each of the ten layers is displayed by different colors, where darker color means higher sludge concentration. A number of polygons is used for painting the settlers layers and edges. These are initialized by the private method `resetPolygons` which is called every time the component is resized. `SettlerComponent` also has some of the functionality of a `JToggleButton`, it can be clicked to be selected or deselected. This is implemented by having a non-visible `JToggleButton` in the field `secretButton`. This button is “clicked” when the user clicks inside the settler by a call to its `doClick` method. If the button is selected the settler gets a black border. The reason for this button functionality is to enable the user to select a basin or a settler.

The class Valve This class displays a valve that looks like a circle with a triangle in it. The valve can be closed or opened, and its color and the orientation of the triangle depends on this. Action listeners can be registered to this class and will then receive action events when the valve is clicked.

The class ProcessImageComponent.Pipe This class simply represents a line that may have one or more turns. It is used to draw pipes in the process image. What is a little bit special is that the coordinates are from 0 to 100, to conform with the `PercentLayout`, so the real coordinates have to be calculated before the pipe is drawn.

`ProcessImageComponent` holds one `BasinCollectionComponent`, one `SettlerComponent` and most of the time several `Pipes`. `PercentLayout` is used for the layout. The most important method of `ProcessImageComponent` is `resetFromModel` which resets it to reflect the process setup, as defined in a `JassModel`. This includes not only displaying the right number of basins, and a settler, which are the easy part, but also displaying pipes for input, and possibly external carbon, internal recirculation, return sludge and excess sludge as well. This is done by going through all `FlowGenerators` of the model and adding a pipe reflecting its flow, together with a `Valve` and a label for displaying the flow.

3.2.7 The class GuiMain

The class `GuiMain` is the main GUI window. It initializes the GUI and is responsible for calling the correct method in `SimulationController` when a GUI event occurs. When the GUI is initialized, dialogs for volumes, process parameters and influent components are created. This is done by instantiating anonymous subclasses of the classes for these dialogs. In the subclasses, the `doApply` method is overridden to send the correct command to the simulation controller when “apply” or “OK” is pressed.

3.2.8 The Dialog classes

A number of classes representing different types of dialogs are used in JASS 3.0. They are described below.

The class JassDialog This abstract class represents a dialog with three buttons : “OK”, “apply” and “cancel”. It is used as a super class of other dialogs. The functionality of the button is partly implemented: “cancel” hides the dialog, “apply” calls an abstract method called `doApply` and “OK”

calls `doApply` and then hides the dialog. Subclasses should override the `doApply` method and implement the desired functionality there. This class also holds an empty panel where subclasses should add what they wish to display.

The class `ControllerDialog` This dialog is used for displaying the GUI of controllers. When the “apply” button is pressed it sends a command to the simulation controller to update the controller. It also updates the controller GUI when the dialog is shown.

The class `MultipleControllerDialog` This dialog displays multiple controller GUI’s in a tabbed pane. It uses the class `ControllerPanel` for each panel.

The class `FlowDialog` This dialog is used for displaying the GUI of a flow generator. It functions similar to `ControllerDialog`.

The class `IncomingDialog` This dialog is used to set the concentration of the components in the incoming water. It has a abstract `doApply` method, which is overridden in an anonymous class created in `GuiMain`.

The class `ProcessParametersDialog` This dialog is used for setting the parameters used for simulating the ASP and the settling. Like `IncomingDialog` it has an abstract `doApply` method and needs to be subclassed.

The class `VolumesDialog` This dialog is used to set the volumes of the compartments, and the area and height of the settler. The `doApply` method is abstract.

3.2.9 Other classes

Here some minor classes will be described. The class `jass3.gui.components.DecimalField` represents a text field that takes only numerical values as input. It can also have max and min values for the input. It is used for all numerical input fields in the GUI. It used the classes `FormattedDocument` and `DecimalFormattedDocument` located in the package `jass.utilities`. The class `jass3.utilities.SignificantDigitsFormat` formats numbers to strings displaying a fixed number of significant digits. It is used for formatting of number in the GUI. The class

`jass3.gui.layout.PercentLayout` represents a layout manager that lets each component take up a fixed fraction of the space available.

3.2.10 Multiple language support

The GUI of JASS 3.0 has been designed to support multiple languages. For this the Java *internationalization* mechanism, with the standard Java class `ResourceBundle` has been used. It enables text strings to be loaded at runtime from a *resource file*. To enable multiple languages, one resource file for each language should be present. The language to use is set in a Java `Locale` object that is passed to the resource bundle. In JASS 3.0, the current locale is stored in the field `GuiConstants.locale`.

This works fine for all text in the GUI that is constant. However, some text depends on the current `JassModel` used. For example, in one menu the names of all controllers are displayed. Of course the names of the controllers could be set to the current language at creation time, but a problem then arises when loading a saved (serialized) model (see section 3.1.8). The name of the controller will then be the same as when the controller was created, and if the language now used is not the same, the GUI will now show the controllers name in the wrong language. To deal with this problem, a controller can have not only a name, but also a the name of a resource file that holds the name. When a serialized controller is deserialized, it will look up its name for the current language in the resource file. In this way, a serialized model can be used in another language than it was created in, as long as each controller has a resource file that contains its name in the current language. The class `ControllerVector` also has this functionality, as the name of a `ControllerVector` also may appear in the GUI.

3.2.11 Initialization of the GUI in class `SimulationController`

`SimulationController` is responsible for some of the work with setting up the GUI to reflect the current model used. It conducts the following tasks:

- Creates the dialogs for controllers, and the controller menu. This is done by going through the all entries in the current `JassModels` controller hash-table, and creating a dialog for each. If the entry is a controller, a `ControllerDialog` is created, containing the controllers GUI. If the entry is a `ControllerVector`, a `MultipleControllerDialog` is created instead. Also, for each dialog, an entry is added to the pop-up-menu that is displayed when the user presses the “Controllers” button in the main GUI window. All this is done in the method `initControllerDialogs`.

- Creates the dialogs for flow generators. This is done by going through all entries in the current `JassModels` `FlowGenerator` hash-table and creating a `FlowDialog` for each flow generator that is not of type `AIR` (flow generators for air does not have dialogs attached to them). For each a `FlowDialog` is created. Also, an `ActionListener` is added to the corresponding `Valve` that shows the dialog when the valve is clicked. Also, if the flow generator is an `ControlledFlowGenerator`, some code is added that disables the valve when the flow generator is automatically controlled.
- Creates the menu for loading models. This is done by getting all model names from the `ModelArchive` and adding them to the load model pop-up-menu. An `ActionListener` is added to each menu item that sends a command to load the model. If the program is run as an application, as opposed to as an applet, a “load from file” menu item is also added that opens a dialog that lets the user choose a file containing a serialized model.

The first two tasks are also performed every time the user changes the model.

3.3 Multi Threading

There are three active threads in Jass 3.0: the GUI thread, the simulation-control thread and the simulation thread. The GUI thread takes care of GUI events. The simulation-control thread runs mainly in the `SimulationController` class. It processes input from the GUI. The simulation thread executes the simulation code. Figure 5 shows an overview on in which parts of the program the different threads spend most of their active time.

Multithreading requires careful programming so that no more than one thread modifies the same data at once. If this is not done, strange results may appear. In Jass 3.0 the data that needs protection is the simulation data, basically the whole `JassModel` that is used. This is done by letting all changes to the `JassModel` be executed by the simulation thread.

3.4 Communication between the GUI and the simulator

Figure 5 shows how the different parts of the program interact. This is described in more detail below.

3.4.1 The simulation controller

Most communication between the gui and the simulator goes through the class `SimulationController`. The class `SimulationControllerCommand` is used for telling the simulation controller what to do. The vector `commands` is used by `SimulationController` as a queue for commands. This is used as a communication channel between the gui thread and the simulationcontroller thread. Commands can be added directly to the vector by calling the `addCommand` method. Also holds `SimulationController` holds some methods that adds specific commands, for example `simulationStart` and `SimulationStop` that adds commands for starting and stopping the simulation, respectively. When an event occurs in the gui that affects the simulation a `SimulationControllerCommand` is placed in the vector. This arrangement removes some work from the gui thread, which is beneficial since the update of the gui may give it a quite high workload. The `simctrl` thread can then perform the actions needed to respond to the actual gui event. Since there is no need for these actions to be performed very quickly, the `simctrl` thread runs a loop every tenth of a second that takes care of all waiting commands, and then sleeps for a tenth of a second.

3.4.2 The JassModel update queue

If changes were done to the `JassModel` at any time during simulation, there would be a possibility that the changes were done at an inappropriate time,

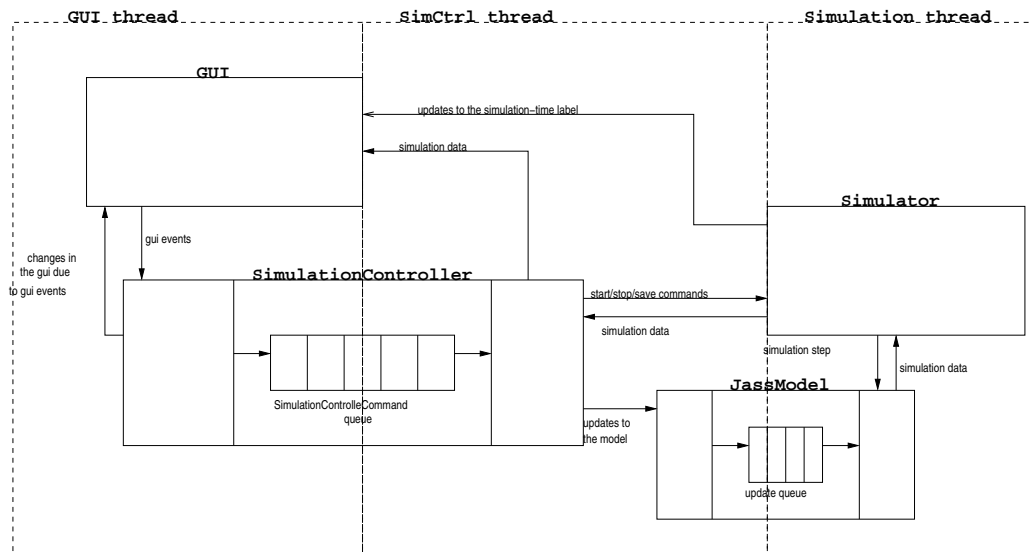


Figure 5: Communication between the gui and the simulator

which could give strange results. Therefore all changes to the model, for example those resulting from GUI input, is stored in a queue in the vector `updates` in `JassModel`. The changes are stored as instances of the class `JassModel.Update`. This class holds a reference to the object that should be changed, and the data with which it should be changed. This arrangement lets the `JassModel` deal with the updates at the right time. This is done every time the `simulateStep` method is called. If the `updates` vector is not empty, all updates are executed. Since this method is called inside the simulation thread, this thread is the only one that changes the model.

3.5 Jass3 : The main class

The class that starts the program is called `Jass3`. It extends `JApplet` and also has a main method, so the program can be run either as an applet or as a stand alone application. This class is responsible for creating the GUI, the simulator and the simulation controller and also for setting them up for working together. It also searches for a java archive (jar) file containing the models for the “load” menu.

If run as an applet, it searches for a file called “models.jar” at the home URL. If no such file is found, only the default model, a ten-compartment predenitrifying one, can be used. It takes three applet parameters, “country”, “language” and “defaultmodel”. The two first decides which resource file is used, and thus the language used, see section 3.2.10. The last decides which model is loaded by default, it should be the name of a model in the “models.jar” file.

If run as an application, it searches for the “models.jar” in the working directory. If it is not found, a dialog is displayed where the user can choose a jar-file containing models. If no file is chosen, only the default model can be simulated. When run as an application, two command line arguments can be given: a country code and a language code, which decides the resource file to use and thus the language, see section 3.2.10.

4 Conclusions

The Netbeans IDE was found to be a powerful tool for Java GUI programming, but not without drawbacks. Large, complex GUI's can be a bit hard to work with, but this can be made easier by dividing it into parts and making the parts Java Beans. This approach was used a lot, most parts of the main GUI are Java Beans. Generation of Java beans is quite simple with NetBeans.

Some parts of the GUI was created with Netbeans and then modified by hand. This approach is not recommended, since the Java code generated by Netbeans often has low readability, and once the auto generated code is changed the GUI can not be edited with the Netbeans GUI tools without Netbeans removing all changes to the auto generated code. This approach may however sometimes be useful as a way of creating a "code skeleton" for a GUI that can then be modified.

Netbeans was however found to be a very good tool for learning Java GUI programming. By creating a GUI and looking at the code it is easy to learn the basics of how things should be done. Some knowledge of Java layout managers is probably necessary for efficient use of Netbeans, but Sun fortunately provides some good tutorials for this, see (Sun 2002).

Netbeans also contains some useful tools that can be used on code generated outside NetBeans, for example for internationalization (multiple language support).

Most of the work was done in Netbeans 3.2.1. This version contained quite a few bugs, some of which made the program crash. Especially cutting and pasting of Java bean GUI elements seemed to crash the program often. This has hopefully been improved in the new version 3.3.1, which was used a little in the end of the project.

It may be worth mentioning that much programming was also done outside Netbeans, using the XEmacs text editor and the Java development kit from Sun. The built in text editor of Netbeans did not match XEmacs in key bindings, which can be a little annoying for users accustomed to XEmacs. It is however possible to use XEmacs to work in Netbeans, but the module for this seems to be in alpha- or beta-stadium and was therefore not tested.

In the resulting program, the desired improvements were implemented together with some other that surfaced during the work. The main goal was to improve the GUI using Netbeans, but since this was the major part of the program, most of the program was rewritten, the exception being the classes `Iaw` and `IawSetta` which was reused with only minor modifications. The code doing the actual numerical simulation was thereby kept rather intact. Comments from participants on courses where the simulator was used has

been used to improve userfriendliness of the GUI and to make it look more like GUI's used on real wastewater treatment plants.

With version 3.0 of JASS, the flexibility of the program has increased in a number of ways:

- The class `ProcessImageComponent` makes it possible to display the layout of plants with different setups in the GUI.
- The introduction of the generic `Controller` class together with the ability of the GUI to dynamically display the GUIs for controllers has made it easy to simulate plants with different types of controller, and also to implement new types of controllers. Implementing a new type of controller and using it for simulating in JASS 3.0 requires no change at all to the existing program.
- Multiple language support has been added. Support for English and Swedish has been included, support for other languages can easily be added.
- The concept of encapsulating a complete plant setup, together with parameters for simulation, in a single class (`JassModel`) makes it possible to use the program to simulate many different plant setups. It has also made it easy to implement loading and saving of different plant setups with the use of Javas serialization mechanism.

A Models for the activated sludge process and the settler

A.1 The IWA activated sludge model no 1

The description of the IAW Activated Sludge Model No 1 (ASM1) below is taken from (Samuelsson 1998), with some small modifications.

One of the most common models for the activated sludge process is the ASM1, and this is the model used in the simulator described in section 4. The model describes the three processes described in the previous sections, i.e removal of organic matter, nitrification and denitrification. See also (Jeppsson 1996) and (Henze *et al.* 1987). An overview of the model and its components is presented below.

A.1.1 Organic matter

In the model, the organic matter (the total COD) is assumed to consist of three main components that is further divided into different subcomponents.

- Biologically degradable organic matter.
There are two different types of this material, readily degradable organic matter, S_S , and slowly degradable organic matter, X_S .
- Non biodegradable organic matter.
This substance is divided into two fractions, soluble inert organic matter, S_I , and particulate inert organic matter. The particulate inert organic matter is divided into two subcomponents, X_P that derives from biomass decay, and X_I that derives from the incoming water.
- Active biomass (microorganisms).
Two main kinds of microorganisms are taken into account in the model: Heterotrophic bacteria, $X_{B,H}$, which under aerobic conditions decompose the main part of the organic matter. These microorganisms may under anoxic conditions cause the desired denitrification process. There are also autotrophic bacteria, $X_{B,A}$ that under aerobic conditions convert ammonium into nitrate (nitrification).

The different reaction phases of the organic matter in the model are schematically described in Figure 6.

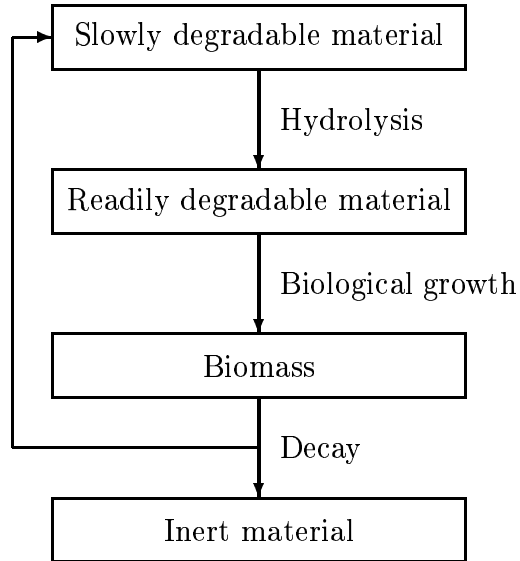


Figure 6: The biological decomposition chain.

A.1.2 Different fractions of nitrogen and other compounds

The nitrogen pollutions are divided into the following groups:

- Ammonium, S_{NH}
- Nitrite and nitrate, S_{NO}
- Soluble organic nitrogen, S_{ND}
- Particulate organic nitrogen, X_{ND}

There are two more components included in the model, soluble oxygen, S_O and the alkalinity S_{ALK} . The soluble oxygen is important in many of the biochemical reactions of the model, while the alkalinity does not affect the reactions, it is just a measure of the pH sensitivity of the water.

A.1.3 Biological and chemical reactions

The behavior of each of the substances described in the previous subsection is in the model described by nonlinear ordinary differential equations. These consist of two terms: An ordinary mass balance term and one reaction term.

A general formula for such a differential equation in one specific compartment is given below.

$$\frac{dZ}{dt} = \frac{Q}{V}(Z_{in} - Z) + R \quad (5)$$

Here, Z is the concentration of a certain substance in the zone, Z_{in} is the concentration of the substance in the water that flows into the zone, Q is the flow through a zone, V is the volume of the zone and R is the biochemical reaction rate for the specific substance in the zone. There is a number of biochemical reactions (R) that are taken into account in the model, and an overview is given below. In the model, these reactions is normally presented in matrix format. This process matrix is found in Appendix A.2.

- Aerobic growth of heterotrophs, $X_{B,H}$.
This process degrades organic matter, and will take place under aerobic conditions if enough organic matter S_S is present. See row 1 in the process matrix.
- Anoxic growth of heterotrophs, $X_{B,H}$.
This is the denitrification process, which will take place when the oxygen concentration is low and sufficient amounts of readily degradable organic matter, S_S , and nitrate S_{NO} are present. See row 2 in the process matrix.
- Aerobic growth of autotrophs, $X_{B,A}$.
This is the nitrification process and takes place if there is enough oxygen, S_O , and ammonium, S_{NH} , in the system. The result is that ammonium is turned into nitrate. See row 3 in the process matrix.
- Decay of heterotrophs.
Hereby, heterotrophs are turned into slowly degradable organic matter, X_S and inert organic matter X_P . Also, some particulate organic nitrogen, X_{ND} is created. See row 4 in the process matrix.
- Decay of autotrophs.
See above and row 5 in the process matrix.
- Ammonification of soluble organic nitrogen, S_{ND} . See row 6 in the process matrix
- Hydrolysis of entrapped organics.
This process describes the transfer of slowly degradable organic matter to readily degradable organic matter. See Figure 6 and row 7 in the process matrix.

- Hydrolysis of particulate organic nitrogen, X_{ND} to soluble organic nitrogen S_{ND} . See row 8 in the process matrix.

Below is a overview of how the different components of the model are affected by the reactions above. The differential equation for each substance is also presented. Note that the mass balance term of equation (1) has been excluded, the differential equations only present the reaction part, R . A short explanation of the different constants (or rather parameters) in the equations and their typical values are found in Appendix A.3. Some of these parameters show very little variation and can therefore be considered as constants while other are strongly dependent of temperature and other environmental factors.

- Readily degradable organic matter, S_S , is consumed by aerobic and anoxic growth of heterotrophs and generated by hydrolysis. The differential equation for the reaction part is presented below.

$$\begin{aligned} \frac{dS_S}{dt} = & -\frac{1}{Y_H} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{S_O}{K_{O,H} + S_O} \right) X_{B,H} \\ & - \frac{1}{Y_H} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \eta_g X_{B,H} \\ & + k_h \frac{\frac{X_S}{X_{B,H}}}{K_X + \frac{X_S}{X_{B,H}}} \left(\left(\frac{S_O}{K_{O,H} + S_O} \right) + \eta_h \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \right) X_{B,H} \end{aligned}$$

- Slowly degradable organic matter X_S is generated by decay of microorganisms and consumed by hydrolysis. The reaction part of the differential equation is presented below.

$$\begin{aligned} \frac{dX_S}{dt} = & (1 - f_P)(b_H X_{B,H} + b_A X_{B,A}) \\ & - k_h \frac{\frac{X_S}{X_{B,H}}}{K_X + \frac{X_S}{X_{B,H}}} \left(\left(\frac{S_O}{K_{O,H} + S_O} \right) + \eta_h \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \right) X_{B,H} \end{aligned}$$

- Autotrophic microorganisms, $X_{B,A}$, grows if oxygen and ammonium are present. The $-b_A X_{B,A}$ term in the equation is caused by decay.

$$\frac{dX_{B,A}}{dt} = \hat{\mu}_A \left(\frac{S_{NH}}{K_{NH} + S_{NH}} \right) \left(\frac{S_O}{K_{O,A} + S_O} \right) X_{B,A} - b_A X_{B,A}$$

- Heterotrophs, $X_{B,H}$, grows under aerobic conditions if enough substrate, S_S , is available. Heterotrophs may also grow under anoxic conditions if there is enough nitrate to replace the oxygen and enough

substrate S_S is available. The $-b_H X_{B,H}$ term in the equation describes the decay.

$$\begin{aligned}\frac{dX_{B,H}}{dt} &= \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{S_O}{K_{O,H} + S_O} \right) X_{B,H} \\ &\quad + \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \eta_g X_{B,H} - b_H X_{B,H}\end{aligned}$$

- Particulate inert organic matter, X_P , is generated by decay of microorganisms. No further terms are present in the equation since this material is inert and does not react further.

$$\frac{dX_P}{dt} = f_P (b_H X_{B,H} + b_A X_{B,A})$$

- Dissolved oxygen, S_O , is consumed by aerobic growth of heterotrophs and autotrophs.

$$\begin{aligned}\frac{dS_O}{dt} &= -\frac{1 - Y_H}{Y_H} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{S_O}{K_{O,H} + S_O} \right) X_{B,H} \\ &\quad - \frac{4.57 - Y_A}{Y_A} \hat{\mu}_A \left(\frac{S_{NH}}{K_{NH} + S_{NH}} \right) \left(\frac{S_O}{K_{O,A} + S_O} \right) X_{B,A}\end{aligned}$$

- Nitrate S_{NO} is turned into nitrogen gas by denitrification (anoxic growth of heterotrophs). Nitrate is formed by aerobic growth of autotrophs (nitrification).

$$\begin{aligned}\frac{dS_{NO}}{dt} &= -\frac{1 - Y_H}{2.86 Y_H} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \eta_g X_{B,H} \\ &\quad + \frac{1}{Y_A} \hat{\mu}_A \left(\frac{S_{NH}}{K_{NH} + S_{NH}} \right) \left(\frac{S_O}{K_{O,A} + S_O} \right) X_{B,A}\end{aligned}$$

- Ammonium, S_{NH} , is turned into nitrate by aerobic growth of autotrophs (nitrification). Some of the ammonium is assimilated in the sludge. Ammonium is formed by ammonification of soluble organic nitrogen.

$$\begin{aligned}\frac{dS_{NH}}{dt} &= -i_{XB} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{S_O}{K_{O,H} + S_O} \right) X_{B,H} \\ &\quad - i_{XB} \hat{\mu}_H \left(\frac{S_S}{K_S + S_S} \right) \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \eta_g X_{B,H} \\ &\quad - \left(i_{XB} + \frac{1}{Y_A} \right) \hat{\mu}_A \left(\frac{S_{NH}}{K_{NH} + S_{NH}} \right) \left(\frac{S_O}{K_{O,A} + S_O} \right) X_{B,A} + k_a S_{ND} X_{B,H}\end{aligned}$$

- Soluble organic nitrogen, S_{ND} , is consumed by ammonification and generated by hydrolysis of particulate organic nitrogen.

$$\begin{aligned} \frac{dS_{ND}}{dt} = & -k_a S_{ND} X_{B,H} \\ & + \frac{X_{ND}}{X_S} k_h \frac{\frac{X_S}{X_{B,H}}}{K_X + \frac{X_S}{X_{B,H}}} \left(\left(\frac{S_O}{K_{O,H} + S_O} \right) + \eta_h \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \right) X_{B,H} \end{aligned}$$

- Particulate organic nitrogen, X_{ND} , is consumed by hydrolysis and generated by decay of microorganisms.

$$\begin{aligned} \frac{dX_{ND}}{dt} = & (i_{XB} - f_{PiXP})(b_H X_{B,H} + b_A X_{B,A}) \\ & - \frac{X_{ND}}{X_S} k_h \frac{\frac{X_S}{X_{B,H}}}{K_X + \frac{X_S}{X_{B,H}}} \left(\left(\frac{S_O}{K_{O,H} + S_O} \right) + \eta_h \left(\frac{K_{O,H}}{K_{O,H} + S_O} \right) \left(\frac{S_{NO}}{K_{NO} + S_{NO}} \right) \right) X_{B,H} \end{aligned}$$

The careful reader may here note that the soluble inert material, S_I , the alkalinity, S_{alk} , and the particulate inert organic matter that derives from the incoming water, X_I , is not included in the overview above. These components are considered not to affect the biological and chemical reactions above and are therefore excluded. These are modeled as pure mass balances.

A.2 ASM1 process matrix

From (Jeppsson 1996)

Component	i	j	Process	i	j	Process Rate, ρ_j [ML ⁻³ T ⁻¹]
Observed Conversion Rates [ML ⁻³ T ⁻¹]	1	2	1	1	1	$r_i = \sum_j V_j \rho_j$
	2	3	1	2	2	
	3	4	1	3	3	
	4	5	1	4	4	
	5	6	1	5	5	
	6	7	1	6	6	
	7	8	1	7	7	
	8	9	1	8	8	
Stoichiometric Parameters: Heterotrophic yield: Y_H Autotrophic yield: Y_A Fraction of biomass yielding particulate products: f_p Mass N/Mass COD in biomass: k_{np} Mass N/Mass COD in products from biomass: k_{xp}	Soluble inert organic matter [M(COD)L ⁻³]	1	1	1	1	$r_i = \sum_j V_j \rho_j$
	Readily biodegradable substrate [M(COD)L ⁻³]	2	1	2	2	
	Particulate inert organic matter [M(COD)L ⁻³]	3	1	3	3	
Kinetic Parameters: Heterotrophic growth and decay: μ_{hp} , K_S , K_{OH} , K_{NO} , b_H Autotrophic growth and decay: μ_{ap} , K_S , K_{OH} , K_{NO} , b_A Correction factor for anoxic growth of heterotrophs: η_h Ammonification: k_a Hydrolysis: k_h , K_X Correction factor for anoxic hydrolysis: η_b	Slowly biodegradable substrate [M(COD)L ⁻³]	4	1	4	4	$r_i = \sum_j V_j \rho_j$
	Active heterotrophic biomass [M(COD)L ⁻³]	5	1	5	5	
	Active autotrophic biomass [M(COD)L ⁻³]	6	1	6	6	
	Particulate products arising from biomass decay [M(COD)L ⁻³]	7	1	7	7	
	Oxygen (negative COD) [M(-COD)L ⁻³]	8	1	8	8	
	Nitrate and nitrite nitrogen [M(N)L ⁻³]	9	1	9	9	
	NH ₄ +NH ₃ nitrogen [M(N)L ⁻³]	10	1	10	10	
	Soluble biodegradable organic nitrogen [M(N)L ⁻³]	11	1	11	11	
	Particulate biodegradable organic nitrogen [M(N)L ⁻³]	12	1	12	12	
	Alkalinity – Molar units	13	1	13	13	
		14	1	14	14	
		15	1	15	15	
		16	1	16	16	
	17	1	17	17		
	18	1	18	18		
	19	1	19	19		
	20	1	20	20		
	21	1	21	21		
	22	1	22	22		
	23	1	23	23		
	24	1	24	24		
	25	1	25	25		
	26	1	26	26		
	27	1	27	27		
	28	1	28	28		
	29	1	29	29		
	30	1	30	30		
	31	1	31	31		
	32	1	32	32		
	33	1	33	33		
	34	1	34	34		
	35	1	35	35		
	36	1	36	36		
	37	1	37	37		
	38	1	38	38		
	39	1	39	39		
	40	1	40	40		
	41	1	41	41		
	42	1	42	42		
	43	1	43	43		
	44	1	44	44		
	45	1	45	45		
	46	1	46	46		
	47	1	47	47		
	48	1	48	48		
	49	1	49	49		
	50	1	50	50		
	51	1	51	51		
	52	1	52	52		
	53	1	53	53		
	54	1	54	54		
	55	1	55	55		
	56	1	56	56		
	57	1	57	57		
	58	1	58	58		
	59	1	59	59		
	60	1	60	60		
	61	1	61	61		
	62	1	62	62		
	63	1	63	63		
	64	1	64	64		
	65	1	65	65		
	66	1	66	66		
	67	1	67	67		
	68	1	68	68		
	69	1	69	69		
	70	1	70	70		
	71	1	71	71		
	72	1	72	72		
	73	1	73	73		
	74	1	74	74		
	75	1	75	75		
	76	1	76	76		
	77	1	77	77		
	78	1	78	78		
	79	1	79	79		
	80	1	80	80		
	81	1	81	81		
	82	1	82	82		
	83	1	83	83		
	84	1	84	84		
	85	1	85	85		
	86	1	86	86		
	87	1	87	87		
	88	1	88	88		
	89	1	89	89		
	90	1	90	90		
	91	1	91	91		
	92	1	92	92		
	93	1	93	93		
	94	1	94	94		
	95	1	95	95		
	96	1	96	96		
	97	1	97	97		
	98	1	98	98		
	99	1	99	99		
	100	1	100	100		

A.3 Process parameters in the simulator

Table 4 shows ASM1 process parameters and their typical values.

Symbol	Simulator	Value	Explanation
Y_A	Ya	0.24	yield for autotrophic biomass
Y_H	Yh	0.67	yield for heterotrophic biomass
i_{XB}	Ixb	$0.086 \text{ g N(g COD)}^{-1}$	mass of nitrogen per mass of COD in biomass
i_{XP}	Ixp	$0.06 \text{ g N(g COD)}^{-1}$	mass of nitrogen per mass of COD in products from biomass in endogenous mass
f_P	fp	0.08	fraction of biomass yielding particulate products
$\hat{\mu}_A$	mya	0.033 h^{-1}	maximum specific growth rate for autotrophic biomass
$\hat{\mu}_H$	myh	0.25 h^{-1}	maximum specific growth rate for heterotrophic biomass
K_S	Ks	20 g COD m^{-3}	half saturation coefficient for heterotrophic biomass
$K_{O,H}$	Koh	$0.2 \text{ g O}_2 \text{ m}^{-3}$	oxygen half saturation coefficient for heterotrophic biomass
K_{NO}	Kno	$0.5 \text{ g NO}_3\text{-N m}^{-3}$	nitrate half saturation coefficient for denitrifying heterotrophic biomass
K_{NH}	Knh	$1.0 \text{ g NH}_3\text{-N m}^{-3}$	ammonium half saturation coefficient for autotrophic biomass
$K_{O,A}$	Koa	$0.4 \text{ g O}_2 \text{ m}^{-3}$	oxygen half saturation coefficient for autotrophic biomass
b_A	Ba	0.00833 h^{-1}	decay rate coefficient for autotrophic biomass.
b_H	Bh	0.026 h^{-1}	decay rate coefficient for heterotrophic biomass
η_g	etag	0.8	correction factor for μ_H under anoxic conditions
η_h	etah	0.4	correction factor for hydrolysis under anoxic conditions
k_a	Ka	$0.0033 \text{ m}^3 \text{ COD (g h)}^{-1}$	ammonification rate
k_h	Kh	$0.125 \text{ g COD (g COD h)}^{-1}$	maximum specific hydrolysis rate
K_X	Kx	$0.03 \text{ g COD (g COD)}^{-1}$	half saturation coefficient for hydrolysis of slowly biodegradable substrate
$S_{O,sat}$		8.67 mg/l	saturated oxygen concentration

Table 4: The default parameter values for the implemented IWA model at 20°C

A.4 The modelling of one mixed zone

This section describes how one mixed zone is modelled. This is implemented in the class `IAWQCompartment`. The description is taken from (Samuelsson 1998).

Since the programming language Java is object oriented one mixed zone can be looked upon as a separate block with its own characteristics such as volume, constants and flows (see also (Luttmer 1995)). Figure 7 shows a schematic overview of a zone. Here Q_{in} is the inflow of wastewater to the zone, Q_{pre} is the flow from the previous zone, Q_{reg} is the flow recirculated from the settler, Q_{car} is the external carbon flow and Air is the air flow. Associated with each flow are of course concentrations of all the components described in the previous section. Note that all zones has these properties, for instance each zone has an inflow of external carbon but it is only possible to set this flow to something else than zero in one of the zones. It may be of

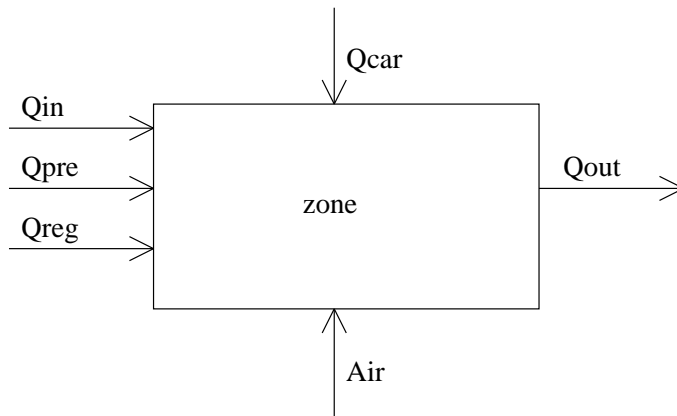


Figure 7: The flows into a zone.

interest that the external carbon source in the simulator is ethanol and has a COD of 1200000 mg/l.

The effect of the external air flow is modeled as $K_L a(u)(S_{O,sat} - S_O)$. Here u is the air flow rate, the function $K_L a(u)$ describes the transfer function of the external aeration system, and $S_{O,sat}$ is the oxygen concentration at saturation. This equation comes in as a positive term in the differential equation for oxygen presented in Appendix A.1.3. The oxygen transfer function $K_L a(u)$ is here modeled as $K_L a(u) = ku$ where k is a constant that describes the efficiency of the process. The simulator will later be extended

to allow for nonlinear and volume dependent $K_L a(u)$.

In the simulations the 12 differential equations of the form $\frac{dZ}{dt} = \frac{Q}{V}(Z_{in} - Z) + R$ described in Appendix A.1.3 are solved numerically for each zone with a fourth order Runge-Kutta method. The algorithm for this method is shown below.

$$\begin{aligned}
\frac{dy_i}{dt} &= f_i(t, y_1, \dots, y_N), i = 1 \dots N \\
k_1^i &= f_i(t_k, y_1^k, \dots, y_N^k) \\
k_2^i &= f_i(t_k + \frac{h}{2}, y_1^k + \frac{h}{2}k_1^i, \dots, y_N^k + \frac{h}{2}k_1^i) \\
k_3^i &= f_i(t_k + \frac{h}{2}, y_1^k + \frac{h}{2}k_1^i, \dots, y_N^k + \frac{h}{2}k_1^i) \\
k_4^i &= f_i(t_k + h, y_1^k + hk_3^i, \dots, y_N^k + hk_3^i) \\
y_i^{k+1} &= y_i^k + \frac{h}{3}(\frac{1}{2}k_1^i + k_2^i + k_3^i + \frac{1}{2}k_4^i) \\
t_{k+1} &= t_k + h
\end{aligned}$$

In our case h is the time step, y_i^k is the concentration of substance i at time step k and $k_1^i, k_2^i, k_3^i, k_4^i$ are the four Runge-Kutta extrapolation coefficients of the i :th substance. The function $f_i(t, y_1, \dots, y_N)$ is simply the process rate given in equation (1) in Appendix A.1.3, i.e the time derivative of the i :th substance concentration y_1 . The time step h is fixed at 0.01 h in the simulations.

A.5 Modelling of the settler

The settler is modelled using a one-dimensional model with equidistant layers and a constant cross-sectional area, as described in (Jeppsson 1996). In this case, a ten layer model is used, with feed in layer 4. The settling velocity v_s is calculated with the *double-exponential* settling velocity function defined as

$$v_s = \max\left(0, \min\left(v'_0, v_0\left(e^{-r_h(X-f_{ns}X_f)} - e^{-r_p(X-f_{ns}X_f)}\right)\right)\right) \quad (6)$$

where v'_0 , v_0 , r_h , r_p and f_{ns} are settling parameters. v'_0 and v_0 are the practical and theoretical settling velocity, respectively, r_h is a parameter characteristic of the hindered settling zone and r_p is a parameter associated with the settling behaviour at low solids concentration. X is the concentration of suspended solids, f_{ns} the non-settleable fraction of the suspended solids and X_f is the concentration of suspended solids in the influent water.

If the flow into the settler in the feed layer is denoted Q_f , the effluent flow of the top layer Q_e and the flow due to sludge outtake in the bottom layer Q_u , the equations for modelling the settler becomes

$$\frac{dX_i}{dt} = \begin{cases} \frac{J_{up,2} - J_{up,1} - J_{clar,1}}{z_1} & \text{if } i = 1 \\ \frac{J_{up,i+1} + J_{clar,i+1} - J_{up,i} - J_{clar,i}}{z_i} & \text{if } 1 < i < 4 \\ \frac{\frac{Q_f X_f}{A} + J_{clar,3} - (V_{up} - V_{dn})X_4 - \min(J_{s,4}, J_{s,4})}{z_4} & \text{if } i = 4 \\ \frac{\frac{dX_i}{dt} = \frac{v_{dn}(X_{i-1} - X_i) + \min(J_{s,i}, J_{s,i-1}) - \min(J_{s,i}, J_{s,i+1})}{z_i}}{z_i} & \text{if } 4 < i < 10 \\ \frac{v_{dn}(X_9 - X_{10}) - \min(J_{s,9}, J_{s,10})}{z_{10}} & \text{if } i = 10 \end{cases} \quad (7)$$

$$J_{up,i} = v_{up}X_i \quad (8)$$

$$v_{up} = \frac{Q_e}{A} \quad (9)$$

$$v_{dn} = \frac{Q_u}{A} \quad (10)$$

$$J_{clar,i} = \begin{cases} J_{s,i} & \text{if } X_{i+1} \leq X_t \\ \min(J_{s,i}, J_{s,i+1}) & \text{if } X_{i+1} > X_t \end{cases} \quad (11)$$

$$J_{s,i} = \begin{cases} \min(v_{s,i}X_i, v_{s,i+1}X_{i+1}) & \text{if } i \geq 4 \vee X_{i+1} > X_t \\ v_{s,i}X_i & \text{if } i < 4 \wedge X_{i+1} \leq X_t \end{cases} \quad (12)$$

where $v_{s,i}$ is the settling velocity in layer i and X_t is a treshhold concentration, in JASS 3.0 the value 3000 mg/l is used.

B Users manual

B.1 Introduction to JASS

B.1.1 ASP models used in JASS

JASS uses the IWA ASM1 to simulate the activated sludge process. When the word model is used in this manual, it does not refer to the model used for simulating the activated sludge process, but rather to a model of an activated sludge process setup, with compartment volumes, controllers etc. This is described in more detail in section B.2.1.

B.1.2 The GUI

In Figure 8 the GUI is shown. Below is a description on how the simulator is used.

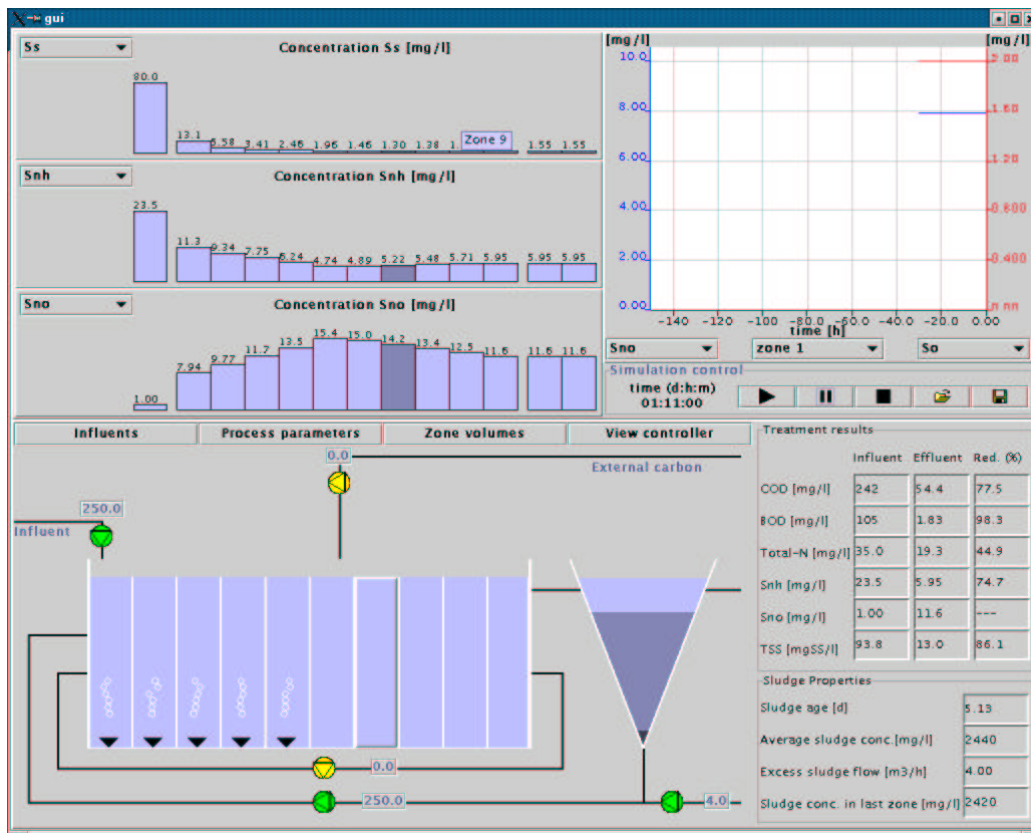


Figure 8: The Jass 3.0 GUI

1. In some parts of the GUI, Java tool-tips are enabled. This means that if the mouse pointer is held still over a part of the GUI, a small help text is displayed. This is enabled in the bar diagrams, and the dialogs for influent components and process parameters.

B.1.3 Running the simulator

2. The simulation is started by pressing the start button (▶).
3. The simulated time is displayed in the *Simulation control* panel in days, hours and minute.
4. If the stop button (■) is pressed the simulation is stopped and reseted. This means that the state of the process is now the same that it was before you started the simulation regarding the concentrations in the zones. This only affects the internal states of the compartments and the settler. Any changes made in the controllers, the influent water composition etc. are left unchanged
5. The pause button (⏸) can be used to stop the simulation temporarily, for instance to study a curve in the plot. If the start button is pressed when the simulation is paused, the simulation will continue from where it was.
6. The load button can be used to load different plant configurations.
7. With the save button the current plant configuration can be saved to a file. Note that this button is disabled when the simulator is run as an applet, since applets are not allowed to save files to disk.

B.1.4 Displaying simulation data

8. In the upper left of the GUI are three bar diagrams. Each of these display the concentration of a component in all zones as well as in influent, effluent and recirculation water. To choose what component to display in a bar diagram, click the combobox to the left of the bar diagram, and choose a component.
9. In the upper right of the GUI, there is a plot with three comboboxes under it. This plot displays the values from the last 150 hours of two components in a specific zone. To change which components to display, use the leftmost and rightmost comboboxes. To change the zone, use

the combobox in the middle. Note that the plot only stores the data that is currently being plot, so when you change the component or zone, the plot will start again at time zero.

10. In the lower right of the GUI some treatment results and some sludge properties are displayed. The treatments results are displayed as influent and effluent concentrations together with the reduction in percent.

B.1.5 Changing simulator properties

11. In the lower left of the GUI a schematic picture of the process is displayed. Here the compartments, the settler and the flow-paths (internal recirculation, return sludge etc) are displayed. Some flow-paths has a pump on them. If the pump is clicked, a dialog is displayed where the flow rate can be set. The color of a pump reflects the current flow rate: a green pump means the flow rate is non-zero, while a yellow pump represents a flow rate of zero. A pump may also be gray, which means that the flow rate is controlled by a controller and may not be set manually. If the user clicks in a compartment or in the settler, the corresponding bar will be highlighted in the bar diagrams.
12. The composition of the influent water can be set by pressing the button labeled “Influents”, which displays a dialog where the concentration of the different components can be set.
13. Parameters used in the ASM1 and parameters used in the settler model can be set in a dialog that is displayed when The button labeled “Process Parameters” is pressed. The parameters are defined by the models used for simulating the ASP and the settler, see Appendix A.
14. The volumes of the compartments, as well as the settlers area and height can be set in a dialog that is displayed when the button labeled “Volumes” is pressed.
15. The button labeled “Controllers” brings up a pop-up menu where you can select one of the controllers used by the process. A dialog is then shown where you can set the controllers parameters. The controllers are described in the next section.

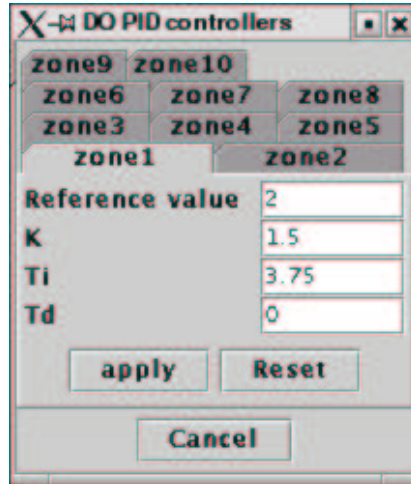


Figure 9: The DO PID controllers dialog

B.1.6 The controllers

16. The oxygen in each compartment is controlled by a PID controller. To change the parameters for one of these, press the “Controllers” button and select “DO PID controllers”. A dialog like the one shown in Figure 9 will be displayed. The dialog holds a tabbed panel where the number of tabs matches the number of compartment. The tabs are named “zone 1”, “zone 2” and so on. Clicking on one of the tabs displays the parameters for the DO controller of that zone. Here it is possible to change the reference value and the PID parameters. After Changes has been made, pressing “apply” will set the new parameters of the controller. If “reset” is pressed the changes will be undone. Pressing “Cancel” closes the dialog.
17. It is also possible to use a supervisory PID for the DO setpoint. By pressing the “Controllers” button and chose “DO setpoint controller” a dialog is displayed, shown in Figure 10. It displays the parameters of the DO setpoint PID. If the “enable as master controller” checkbox is checked, this controller will provide the reference values for the DO PID controllers base on the NH_4 concentration in the reference zone. The reference zone can be chosen in the dialog, as well as the reference value for NH_4 and the PID parameters. Note that no changes take effect until “OK” or “apply” is pressed. “OK” also closes the dialog.

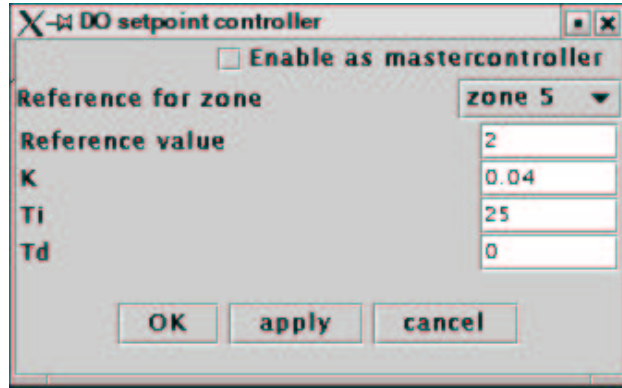


Figure 10: The DO setpoint controller dialog

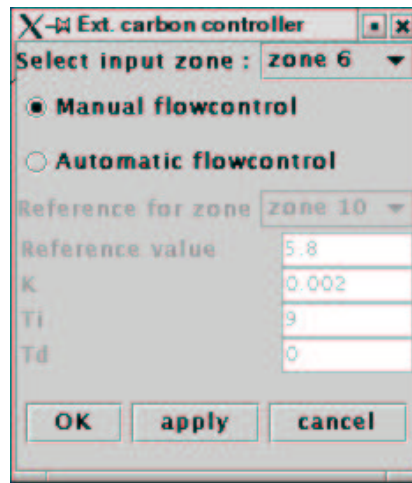


Figure 11: The Ext. carbon controller dialog

18. The carbon controller is a PID controller that controls the NO_3^- concentration in one zone by changing the flow of external carbon. By choosing “Ext. carbon controller” from the controllers menu the dialog shown in Figure 11 is displayed. In this dialog it is possible to set in which zone the carbon should be added, in which zone the NO_3^- concentration should be controlled, the reference NO_3^- value and the PID parameters. It is also possible to select manual mode, in which case the flow can be set manually by clicking on the carbon pump symbol.
19. The excess sludge controller keeps a fixed sludge age by controlling the excess sludge flow. It uses a mass balance equation to calculate the flow. By choosing “Excess sludge controller” in the controllers menu the dialog shown in Figure 12 is displayed. In this dialog settings for

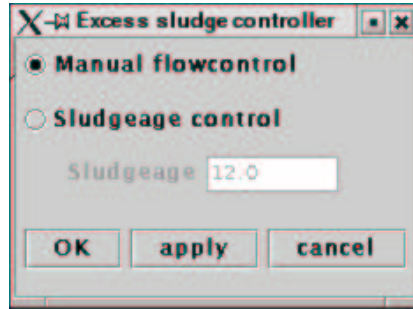


Figure 12: The Excess sludge controller dialog

the excess sludge controller is displayed. The controller can be set to manual mode, which allows for setting the excess sludge flow rate by clicking on the corresponding pump symbol. If the flow is set to automatic control, the desired sludge age should be given.

B.2 Working with JassModels

B.2.1 Introduction to JassModels

A JassModel is a model representing the activated sludge process setup, and is encapsulated in the Java class JassModel. This class is described in Section 3.1, and the reader is advised to read this section before continuing.

B.2.2 Creating new models

Modifying an existing model using JASS One method for creating a new model is to load an old model into JASS, modifying it and then saving it. This method has a limited use, since there are many parts of the process setup that can not be changed. These include the number of compartments, the types of controllers and more. What can be changed are the compartment volumes, the ASM1 parameters, the settling parameters and parameters for the controllers, and also the composition of the influent water. Also all flows like influent, internal recirculation etc can be changed.

Modifying an existing model using Java programming A user that has some experience of Java programming can use Java programming to modify an existing model. The existing model can be a serialized (saved) model or an Instance of class JassModel or one of its subclasses. The advantage of this method compared to the one described above is that it gives the user the possibility to add and remove controllers and flowgenerators, even new types of controllers and flowgenerators can be added to the model (see Section B.2.3 for more info on this). If you start with a saved model, it is not possible to change the number of compartments. To create a model with any number of compartments, you can either instantiate JassModel directly to create an empty model with the given number of compartments. The model is empty in the sense that it has no FlowGenerators and no controller, so it will be necessary to add those that are needed. A simpler way is to instantiate the class OldJassModel which creates a model with a given number of compartments, and also adds standard features such as return sludge flow, internal recirculation, aeration and such.

To modify a model in this way, the user writes a program that creates an instance of JassModel or one of its subclasses, either by instantiating an existing class or by loading a saved model. Then the desired changes to the model can be done, and then the model should be saved. Methods for saving and loading models are implemented in the java class ModelFileHandler. It is of course also possible to copy the source code from a class that inherits JassModel, and then modify and rename it to get a new type of model.

B.2.3 Implementing new controllers

Implementing a new type of controller should be fairly easy. Normally it is necessary to implement three classes:

1. A class representing the actual controller
2. A class representing the controllers GUI
3. A parameter class used for passing data between the controller and the GUI

The two last of these can be omitted, but only if no properties of the controller should be changed during simulation.

Implementing the actual controller The class implementing the controller should be a subclass of `Controller`. The most simple case is implementing a single-in single-out controller. In this case, it should be a subclass of `SISOController`. To make the controller functional, the abstract method `newInput(double input)` should be overridden. This method takes an input, most likely a sampled value of the variable to be controlled, and should then calculate a new out signal that should be stored in the double field `u`. The out signal can then be accessed by calling the `getValue()` method. For example, say that one wishes to implement a P controller. The class could be called `PController` and the Java code would look like this:

```
import jass3.simulation.controller.*;
public class PController extends SISOController
    implements java.io.Serializable{
    /* k is the K parameter of the P controler */
    private double k;

    public PController(){
        k = 2;
    }

    public void newInput( double input ){
        double e = yref - input;
        u = k * e;
    }
}
```

where `yref` is a double field defined in `SingleOutController` that holds the reference value. We now have a functional controller: when `newInput` is called it calculates the control error and then a new out signal, that we can read with `getValue()`. The implements `java.io.Serializable` part makes it possible to save the controller to a file. This is necessary to be able to save a `JassModel` using the controller.

Also if the controller has a GUI, the methods `hasGui()` and `getGui()` from class `Controller` should be overridden. The first one should return true, and the other one should return an instance of the GUI class.

Implementing a parameter class JASS 3.0 uses special parameter classes for passing data between a controller and its GUI. Such a class is needed for every controller that has a GUI. The parameter class should be a subclass of `Controller.Parameters`. Note that this is an inner class of `Controller`. Making the parameter class an inner class is a good idea. The parameter class should hold data representing the properties of the controller. For example, if a parameter class were written for `PController` it would hold information about the K value and the reference value. If it were implemented as an inner class the code for `PController` would look like:

```
import jass3.simulation.controller.*;
public class PController extends SISOController
    implements java.io.Serializable{
    /* k is the K parameter of the P controller */
    private double k;

    public static class Parameters extend SingleOutController.Parameters{
        public double k;

        public Parameters( double ref, double k ){
            super( ref );
            this.k = k;
        }
    }

    public PController(){
        k = 2;
    }

    public void newInput( double input ){
        double e = yref - input;
```

```

        u = k * e;
    }
}

```

The parameter class here inherits `SingleOutController.Parameters`. It could also have inherited `Controller.Parameters`, but then a field holding the reference value would have to be added, since the `ref` field would not have been inherited.

Implementing the controllers GUI The GUI should be a subclass of `ControllerGui`. This class has two abstract methods that will have to be implemented: `getData()` and `setParameters(Controller.Parameters parameters)`. `getData()` should return the input data from the GUI, encapsulated in the parameter class of the controller. `setParameters` should set the GUI element to reflect the current values of the controllers properties. For example, a GUI to the class `PController` could be a class called `PControllerGui`. It could have two `jass3.gui.components.DecimalField` for the reference value and the K value respectively, called `refField` and `kField`. The implementation of the above mentioned methods would then be:

```

public Controller.Parameters getData(){
    return new PController.Parameters( refField.getValue(),
                                       kField.getValue() );
}

public void setParameters( Controller.Parameters parameters ){
    PController.Parameters p = (PController.Parameters) parameters;
    refField.setValue( p.ref );
    kField.setValue( p.k );
}

```

B.3 Compiling JASS 3.0

A Makefile is provided for JASS 3.0 that makes it easy to compile. A Java compiler of at least version 1.3.1 should be used. Before compiling, some parameters need to be set in the makefile:

- JAVAC should be set to the path to the Java compiler.

- SOURCEPATH should be set to the path to the source code.
- DESTPATH should be set to the path where the class files should be put.

When this is done, simply typing *make* or *make all* in the directory where the makefile resides compiles all classes. It is also possible to compile only one class by typing *make classname.class*. This will also compile dependents of the class if their source file has changed. Some other useful make-ables are also present:

- *make resources* should be called when the resource files have been changed. This will copy the resource files in the source directory to the class directory.
- *make clean* removes all class files.
- *make modeljar* will create a jar file containing some common models.

C JASS 3.0 package and classes overview

Table 5 lists all packages in jass3 together with a short description of each. Table 6 lists all classes in JASS3 sorted by package. Classes whose names are written in *italic* are actually not really classes, but interfaces.

Table 5: Overview of packages

jass3	Contains, together with its subpackages, all classes for JASS 3.0
jass3.gui	Contains a class for layout used in the JASS3 gui
jass3.gui.components	Contains, together with its subpackages, some basic classes for the JASS3 gui
jass3.gui.components.ggplot	Contains, classes for the plot in the JASS3 gui
jass3.gui.dialogs	Contains classes used for the dialogs in the JASS3 gui
jass3.gui.layout	Contains, together with its subpackages, all code for JASS 3.0
jass3.gui.panels	Contains some classes that represent different parts of the JASS3 main gui window
jass3.gui.processimage	Contains classes for displaying an image of the ASP plant
jass3.simulation	Contains, together with its subpackages, all code for the simulation part of JASS 3.0
jass3.simulation.controller	Contains controller classes used in JASS 3.0
jass3.simulation.flowgenerator	Contains classes used for managing flows in JASS3
jass3.simulation.iawqasm	Contains some classes used in JASS 3.0 that are closely related to IAWQ ASM1
jass3.simulation.models	Contain some classes representing models of plants for use in JASS3
jass3.utilities	Contains some utility classes used in JASS3

Table 6: All classes in jass3

package jass3	
Jass3	Runs JASS3, either as an applet or as a standalone application.
package jass3.gui	
GuiConstants	Holds some constants used in the jass3 gui
GuiMain	Represents the main gui window of the jass3 simulator.
package jass3.gui.components	
BarDiagram	Displays a bardigram
BarDiagramBeanInfo	BeanInfo class for BarDiagram
DecimalField	Represents a textfields that only accepts decimal numbers as input
package jass3.gui.components.ggplot	
GGPlot	Class for plotting
GGPlotBeanInfo	BeanInfo class for GGPlot
PlotArea	Displays a plot with a user defined number of dataseries with data that belongs to one of two y-axis scales
PlotAreaBeanInfo	BeanInfo class for PlotArea
PlotGrid	Represents a grid to be used in a plot area
PlotLine	Represents a curve, to be used in a plot
PlotScale	Represents a scale for use in a plot
PlotScaleBeanInfo	BeanInfo class for PlotScale
package jass3.gui.dialogs	
ControllerDialog	Creates a dialog for displaying the gui of a Controller
ControllerPanel	Used for displaying multiple controller guis in one dialog
FlowDialog	Dialog for displaying the gui of a Flow-Generator
IncomingDialog	Dialog for setting the composition of the influent water
JassDialog	Parent class for creating dialogs with ok-, apply- and cancelbutton

Table 6: (continued)

MultipleControllerDialog	Dialog used to hold the guis of a number of controllers, where each gets its own panel in a tabbed pane Dialog that allows setting of parameters for the IAWQ ASM1 process Dialog used for setting the volumes of the basins and the settler
ProcessParametersDialog	
VolumesDialog	
package jass3.gui.layout	
PercentLayout	Lays out components within a Container such that each component takes a fixed percentage of the size Used to define constraints for PercentLayout
PercentLayout.Constraint	
package jass3.gui.panels	
BarDiagramPanel	Displays a panel containing three BarDiagrams, each with a combobox for selecting the component to display BeanInfo class for BarDiagramPanel Displays a panel with a GGPlot and three comboboxes for selecting the location and the components to plot BeanInfo class for RTPlotPanel Displays a panel with labels where treatment results and sludge properties can be displayed BeanInfo class for TreatmentResultPanel
BarDiagramPanelBeanInfo	
RTPlotPanel	
RTPlotPanelBeanInfo	
TreatmentResultPanel	
TreatmentResultPanelBeanInfo	
package jass3.gui.processimage	
BasinCollectionComponent	Represents an image of a group of treatment basins, useful for creating an image of an activesludgeprocess layout Represents an image of a basin to be used in a image of a activated sludge process layout Gives a view of the activatedsludge process layout with treatment basins, settler, recirculation etc. Class used for holding pipes, displayed as lines
BasinComponent	
ProcessImageComponent	
ProcessImageComponent.Pipe	

Table 6: (continued)

ProcessImageComponentBeanInfo	BeanInfo class for ProcessImageComponent
SettlerComponent	Provides a view of a settler, where the sludge concentration in the layers are displayed as colors of different intensity
Valve	Provides a visual representation of a valve that can be open or closed
package jass3.simulation	
<i>DataSampler.MultipleSampleTarget</i>	Interface for classes that wish to receive data from a subclass of DataSampler that gives more than one value
<i>DataSampler.SampleSource</i>	Interface for classes that wish to give data to a DataSampler
<i>DataSampler.SampleTarget</i>	Interface for classes that wish to receive data from a DataSampler
DataSampler	Represents a sampler that takes sample from a source periodically and passes the data on to a target
IAWQCompartment	Class used for simulating a compartment in an activated sludge process
IAWQSettler	Class used for simulating a settler in an activated sludge process
JassModel	Represents a model of an activated sludge process consisting of a number of treatmentbasins, a settler, and different "flowpaths" like internal recirculation, return sludge, external carbon and inputs
JassModel.Update	Represents an update to a JassModel
ModelFileHandler	Provides for loading and saving of JassModels to file
ModelGenerator	Contains some data and help methods useful for generating JassModels
SimulationController	Serves as the interface between the gui and the simulator
SimulationControllerCommand	Represents a command for a SimulationController
SimulationDisplayData	Holds simulation data needed for updating gui plot and bardigram

Table 6: (continued)

Simulator	Runs the simulation thread
package jass3.simulation.controller	
CarbonController	This class is a controller used for controlling external carbon in the JASS
CarbonController.Parameters	This class holds parameters for the carboncontroller
CarbonControllerGui	Provides a gui for CarbonController
Controller	This class represents an abstract controller with some parameters like name and such
Controller.Parameters	Empty class which can be subclassed and used for passing parameters to and from controllers
ControllerGui	This class represent the gui of a controller
ControllerVector	A named Vector intended to hold controllers
ExcessSludgeController	This class implements a controller that can be used to controll the excess sludge age so that the sludge age becomes that of a given reference value
ExcessSludgeController.ExcessSludgeSampler	This class iplements a sampler thath samples the data needed for the excesssludgecontroller
ExcessSludgeController.Parameters	This class holds configuration parameters for ExcessSludgeController
ExcessSludgeControllerGui	This class implements a gui for ExcessSludgeController
PIDController	Represents a PID controller. PIDController.Parameters Class containg parameters for PIDController
PIDControllerGui	Provides a gui for PIDController
SelectableZonePID	This class represents a PID controller for which the reference location can be set in its gui
SelectableZonePID.Parameters	Holds parameters for SelectableZonePID
SelectableZonePIDGui	Provides a gui for SelectableZonePID

Table 6: (continued)

SingleOutController	This abstract class represents a controller that controls one variable through one output signal, but may have multiple inputs Hold parameters for SingleOutController Represents a controller with a single input and a single output
SingleOutController.Parameters	
SISOController	
package jass3.simulation.flowgenerator	
<i>MobileOutputController</i>	This interface should be implemented by controller that controls a flowgenerator and lets the user change the outputlocation of the flowgenerator in the gui of the controller This class extends the FlowGenerator class so that the flow can be controlled by a controller. Parameters for ControlledFlowGenerator, used for passing data Can be thought of as representing a pipe, which takes wastewater form one location in the asm proces to another location This class holds Parameters for FlowGenerator and is used for passing data between the gui and the FlowGenerator This class is the gui that enables the user to change the flow and possibly the outputLocation of a FlowGenerator Represent a FlowGenerator with a flow that changes with time and is sinus shaped Parameters used for passing data to and from SinusFlowGenerator Provides a gui for SinusFlowGenerator
ControlledFlowGenerator	
ControlledFlowGenerator.Parameters	
FlowGenerator	
FlowGenerator.Parameters	
FlowGeneratorGui	
SinusFlowGenerator	
SinusFlowGenerator.Parameters	
SinusFlowGeneratorGui	
package jass3.simulation.iawqasm	
Flow	Represents a flow that has a magnitude and concentrations of the iawaasm components

Table 6: (continued)

ProcessParameters	Represents a set of process parameters for the iawqasm process
SettlerParameters	Contains parameters used for simulating a settler with the double-exponential model
State	Encapsulates the state variables of the iawq asm
package jass3.simulation.models	
BenchmarkModel	Represents the plant layout used for Benchmark according to COST
ModelArchive	Encapsulates one or more files containing serialized models
OldJassModel	Gives a model used in the old Jass simulator
package jass3.utilities	
DecimalFormattedDocument	Represents a document formatted as a decimal number, usefull together with TextFields
FormattedDocument	Represents a Document with a format
SelectionSet	Represents a one-to-one mapping between a set of numbers and a set of Strings
SignificantDigitsFormat	Formats decimal number to a given number of significant digits
SignificantDigitsFormat.ScientificDouble	Represents a number with a base-10 exponent and a mantissa

References

- Copp, J.B., Ed.) (2002). *The COST simulation benchmark*. Office for Official publications of the European Communities.
- Henze, M., C. P. L. Grady Jr., W. Gujer, G. v. R. Marais and T. Matsuo (1987). Activated sludge model no. 1. *Scientific and Technical Report No. 1*. IAWQ, London, Great Britain.
- Jeppsson, U. (1996). Modelling aspects of wastewater treatment processes. PhD thesis. Lund Institute of Technology. Dept. of Industrial Electrical Eng. and Automation, Lund, Sweden.
- Luttmer, J. (1995). Design of a simulator for an activated sludge process. Master's thesis. Systems and Control Group, Uppsala University. UP-TEC 95149E.
- Samuelsson, P., M. Ekman and B. Carlsson (2001). A java based simulator of activated sludge processes. *Mathematics and Computers in Simulation* **56**, 333–346.
- Samuelsson, Pär (1998). Jass: A java based activates sludge process simulator. Master's thesis. Uppsala university school of engineering, Department of Systems and Control.
- Skansholm, Jan (1987). *Java direkt*. Studentlitteratur, Lund.
- Sun (2002). Laying out components within a container.
- Tchobanoglous, G. and F.L. Burton (1991). *Wastewater engineering*. McGraw-Hill.