



<http://intevation.net>

**GREAT-ER II**

# **Description GREAT-ER II Model Suite**

Intevation GmbH  
Georgstrasse 4  
49074 Osnabrück  
Germany

Date: July 28, 2003

This document has been designed with L<sup>A</sup>T<sub>E</sub>X. It is available as source code, PDF- and HTML-format.

Version: 1.0.0

Date: July 28, 2003

Author: Frank Koormann

Related documents:

- Specification GREAT-ER II Model Suite, Intevation GmbH, 2003
- GREAT-ER Technical Documentation - Chemical Fate Models Geert Boeije, University of Gent, 1999

Copyright (c) 2002 Intevation GmbH.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Architecture . . . . .	5
1.2	Scheduler Concept . . . . .	7
1.3	Worker Concept . . . . .	8
1.4	Implemented Models . . . . .	8
1.5	Python . . . . .	9
<b>2</b>	<b>Structure and Helper Classes</b>	<b>11</b>
2.1	Segment Class . . . . .	12
2.2	Graph Class . . . . .	17
2.3	Data Class . . . . .	19
2.4	Stochastic Result Class . . . . .	20
2.5	Parameter Class . . . . .	22
<b>3</b>	<b>Core Model System</b>	<b>25</b>
3.1	The ModelList . . . . .	25
3.2	The Model Class . . . . .	26
3.2.1	Parameter information . . . . .	26
3.2.2	Results . . . . .	29
3.2.3	Implemented methods . . . . .	29
3.2.4	Helper Functions . . . . .	33
<b>4</b>	<b>Examples</b>	<b>37</b>
4.1	Model Overview . . . . .	37
4.2	Household Emission . . . . .	41
4.2.1	General . . . . .	42
4.2.2	Setup . . . . .	42
4.2.3	Compute . . . . .	43
4.3	River Mode 2 . . . . .	44
4.3.1	General . . . . .	44
4.3.2	Missing Parameters . . . . .	44
4.3.3	Setup . . . . .	45
4.3.4	Compute . . . . .	46
4.3.5	First order elimination . . . . .	47

**A GNU Free Documentation License**

**49**

# 1 Introduction

This document aims to give an overview about the concepts and implementation of the GREAT-ER II Model Suite. The emphasis lies on illustrating how to implement new models within the Model Suite. Detailed information about the implemented classes and methods can be found in the HTML-based documentation of the class hierarchy<sup>1</sup>.

GREAT-ER is an environmental exposure assessment tool with deterministic models. Variability and parameter uncertainty are considered by applying the Monte-Carlo approach.

The flexible, transparent and extensible reimplementations of the GREAT-ER Model system was a core task of the GREAT-ER II software development project.

## 1.1 Architecture

The architecture of the Model Suite was designed under the requirements of high scalability: The Suite is used in a single user desktop environment as well as on server environments for the GREAT-ER Web Version.

Hence a two component architecture was designed with a Scheduler and a Worker. The Scheduler controls the simulation runs, while a Worker actually performs a simulation. Each user interface, either GREAT-ER Desktop Client or GREAT-ER Web (both Clients in general) are connected to one Scheduler, while a Scheduler can communicate with several Workers. Clients do not communicate directly with a Worker.

This concept allows various combinations:

- Scheduler and one Worker on one computer  
This is the usual configuration for a single user desktop environment. When running the GREAT-ER Desktop Client this combination is started automatically on authentication. The Model Suite can perform one simulation at time.

---

<sup>1</sup>Files: scheduler.html and worker.html

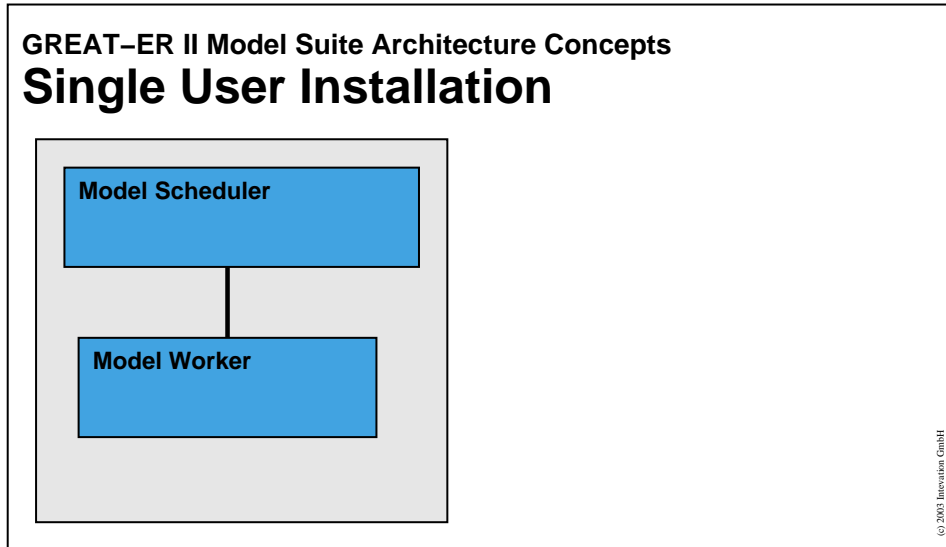


Fig. 1.1: GREAT-ER II Model Suite Architecture: Single User Installation

- Scheduler and several Workers on one computer  
The Scheduler delegates complete simulation jobs to single Workers, the Model Suite can perform as many simulations in parallel as Workers are running. This setup is used for GREAT-ER Web. It is also usable for network installations of GREAT-ER, if the Clients are running on low performance terminals, with the Model Suite running on a computing server.

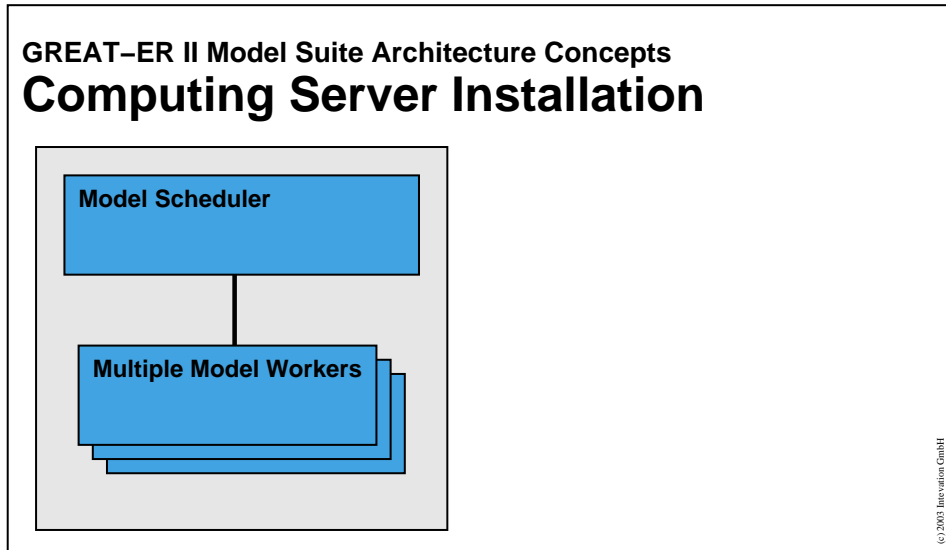


Fig. 1.2: GREAT-ER II Model Suite Architecture: Computing Server Installation

- One Scheduler and multiple Workers on several computers  
The Scheduler splits one simulation job to different workers (running on dif-

ferent computers), the Model Suite can perform at least as many simulations in parallel as Workers are running on different computers. This installation can be used for highly frequented GREAT-ER Web Installations or for computing farms when running high numbers of Monte-Carlo shots on large catchments.

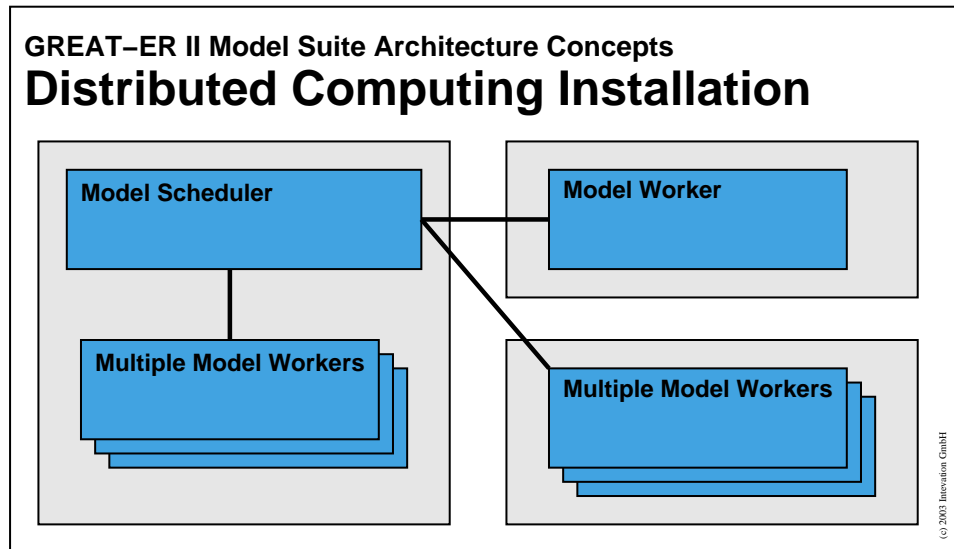


Fig. 1.3: GREAT-ER II Model Suite Architecture: Distributed Computing Installation

## 1.2 Scheduler Concept

The Scheduler is designed independent of specific model implementations for the currently considered compartments. All model requirements are defined by the Worker and handed over to the Scheduler on demand.

The Scheduler performs the following actions:

- Interface "required parameter" requests from a Client to a worker.
- Interface "missing parameter" requests from a Client to a worker.
- Run a simulation requested by a Client:
  1. Request a Worker instance for a list of parameters to be considered as distributed (covers also correlation information) for the simulation run.
  2. Generate batch of Monte-Carlo shots.
  3. Delegate the simulation run to one or more Workers (depending on the scenario as described under 1.1).
  4. During simulation collect progress information from Workers and report to the Client on demand.

5. After the end of the simulation collect all results and write them into the database.

- Stop a running simulation on request.

See the document "GREAT-ER II Model Specification", section "Scheduler" for interface details.

## 1.3 Worker Concept

The Worker implements the GREAT-ER Models and all information on required parameters:

- reply on "required parameters" queries
- reply on "missing parameters" queries
- reply on "stochastic parameters" queries
- compute a batch of Monte-Carlo shots for a specified session.

A Worker is related to one Scheduler process, both communicate via a perspective broker protocol. The Worker was designed to keep most of the administrative overhead (I/O, select loops, process communication) out of the model implementation, which is encapsulated in the Model class (section 3.2).

See the document "GREAT-ER II Model Specification", section "Worker" for interface details.

## 1.4 Implemented Models

To prove the new concept of the GREAT-ER II Model Suite the models of GREAT-ER 1.0 have been reimplemented. These cover four compartments with various levels of model complexity:

- Emission:  
For household chemicals the emission is calculated from the number of inhabitants connected to a WWTP and the per year per capita consumption of a chemical. Additional inputs into the WWTP can be specified as mass per year separately.
- Sewer:  
A percentage removal is applied for the sewer system, the model can be switched off.

- Waste Water Treatment Plant (discharge into river):  
The WWTP component covers a primary settler, and a main processing step, either activated sludge or trickling filter. For both percentage removal is applied, the activated sludge chain (including the primary settler) is also implemented similar to the SimpleTreat model (Boeije, section 5.2).
- River:  
The river model implements an extended version of the WATER model, a one-dimensional first-order elimination model. The elimination rate can be considered with three different complexities:
  1. Over all elimination rate (from literature)
  2. Partitioning and separate elimination rates for sorbed and dissolved fractions and volatilization.
  3. Separate processes: Hydrolysis, photolysis, bio-degradation, sedimentation and volatilization.

Chapter 4 (page 37) illustrates by some examples how to implement a model for the GREAT-ER II Model Suite. Details about the implemented models are described in the "GREAT-ER Technical Documentation - Chemical Fate Models".

## 1.5 Python

Python<sup>2</sup> is an object-oriented scripting language with byte compilation available as Free Software. It provides multiple inheritance, overloading of methods and operators as well as polymorphisms. Aside the usual data types specific types as dynamic arrays, lists, tuples and dictionaries are available.

Python is an easy to learn programming language resulting in a transparent source code due to the indentation oriented structuring scheme. The basic distribution is enhanced by numerous modules for various tasks, partly especially for scientific computing. Python programs can be extended with C-implemented modules for time-consuming computations.

---

<sup>2</sup><http://www.python.org>



## 2 Structure and Helper Classes

The backbone of the system structure of GREAT-ER 1.0 was the river network. The network is split into stretches of different types: spring, confluence, bifurcation and discharge. Based on these elements the river network can be seen as a directed graph, where the first three stretch types have mainly topological meaning. The discharge element is more special because an additional chain of models is embedded: Emission, sewer and WWTP. However, the graph structure is limited to the river network. Figure 2.1 illustrates the situation.

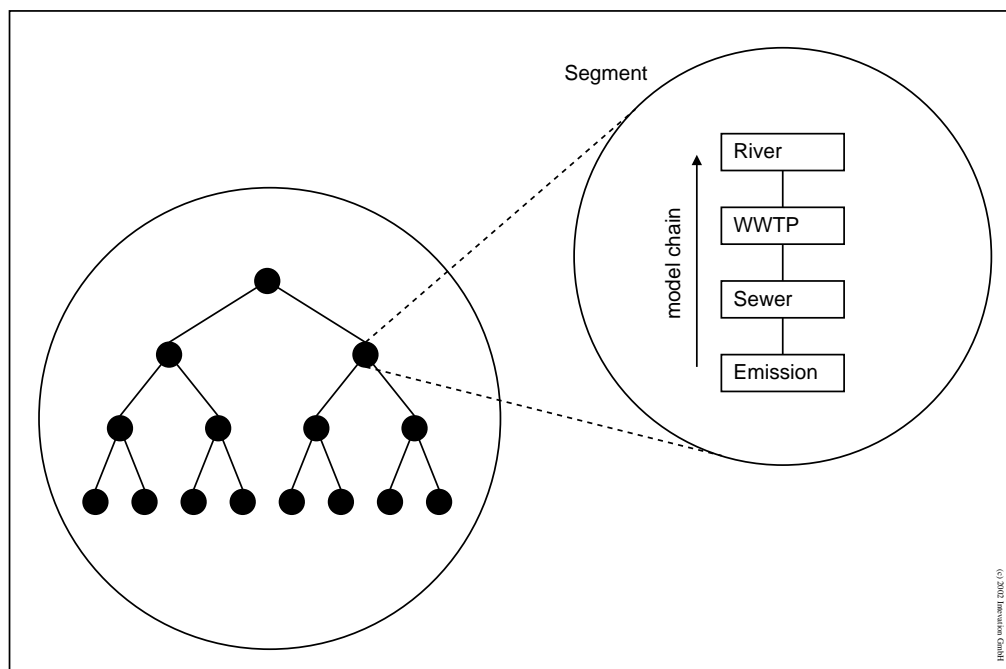


Fig. 2.1: System structure GREAT-ER 1.0

The approach for GREAT-ER II extends the segment concept to all models. The directed graph is not limited to the river network with a static chain of models linked to a discharge point as described above. The different models are represented by specific segments and therewith part of the directed graph (Figure 2.2). This approach provides higher flexibility in extending and combining the model collection.

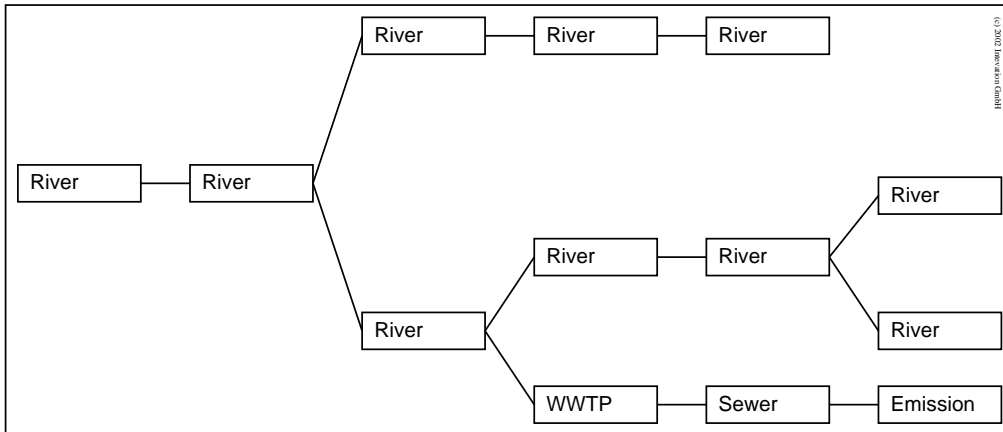


Fig. 2.2: New system structure for GREAT-ER II

The structure implemented by two class-hierarchies, the **Segment** class and the **Graph** class.

For a better understanding: Child/Parent terminology in the following section is used with view from the graph root (which must be unique): In relation to a given segment the child segment is the upstream segment, the parent segment is the downstream one.

## 2.1 Segment Class

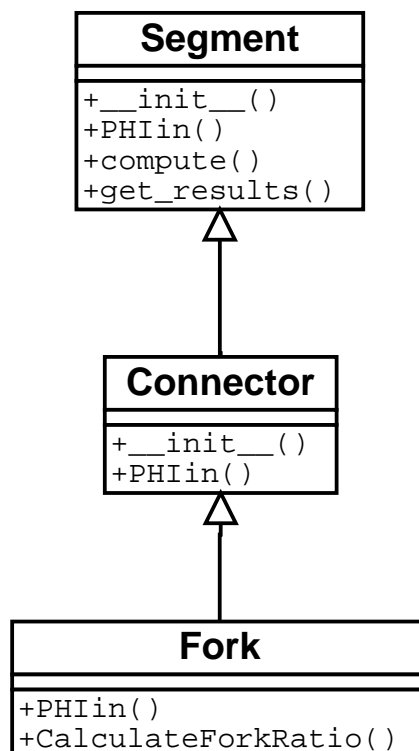


Fig. 2.3: Class Hierarchy Segment

The **Segment** class hierarchy implements the core of the directed graph, the segment objects, as described above. The graph is binary, therewith three types of segments are sufficient to build the graph: a simple segment, a connector and a fork. The segments implementation in independent from the model implementations following the directed graph concept.

## Segment

The class **Segment** implements a simple segment, with one or none parent and one or no child. Methods:

```
__init__(self, par, model)
```

<b>Name</b>	<code>__init__</code>
<b>Description</b>	Constructor: The <code>__init__</code> method initializes instance variables provided by the parameter dictionary and a pointer to the model object depending on the segment type.
<b>In</b>	<code>par, model</code>
<b>In definitions</b>	<code>par:</code> Parameter dictionary <code>model:</code> Pointer to model object
<b>Return</b>	-
<b>Return definitions</b>	-

```
PHIin(self)
```

<b>Name</b>	<code>PHIin</code>
<b>Description</b>	Calculate the segment mass flux input from the child segment mass flux output.
<b>In</b>	-
<b>In definitions</b>	-
<b>Return</b>	<code>PHIin</code>
<b>Return definitions</b>	Return the input mass flux of the segment.

`compute(self, shot_num)`

<b>Name</b>	<code>compute</code>
<b>Description</b>	Run the segment model and set the instance variable <code>PHIout</code> .
<b>In</b>	<code>shot_num</code>
<b>In definitions</b>	<code>shot_num</code> : Number (identifier) of Monte-Carlo shot.
<b>Return</b>	-
<b>Return definitions</b>	-

`get_results(self)`

<b>Name</b>	<code>get_results</code>
<b>Description</b>	Return the results for the segment.
<b>In</b>	-
<b>In definitions</b>	-
<b>Return</b>	Tuple ( <code>segmentid</code> , <code>segment.model.results</code> ).
<b>Return definitions</b>	Returned is a tuple of <code>segmentid</code> (which uniquely identifies the result set and <code>segment.model.results</code> (a list of <b>StochasticResult</b> objects (see 2.4)).

## Connector

The class `Connector` connects two segments (i.e. it has two children). It inherits from `Segment` and overrides some methods:

`__init__(self, par, model)`

<b>Name</b>	<code>__init__</code>
<b>Description</b>	Constructor: The <code>__init__</code> method initializes instance variables provided by the parameter dictionary and a pointer to the model object depending on the segment type. In difference to <code>Segment.__init__</code> instance variables for both children are initialized.
<b>In</b>	<code>par</code> , <code>model</code>
<b>In definitions</b>	<code>par</code> : Parameter dictionary <code>model</code> : Pointer to model object
<b>Return</b>	-
<b>Return definitions</b>	-

PHIin(self)

<b>Name</b>	PHIin
Description	Calculate the segment mass flux input as sum of the mass flux output of both children.
In	-
In definitions	-
Return	PHIin
Return definitions	Return the input mass flux of the segment.

## Fork

The class Fork inherits from the Connector class. In combination with a second fork segment this class represents a bifurcation in a river network. The segment stores a pointer to the related fork segment using the second child variable structures. This is similar to the data stored in the stretch\_tab, as a consequence a fork segment has always only one child segment.

PHIin(self)

<b>Name</b>	PHIin
Description	Calculate the segment mass flux from child segment with splitting to related fork segment.
In	-
In definitions	-
Return	PHIin
Return definitions	Return the input mass flux of the segment.

CalculateForkRatio(self)

<b>Name</b>	CalculateForkRatio
Description	Initially calculate fork ratio, depending on flow for the two segments. The calculation has to be performed after the entire graph is build and all relations are determined.
In	-
In definitions	-
Return	-
Return definitions	-

## 2.2 Graph Class

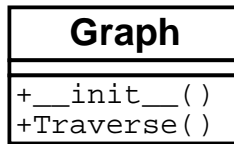


Fig. 2.4: Graph Class

Although implementing only two methods, the **Graph** Class provides the backbone feature of a **Worker**. A graph object stores the directed graph representing the system under investigation as described above.

```
__init__(self, session)
```

<b>Name</b>	<code>__init__</code>
<b>Description</b>	<p>The <code>__init__</code> method initializes the graph in various steps:</p> <ul style="list-style-type: none"> <li>• Derive the list of stochastic parameters per model type.</li> <li>• Instantiates all segments according to the structure information read from the database.</li> <li>• Bind appropriate model objects to all segments.</li> <li>• Call the <code>setup</code> method of each segments model object (see below, section 3.2.3) to complete initialization.</li> <li>• Link all segments with its child segments.</li> <li>• Perform a final test on network integrity.</li> <li>• Call the <code>CalculateForkRatio</code> method for all bifurcation segments.</li> <li>• Derive a list representing a post-order traverse of the directed graph.</li> </ul>
<b>In</b>	<code>session</code>
<b>In definitions</b>	<code>session:</code> Pointer to the <b>Data</b> object (see 2.3) storing the session information and data.
<b>Return</b>	-
<b>Return definitions</b>	-

Traverse(self)

<b>Name</b>	Traverse
<b>Description</b>	Return the post-order traverse for the graph.
<b>In</b>	-
<b>In definitions</b>	-
<b>Return</b>	List of <b>Segment</b> objects.
<b>Return definitions</b>	The returned list of objects represents a post-order traverse of the graph.

Since the graph must be strictly binary, the insertion of virtual segments is necessary during initialization, e.g. if a confluence segment is a river stretch receiving a discharge. Such virtual stretches are instantiated with a neutral model only handing over the mass flux.

The information about a segment and its model is stored in two instance variables of a segment:

- **ID:**  
Unique string to identify the segment, combination of general model type and ID as read from the database. The general model type can be "H" (household), "S" (sewer), "D" (discharge), "R" (river) or "V" (virtual). This type information is evaluated when storing the simulation results (currently discharge and river results).
- **type:**  
A more general key storing model specific segment type information required while building the graph and deriving the traverse.

Source:	0	Discharge:	4
Normal Segment:	1	Sewer:	5
Confluence:	2	Household:	6
Bifurcation:	-2		
Lake:	3		

## 2.3 Data Class

Data
+__init__() +disconnect() +connect() +LoadDischargeClasses() +LoadDischargeData() +LoadEmissionData() +LoadEnvironmentData() +LoadModelModes() +LoadSessDischData() +LoadStretchClasses() +LoadStretchData() +LoadSubstanceData() +LoadDiffuseInputDSS()

Fig. 2.5: Data Class

The **Data** class implements the read access to the data base. Initialized with a session ID all required data can be loaded depending on the requested Worker action. A worker has one **Data** object, all Model objects store a pointer to this global source of data. The **Data** object stores connection data and the session ID and provides a dictionary (`data`), which keys are set according to the loaded data (e.g. 'SESSION', 'STRETCH', 'DISCHARGE', 'SUBST', etc.).

Each parameter can be identified by its BlockID, FieldID pair as used in the database `PARA_TREE_DEF_TAB`. Tabular data can be accessed by a BlockID, RecordID pair in combination with the needed FieldID. Please note that as used in this example, the **Data** object is usually stored in the variable `session`:

- `session.data['STRETCH'][1].q_val`: mean flow value of the stretch with ID 1.
- `session.data['SUBST']['K_OW'].value`:  $K_{OW}$  from the substance data.

## 2.4 Stochastic Result Class

<b>StochasticResult</b>
<pre> +__init__() +add() +combine() +get() +get_raw() +reset() </pre>

Fig. 2.6: StochasticResult Class

The `StochasticResult` Class implements an object to collect stochastic results to compute mean and standard deviation. Internally the object stores the number of sampled values ( $n$ ), sum of values ( $\sum_i^n x_i n$ ) and the sum of value squares ( $\sum_i^n x_i^2$ ).

From these mean and standard deviation can be derived:

$$\mu = \frac{\sum_i^n x_i}{n} \tag{2.1}$$

$$\sigma = \sqrt{\frac{\sum_i^n x_i^2 - \frac{(\sum_i^n x_i)^2}{n-1}}{n-1}} \tag{2.2}$$

`__init__(self)`

<b>Name</b>	<code>__init__</code>
Description	Constructor: Instantiate the <code>StochasticResult</code> object.
In	-
In definitions	-
Return	-
Return definitions	-

`add(self, value)`

<b>Name</b>	
Description	Add a value to the collection.
In	<code>value</code>
In definitions	<code>value:</code> <code>float value</code>
Return	-
Return definitions	-

`combine(self, param)`

<b>Name</b>	<code>combine</code>
Description	Combine two <b>StochasticResult</b> objects by adding the descriptive instance variables (sampled values, sum of values, sum of value squares).
In	<code>param</code>
In definitions	<code>param:</code> <b>StochasticResult</b> object to be combined with the object
Return	-
Return definitions	-

`get(self)`

<b>Name</b>	<code>get</code>
Description	Return mean, standard deviation and number of sampled values, derived from instance variables as described in equations 2.1 and 2.2
In	-
In definitions	-
Return	Tuple (mean, stddev, num)
Return definitions	Returned are the descriptive parameters of the stochastic result: Mean, standard deviation and the number of single results the stochastic result is based on.

`get_raw(self)`

<b>Name</b>	<code>get_raw</code>
Description	Return the raw sums as described above. Helpful in case combining numerous stochastic results in other applications, when the <b>StochasticResult</b> class is not usable.
In	-
In definitions	-
Return	Tuple of ( <code>sum</code> , <code>sumsqr</code> , <code>n</code> )
Return definitions	Return the sum, sum of squares and number of collected values.

`reset(self)`

<b>Name</b>	<code>reset</code>
Description	Reset the object to initial state (sums and numbers set back to zero).
In	-
In definitions	-
Return	-
Return definitions	-

## 2.5 Parameter Class

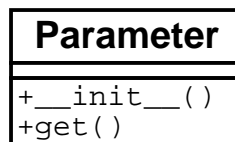


Fig. 2.7: Parameter Class

The Parameter Class encapsulates the handling of distributed parameters. Supported are undistributed parameters, normal, logarithmic normal and uniform distributions:

```
__init__(self, key, parameter, stochastic_parameter_list,
stochastics, ID = None)
```

<b>Name</b>	<code>__init__</code>
<b>Description</b>	Constructor: Initialize the parameter encapsulation as normal, logarithmic normal and uniform or undistributed parameter. If the information about distributions is incomplete on instantiation an undistributed parameter is instantiated.
<b>In</b>	<code>key, parameter, stochastic_parameter_list, stochastics, ID</code>
<b>In definitions</b>	<p><code>key:</code> Tuple (<code>blockid, fieldid</code>) identifying the parameter in the session data.</p> <p><code>parameter:</code> String representation of the parameter (either single value or semicolon separated pair of descriptive values)</p> <p><code>stochastic_parameter_list:</code> List of stochastic parameters (see 3.2.1 for list details). The actual distribution type of the parameter is derived from this list (if the parameter is found).</p> <p><code>stochastics:</code> Pointer to the batch (a dictionary of lists) of Monte-Carlo shot values.</p> <p><code>ID:</code> (optional) identifier for distinct segments. This can be used to use separate distributions for specific objects (if derived by <code>Model.StochasticParams</code>, see 3.2.3).</p>
<b>Return</b>	-
<b>Return definitions</b>	-

```
get(self, shot)
```

<b>Name</b>	<code>get</code>
<b>Description</b>	Return value from the parameter for the given Monte-Carlo shot considering the distribution determined during instantiation.
<b>In</b>	<code>shot</code>
<b>In definitions</b>	<code>shot:</code> Number (identifier) of the current Monte Carlo shot.
<b>Return</b>	<code>value</code>
<b>Return definitions</b>	Float value from parameter distribution for current Monte-Carlo shot.



## 3 Core Model System

As mentioned above the new GREAT-ER Model Suite has been designed to keep all administrative tasks out of the central model implementation, to make implementation of new models within the given framework easier. While the previous chapter introduced some objects and methods used to implement a model this chapter is concentrating on the model itself.

The core of the GREAT-ER II Model Suite is represented in two files, `model.py` and `modelbase.py`, while the implementation of specific models is placed in separate files.

- `model.py`  
provides a unique interface to all implemented models. It imports all models and lists available models in the `ModelList`.
- `modelbase.py`  
provides the basic `Model` class from which all implemented models are inherited.

### 3.1 The ModelList

The `ModelList` stores a list of all models implemented by the GREAT-ER II Model Suite as a list of Python tuples. Hook in each new implemented model here with an entry of three values.

The following grammar in Backus-Naur notation describes the structure of the list:

[...] refers to a Python list, (...) to a Python tuple.

```
ModelList      : [ Models ]
```

```
Models        :  
              | Model, Models  
              | Model
```

```
Model         : ( FlagKey, Model Name, Model Group )
```

- `Flag`: String  
Key of item in model mode section flagging a model mode (e.g. 'MODE\_SEWER').  
If value is *None* the model is assumed to be applied in any mode.

- **Model Name:** String  
 Name of the class implementing the given model, e.g. 'SewerModel'.
- **Model Group:** String  
 Model group the model belongs to. Currently considered are: 'Emission', 'Sewer', 'River', 'DischargePrimary', 'DischargeAS' and 'DischargeTF'.

## 3.2 The Model Class

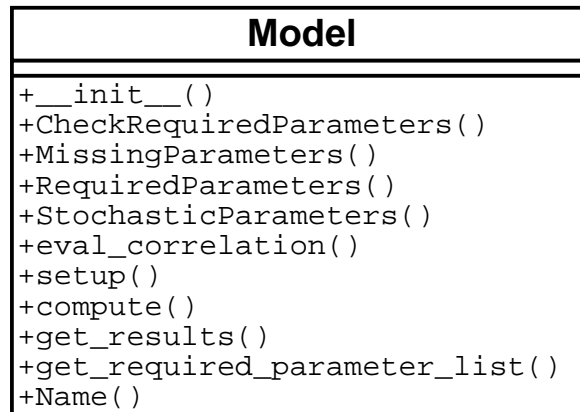


Fig. 3.1: Model Class

The methods of the Model class fall in four groups: Initialization, Parameter Information, Simulation and Administrative Framework. Depending on the specific model implementation some of the methods have to be overridden. The methods are discussed below followed by some examples illustrating the implementation of models with the GREAT-ER II Model Suite.

### 3.2.1 Parameter information

The GREAT-ER II Model Suite architecture requires the parameters needed for a simulation to be hard coded in the model. Since all model equations are implemented here as well a unique location is available to store the most relevant information on a model in one place.

#### Model Parameter Description

This section describes the two structures used for the parameter description. Their interpretation in the methods to answer the Scheduler's parameter requests are discussed below (3.2.3).

**Required parameters** The parameters required for a model are stored in a list represented by the variable `_required_parameters`. The list stores a sublist for each model implemented by a model class.

```
_required_parameters : [ Lists ]

Lists                :
                    | List, Lists
                    | List

List                 : [ Mode, Parameters ]

Mode                 : ( BlockID, FieldID, value )

Parameters           : Parameter, Parameters
                    | Parameter

Parameter            : ( BlockID, FieldID )
```

The parameters are represented by their BlockID, FieldID pairs, as used also in the database `PARA_TREE_DEF_TAB`. The Mode-Descriptor identifies a special parameter, on which is decided if the model is used in the current model mode setting or not: If the identified parameter for the current session equals the defined value, the model is to be used.

**Note:** If a model is to be used always, independent from the mode selection, the `Mode` tuple has to be set to *(None, None, None)*.

### Correlated parameters

Similar to the `_required_parameter` list the `_correlated_parameter` list describes all parameters which are considered as correlated by the model implementation:

```
_correlated_parameters : [ List ]

List                  :
                    | CorrelationInfo, List
                    | CorrelationInfo

CorrelationInfo       : ( Parameter, Parameter, Correlation )

Correlation           : Parameter
                    | numerical value [-1.0,1.0]

Parameter             : ( BlockID, FieldID )
```

As described, the correlation information consists of the elements:

- the correlated parameter,
- the parameter the current parameter is correlated with,

- the correlation as either a fixed value or a third parameter describing the correlation.

### Parameter Information Output

Based on the lists described above the models parameter information methods derive two different lists on demand.

**Parameter List** The parameter list simply lists parameters identified by a BlockID, FieldID tuple. Lists of this type are used for required and missing parameters:

```
parameter_list      : [ List ]

List                :
                    | Parameter, List
                    | Parameter

Parameter           : ( BlockID, FieldID )
```

**Stochastic Parameter List** is returned by the StochasticParameters method and lists all distributed parameters with their correlation information:

```
stochastic_parameters : [ List ]

List                  :
                    | StochasticParameter, List
                    | StochasticParameter

StochasticParameter  : (( BlockID, FieldID, Distribution, ObjectID ),
                        Correlation )

Distribution           : 'none'
                    | 'normal'
                    | 'lognormal'
                    | 'uniform'

ObjectID              : '-1'
                    | Identifier of a segment

Correlation           : None
                    | Parameter
                    | numerical value [-1.0,1.0]

Parameter             : ( BlockID, FieldID )
```

### 3.2.2 Results

Results are stored in a dictionary (`Results`) as pointers to `StochasticResult` objects (see 2.4). The method `get_results` provides external access on the results.

### 3.2.3 Implemented methods

The `Model` class implements a set of generic methods necessary to provide all model information and to run a simulation. Depending on the specific model various methods have to be overridden by a model implementation (as illustrated below, section 4), at least `setup` and `compute` which implement the actual data access and model computation.

```
__init__(self, session, st_params)
```

<b>Name</b>	<code>__init__</code>
<b>Description</b>	Constructor: Called during graph initialization (2.2). The initialization is completed later with the <code>setup</code> method. The method is usually not overridden.
<b>In</b>	<code>session, st_params</code>
<b>In definitions</b>	<code>session:</code> Pointer to the sessions data object (2.3) <code>st_params:</code> Pointer to the batch of Monte-Carlo shots (2.2)
<b>Return</b>	-
<b>Return definitions</b>	-

```
setup(self, segmentID)
```

<b>Name</b>	<code>setup</code>
<b>Description</b>	The <code>setup</code> method is used to initialize the parameters needed for the model for the specific segment from the <code>Data</code> object. The method is called with the <code>segmentID</code> (which is fixed at this stage) as a reference to the segment in the graph. In addition the <code>StochasticResult</code> objects needed by the model are initialized and stored in in the <code>Result</code> dictionary. Since the generic method does not load any data it has to be overridden when implementing a new model.
<b>In</b>	<code>segmentID</code>
<b>In definitions</b>	<code>segmentID:</code> Identifier of the graph's <code>Segment</code> object the <code>Model</code> object is related with.
<b>Return</b>	-
<b>Return definitions</b>	-

`compute(self, segment, shot_num)`

<b>Name</b>	<code>compute</code>
<b>Description</b>	The <code>compute</code> method finally implements the model equations. The generic implementation simply hands through the input mass flow, hence the method has to be overridden usually.
<b>In</b>	<code>segment, shot_num</code>
<b>In definitions</b>	<code>segment:</code> Pointer to the segment the model is linked to to call the segments <code>PHIin</code> method. : Number of Monte-Carlo shot to obtain the correct values from the models <code>Parameter</code> objects.
<b>Return</b>	<code>PHIout</code>
<b>Return definitions</b>	The model returns the mass flux out of the segment for the current Monte-Carlo shot.

`get_results()`

<b>Name</b>	<code>get_results</code>
<b>Description</b>	Return the models <code>StochasticResult</code> objects, referenced in the <code>Result</code> dictionary.
<b>In</b>	-
<b>In definitions</b>	-
<b>Return</b>	Dictionary of <code>StochasticResult</code> objects.
<b>Return definitions</b>	Returned is a dictionary of <code>StochasticResult</code> objects, from which the descriptive values can be interrogated. The dictionary's keys clearly identify the separate objects.

`CheckRequiredParameters(self, parameter_list=None)`

<b>Name</b>	<code>CheckRequiredParameters</code>
<b>Description</b>	On initialization of the worker the method is called to check the required parameter definition syntax. By default ( <code>parameter_list</code> is <code>None</code> ) process the model's <code>_required_parameters</code> list. If available the method is called recursively with sublists of <code>_required_parameters</code> . The method is usually not overridden.
<b>In</b>	<code>parameter_list</code>
<b>In definitions</b>	<code>parameter_list</code> List of parameters to check
<b>Return</b>	Boolean
<b>Return definitions</b>	The method returns a boolean value identifying correctness.

RequiredParameters(self, parameter\_list=None)

<b>Name</b>	RequiredParameters
<b>Description</b>	Return the list of parameters required by the model according to the model mode selections. By default ( <code>parameter_list</code> is <code>None</code> ) process the model's <code>_required_parameters</code> list. If available the method is called recursively with sublists of <code>_required_parameters</code> . Depending on the complexity of the implemented model it can be necessary to override the <code>RequiredParameters</code> method (see 4).
<b>In</b>	<code>parameter_list</code>
<b>In definitions</b>	<code>parameter_list</code> Raw list of required parameters
<b>Return</b>	Parameter list
<b>Return definitions</b>	A list of parameters required for the current model settings, as defined in section 3.2.1.

MissingParameters(self, parameter\_list=None)

<b>Name</b>	MissingParameters
<b>Description</b>	Return the list of parameters still missing in the sessions parameter sets according to the model mode selections. By default ( <code>parameter_list</code> is <code>None</code> ) process the model's <code>_required_parameters</code> list. If available the method is called recursively with sublists of <code>_required_parameters</code> . If the implemented model provides parameter estimate functions the method has to be overridden to take these dependencies into account (see 4).
<b>In</b>	<code>parameter_list</code>
<b>In definitions</b>	<code>parameter_list</code> Raw list of required parameters
<b>Return</b>	Parameter list
<b>Return definitions</b>	A list of parameters required for the current model settings but not yet set to a valid value is returned. The format is defined in section 3.2.1.

```
StochasticParameters(self, parameter_list=None,
correlated_parameters=None)
```

<b>Name</b>	StochasticParameters
<b>Description</b>	Return the list of parameters considered as distributed for the specific segment according to the to the model mode selections and parameter settings. By default ( <code>parameter_list</code> and <code>correlated_parameters</code> are <i>None</i> ) process the model's lists. If available the method is called recursively with sublists of <code>_required_parameters</code> . The method calls <code>eval_correlation</code> to determine correlation dependencies. Due to its generic structure the method has usually not to be overridden for a model implementation.
<b>In</b>	<code>parameter_list</code> , <code>correlated_parameters</code>
<b>In definitions</b>	<code>parameter_list</code> : Raw list of required parameters <code>correlated_parameters</code> : Correlation information
<b>Return</b>	Parameter list.
<b>Return definitions</b>	A list of correlated parameters with correlation information according to the current model settings, as defined in section 3.2.1.

```
eval_correlation(self, key, correlated_parameters)
```

<b>Name</b>	<code>eval_correlation</code>
<b>Description</b>	The method is called by <code>StochasticParameters</code> to evaluate potential correlation dependencies. Since it in turn makes use of the model's <code>StochasticParameters</code> method to derive the full tuple for a stochastic parameter description it is implemented as a class method instead of a simple helper function.
<b>In</b>	<code>key</code> , <code>correlated_parameters</code>
<b>In definitions</b>	<code>key</code> : Tuple ( <code>blockid</code> , <code>fieldid</code> ) to identify a parameter <code>correlated_parameters</code> : List of tuples providing correlation information according to 3.2.1.
<b>Return</b>	<i>None</i> or Tuple: ( <code>blockid</code> , <code>fieldid</code> , <code>distribution</code> , <code>local</code> ), <code>correlation</code> )
<b>Return definitions</b>	If the tested parameter is correlated with a second one, return a tuple of parameter identification, distribution information and correlation information (as defined under 3.2.1).

Name(self)

<b>Name</b>	Name
Description	Return the model name.
In	-
In definitions	-
Return	name
Return definitions	The model name as defined in <code>self.name</code> .

get\_required\_parameter\_list(self)

<b>Name</b>	get_required_parameter_list
Description	Return the required parameter definition list.
In	-
In definitions	-
Return	Parameter list.
Return definitions	Return the models list of required parameters ( <code>self._required_parameters</code> ).

### 3.2.4 Helper Functions

In addition to the Model class the `modelbase.py` provides some helper functions needed for model implementation. These functions wrap some frequently used checks or actions to keep administrative overhead out of the model implementation and make it more readable.

`eval_parameter(data, parameter)`

<b>Name</b>	<code>eval_parameter</code>
<b>Description</b>	The function wraps the lengthy check if a parameter (specified by a sequence of keys in the <code>parameter</code> value) exists in the data dictionary.
<b>In</b>	<code>data, parameter</code>
<b>In definitions</b>	<p><code>data</code>: Pointer to the sessions data dictionary.</p> <p><code>parameter</code>: A tuple of (<code>BlockID</code>, <code>FieldID</code>, <code>value</code>) to identify a parameter and provide access to its contents. (e.g. (<code>'SUBST'</code>, <code>'K_OC'</code>, <code>'value'</code>), to access the substance <math>K_{OC}</math> parameter or (<code>'SESS_DISCH'</code>, <code>'12'</code>, <code>'r_wwtp'</code>) to access the local <math>R_{WWTP}</math> parameter for the discharge the the ID 12.</p>
<b>Return</b>	Value or <i>None</i>
<b>Return definitions</b>	Return parameter value as read from the data (no further type conversion or processing is applied) if it exists, else <i>None</i> .

`remove_parameter(parameter_list, parameter)`

<b>Name</b>	<code>remove_parameter</code>
<b>Description</b>	<p>Try to remove parameter from a list.</p> <p>The method is frequently used in the <code>MissingParameters</code> methods if a model provides estimate functions and catches the exception if the parameter does not exists in the list.</p>
<b>In</b>	<code>parameter_list, parameter</code>
<b>In definitions</b>	<p><code>parameter_list</code>: List of parameters ((<code>BlockID</code>, <code>FieldID</code>) tuples)</p> <p><code>parameter</code>: Tuple (<code>BlockID</code>, <code>FieldID</code>) to be removed from the list.</p>
<b>Return</b>	-
<b>Return definitions</b>	-

`eval_local_setting(data, local, general)`

<b>Name</b>	<code>eval_local_setting</code>
<b>Description</b>	Test if a local setting is available which overwrites the general setting (e.g. consumption data).
<b>In</b>	<code>data, local, general</code>
<b>In definitions</b>	<code>data:</code> Pointer to the sessions data dictionary. <code>local:</code> Tuple ( <code>BlockID</code> , <code>FieldID</code> , <code>value</code> ) identifying a potential local value. <code>general:</code> Tuple ( <code>BlockID</code> , <code>FieldID</code> , <code>value</code> ) identifying the general local value.
<b>Return</b>	Tuple ( <code>value</code> , <code>flag</code> )
<b>Return definitions</b>	The returned tuple provides the parameter to use with some further information: <code>value</code> is the local value if available, otherwise the general one. <code>flag</code> is a boolean flagging if the value is local ( <i>True</i> ) or general ( <i>False</i> ).

`eval_local_stochastics(status, stochastics)`

<b>Name</b>	<code>eval_local_stochastics</code>
<b>Description</b>	Test if a distributed parameter is overwritten by a local value.  The function is used in combination with the Parameter Object <code>_init_</code> method.
<b>In</b>	<code>status, stochastics</code>
<b>In definitions</b>	<code>status:</code> Boolean flag, the return value of <code>eval_local_setting</code> <code>stochastics:</code> Pointer to the batch of Monte-Carlo shots.
<b>Return</b>	<i>None</i> or <code>stochastics</code>
<b>Return definitions</b>	The function evaluates <code>status</code> and returns <i>None</i> on <i>True</i> , <code>stochastics</code> else.



# 4 Examples

## 4.1 Model Overview

Based on the Model Class and its framework the GREAT-ER II Model Suite implements the models of GREAT-ER 1.0:

- Household Emission
- Sewer
  - No Removal
  - Percentage Removal
- Discharge
  - Primary Settler: Percentage Removal and SimpleTreat
  - Activated Sludge: Percentage Removal and SimpleTreat
  - Trickling Filter: Percentage Removal
- River
  - First order elimination
  - First order elimination with fractioning
  - First order elimination with combining elimination rate from detailed processes

All models are inherited from the base Model class. Further levels of inheritance are depending on the specific models.

### Household emission

The household emission model implements an approach to calculate the emission from annual per capita consumption and the number of inhabitants at a discharge site.

## Sewer

The sewer model is split into two modes: One with no elimination (mass flux is simply handed over) and a second with percentage removal.

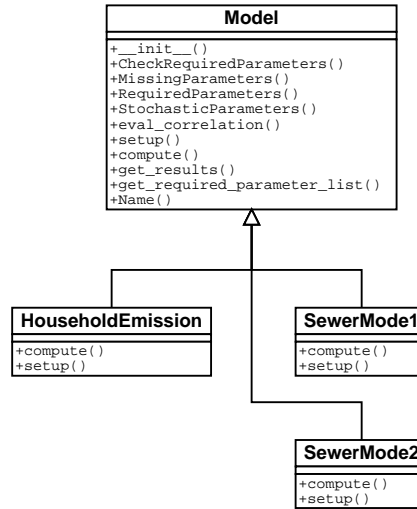


Fig. 4.1: Class Hierarchy Model (Emission and Sewer Models)

## Discharge

The discharge model module implements various complexity modes for different types of treatment plants: The mode 1 is implemented in a generic model from which the specific models Primary Settler, Activated Sludge and Trickling Filter are inherited. For first two also a second mode (Simple Treat) is implemented inherited from Model with a mix-in of some helper methods needed for both.

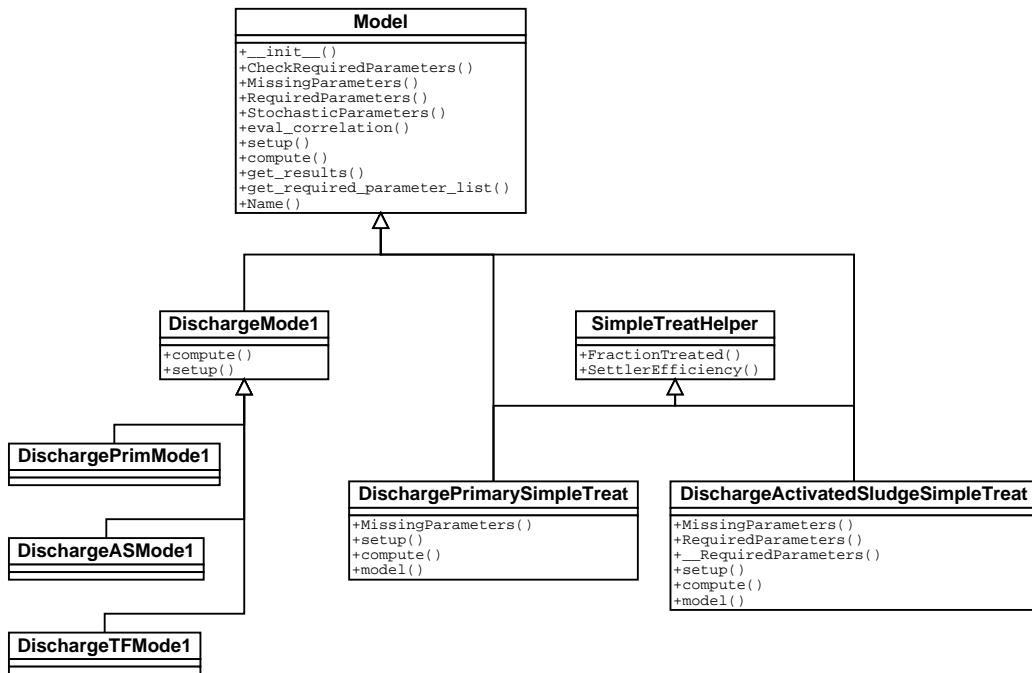


Fig. 4.2: Class Hierarchy Model (Discharge Models)

## River

The three river model modes have the first order elimination approach in common. Hence this is implemented as a method of mode 1 from which the other two mode implementation inherit and override the elimination rate calculation.

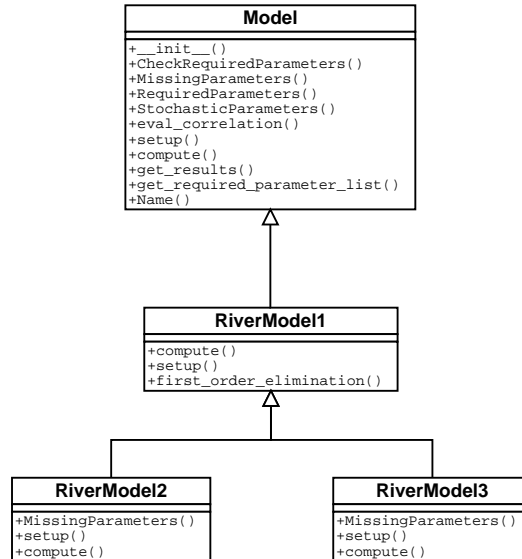


Fig. 4.3: Class Hierarchy Model (River Models)

In the following some model implementation are described in detail to illustrate the use of the classes and methods described above.

## 4.2 Household Emission

The household emission model implements an approach to calculate the emission from annual per capita consumption and the number of inhabitants at a discharge site. The simple approach allows a clear overview over the elements of the Model class to be usually overridden and the concepts to implement a model using the GREAT-ER II Model framework.

```
class HouseholdEmission( Model ):
    """Implements GREAT-ER Household Emission Model."""

    # The emission model is applied always, independent of the model mode
    # selection
    _required_parameters = [(None , None, None),
        ('DISCH', 'POP'), ('SUBST', 'DEFAULT_CONSUMPTION')
    ]

    name = 'Household Emission Model'

    def setup(self, segmentID):
        """Set up Household Emission Model."""

        # Derive stochastic paramaters
        st_params = self.st_params

        # Shortcuts
        data = self.session.data
        ID = segmentID

        # Some parameter which only have to be processed once
        self.ID = ID
        self.Pop      = float(eval_local_setting(data,
            ('SESS_DISCH', ID, 'pop'),
            ('DISCH', ID, 'pop')
        )[0])
        Consumption, status = eval_local_setting( data,
            ('EMISSION', ID, 'consumption'),
            ('SUBST', 'DEFAULT_CONSUMPTION', 'value')
        )
        self.Consumption = Parameter(('SUBST', 'DEFAULT_CONSUMPTION'),
            Consumption,
            eval_local_stochastics(status, st_params),
            data['STOCHASTICS'])

    def compute( self, segment, shot_num ):
        """Compute Household Emission Model."""

        Pop      = self.Pop
        Consumption = self.Consumption.get(shot_num)

        PHIout = Pop * Consumption / 31536000.0          # [kg/s]

        return PHIout
```

The model implementation is explained in sections below.

## 4.2.1 General

```
class HouseholdEmission( Model ):
    """Implements GREAT-ER Household Emission Model."""

    # The emission model is applied always, independent of the model mode
    # selection
    _required_parameters = [(None, None, None),
                           ('DISCH', 'POP'), ('SUBST', 'DEFAULT_CONSUMPTION')
                           ]

    name = 'Household Emission Model'
```

- The class `HouseholdEmission` inherits from the `Model` class.
- The `_required_parameters` lists the required parameters: POP from the discharge data and the `DEFAULT_CONSUMPTION` from the substance data. Since the emission model is used for all model modes, the mode tuple is set to *(None, None, None)*, according to 3.2.1.
- The name of the model is specified.

## 4.2.2 Setup

```
def setup(self, segmentID):
    """Set up Household Emission Model."""

    # Shortcuts
    st_params = self.st_params
    data = self.session.data
    ID = segmentID

    # Some parameter which only have to be processed once
    self.ID = segmentID
    self.Pop = float(eval_local_setting(data,
                                       ('SESS_DISCH', ID, 'pop'),
                                       ('DISCH', ID, 'pop')
                                       )[0])

    # Potentially distributed parameters
    Consumption, status = eval_local_setting( data,
                                              ('EMISSION', ID, 'consumption'),
                                              ('SUBST', 'DEFAULT_CONSUMPTION', 'value')
                                              )
    self.Consumption = Parameter(('SUBST', 'DEFAULT_CONSUMPTION'),
                                Consumption,
                                eval_local_stochastics(status, st_params),
                                data['STOCHASTICS'])
```

- Some shortcuts to make the code more transparent. `self.st_params` and `self.session` have been initialized on instantiation of the model object by the graph initialization (see 2.2).
- The segment ID is available on setup and is stored for later referential use.
- The population is not considered as distributed, hence the value is stored directly under `(self.Pop)`:

- Since the default value (from the catchment data, discharge table) can be overwritten by a value entered in the emission dialogs, first evaluate if a local setting is present.
  - According to 3.2.4 `eval_local_setting` returns a tuple of value and a status. Here we need only the value, so take the value out of the tuple.
  - The value is provided as a String, for later calculations we need a number, so cast the value to a Float.
- Evaluate a potential local setting and store consumption value and status.
  - The consumption is potentially distributed, hence we have to instantiate a **Parameter** object:
    - The (BlockID, FieldID) tuple identifies the series of Monte Carlo shots.
    - Consumption is the value (or pair of descriptives) derived above.
    - `eval_local_stochastics` checks if the default consumption is overwritten by a local value. In case, the function returns *None* (see 3.2.4), which effectively switches of stochastics for this objects local consumption. Local values are fixed by definition.
    - `data` provides the dictionary of series of Monte Carlo shots.

### 4.2.3 Compute

```
def compute( self, segment, shot_num ):
    """Compute Household Emission Model."""

    Pop          = self.Pop
    Consumption = self.Consumption.get(shot_num)

    PHIout = Pop * Consumption / 31536000.0          # [kg/s]

    return PHIout
```

- Pop is a static value.
- For the consumption the value for the current Monte Carlo shot (identified by `shot_num`) has to be obtained from the consumption parameter object.
- Since consumption is provided as  $[kg/(capita \cdot year)]$  a unit conversion has to be applied.
- The mass flux out of the segment has to be returned.

A compact version of the `compute` could be also:

```
def compute( self, segment, shot_num ):
    """Compute Household Emission Model."""

    return self.Pop * self.Consumption.get(shot_num) / 31536000.0
```

## 4.3 River Mode 2

While the Household Emission model introduced the very basic concepts of the model implementation in the GREAT-ER II Model Suite, the River model mode 2 presents further items like results or estimate functions.

This section focus on the most important parts, the complete implementation can be found in the GREAT-ER Model source codes under `lastpageGreaterModel/Worker/rivermodels.py`

### 4.3.1 General

```
class RiverModel2( RiverModel1 ):
    """Implements GREAT-ER River Model Mode 2."""

    _required_parameters = [('MOD', 'MODE_RIVER', '2'),
                            ('STRETCH', 'Q_VAL'), ('STRETCH', 'Q_DIST'),
                            ('STRETCH', 'V_VAL'), ('STRETCH', 'V_DIST'),
                            ('STRETCH', 'L'), ('STRETCH', 'STRETCH_CLASS_ID'),
                            ('STRETCH_CLASS', 'SS_VAL'), ('STRETCH_CLASS', 'SS_DIST'),
                            ('ENV', 'SS_CORR'),
                            ('STRETCH_CLASS', 'F_OC'),
                            ('SUBST', 'K_DEG'), ('SUBST', 'K_SED'), ('SUBST', 'K_VOL'),
                            ('SUBST', 'KD_RIVER'), ('SUBST', 'K_OC')
                            ]

    _correlated_parameters = [ (('STRETCH', 'V_VAL'), ('STRETCH', 'Q_VAL'), 1.0),
                               (('STRETCH_CLASS', 'SS_VAL'),
                                ('STRETCH', 'Q_VAL'), ('ENV', 'SS_CORR')) ]

    name = 'River Model, Mode 2'
```

- The `RiverModel2` inherits from the `RiverModel1`.
- The mode tuple identifies the parameter and the value on which the application of the River model mode 2 is decided.
- The `_correlated_parameters` lists the correlation of flow velocity with the volumne flow (fixed) and the suspended solids concentration with the flow (specified by correlation factor as an environmental parameter)

### 4.3.2 Missing Parameters

The River model mode 2 provides an estimation function for the  $K_d$ :  $K_d = K_{OC} \cdot f_{OC}$ , if  $K_d$  is not present. Therefore the `MissingParameter` method has to be extended to take this into account.

```
def MissingParameters(self, parameter_list = None):
    """Implement River Model, Mode 2 specific Missing Parameter Check."""

    parameters = []

    if parameter_list is None:
        parameter_list = self._required_parameters

    # First perform the default analysis
    parameters = RiverModel1.MissingParameters(self, parameter_list)
```

```
# Second check the specifics, if the model is currently selected:
if (eval_parameter(data, ('MOD', 'MODE_RIVER', 'value')) == '2':
    # We have an estimate function for  $K_d$ river =  $f_{oc} * K_{ow}$ 
    # I.e. if  $K_{ow}$  is well defined, don't force  $KD\_RIVER$ .
    #  $f_{oc}$  as a river class parameter must be present by definition.
    if eval_parameter(self.session.data, ('SUBST', 'K_OC', 'value')):
        remove_parameter(parameters, ('SUBST', 'KD_RIVER'))
    # In any other case,  $KD\_RIVER$  is well defined or already in the
    # list of missing parameters, so we don't need do force  $K_{OC}$ .
    else:
        remove_parameter(parameters, ('SUBST', 'K_OC'))

return parameters
```

- The method header and preparatory steps have to be equal with the basic method.
- First perform the default analysis by calling the parent class `MissingParameters` method with the current classes `_required_parameters` list.
- Second perform the analysis on the parameters affected by the estimate function:
  - Ensure we are in the correct River model mode.
  - If the  $K_{OC}$  is well defined in the data, the  $K_d$  can be estimated so try to delete it from the list of missing parameters (`remove_parameter`, 3.2.4).
  - If the  $K_{OC}$  is not present, the  $K_d$  must be defined for a complete parameter set. Either it is or it is already in the list of missing parameters. In any case, the  $K_{OC}$  is not mandatory and can be removed from the list of missing parameters.
- Finally the derived list has to be returned.

### 4.3.3 Setup

```
def setup(self, segmentID):
    """Set up River Model, Mode 2."""

    # Instantiate the Results
    self.Results['CSIMSTART'] = StochasticResult()
    self.Results['CSIMEND'] = StochasticResult()
    self.Results['CSIMINTERN'] = StochasticResult()

[...]
```

```
# Some parameters which only have to be processed once
self.L = atof(data['STRETCH'][ID].l)

if eval_parameter(data, ('SUBST', 'KD_RIVER', value):
    self.kd = atof(data['SUBST']['KD_RIVER'].value)
else:
    f_oc = atof(data['STRETCH_CLASS'][self.ClassID].f_oc)
    k_oc = atof(data['SUBST']['K_OC'].value)
    self.kd = f_oc * k_oc

[...]
```

- The River model calculates three different results, the according **StochasticResult** objects are instantiated during model setup and stored in the **Results** dictionary.
- Object specific data (like the stretch length) are accessible within the data by table name, record identifier and field name.
- If the  $K_d$  is present, make use of it. Else  $f_{OC}$  and  $K_{OC}$  are present<sup>1</sup> so apply the estimate function.
- The rest of the setup is similar to the Household Emission model.

### 4.3.4 Compute

```
def compute( self, segment, shot_num ):
    """Compute GREAT-ER River Model Mode 2 for segment."""

    data = self.session.data
    # Get the flux input
    PHI = segment.PHIin()

    # Get the parameters from the data
    Q = self.Q.get(shot_num)
    v = self.v.get(shot_num)
    L = self.L
    SS = self.SS.get(shot_num)

    k_deg = self.k_deg.get(shot_num)
    k_sed = self.k_deg.get(shot_num)
    k_vol = self.k_deg.get(shot_num)

    f_d = 1.0 / (1.0 + 1.0e-6 * self.kd * SS)
    f_s = 1.0 - f_d

    # Avoid division by zero
    if v > 0.0 and Q > 0.0 and L > 0.0:
        # Hydraulic Residence time in stretch [h]
        if segment.type == model.RiverSegmentType['LAKE']:
            V = L          # In case of a lake the length stores the
            HRT=L/(Q*3600) # volume by definition.
        else:
            V = Q * L / v
            HRT = L/(v*3600.0)

        # 1st order river elimination process
        k = k_deg + f_s * k_sed + f_d * k_vol
        exp_term = exp( - k * HRT )
        PHIout = self.first_order_elimination(PHI, Q, v, HRT, L,
                                             k, exp_term)

    else:
        # flux output
        PHIout = PHI

    # Collect different concentrations
    self.Results['CSIMSTART'].add( 0.0 )
    self.Results['CSIMEND'].add( 0.0 )
```

<sup>1</sup>Before actually starting a simulation a final analysis on parameter completeness is performed. Hence we can be sure that  $K_{OC}$  is defined if  $K_d$  is not.

```
self.Results['CSIMINTERN'].add( 0.0 )  
return PHIout
```

- The mass flux into the segment is provided by the segments method `PHIin`. The model implementation itself has not to take care about the graph structure.
- For better transparency a couple of shortcuts are used for the Parameter objects values.
- Fractioning values are depending on potentially distributed parameters and there-with have to be computed per Monte Carlo shot.
- Check all values used as quotient to avoid a division by zero. If a division by zero would occur, consider the stretch segment as virtual, store invalid results and hand through the mass flux income.
- Else, calculate the hydraulic residence time, the first order elimination rate and perform the first order elimination

### 4.3.5 First order elimination

For completeness we show in addition to the River model mode 2 the first order elimination as implemented as method of the River model mode 1.

```
def first_order_elimination(self, PHI, Q, v, HRT, L, k, exp_term):  
    """Generic first order elimination for rivers."""  
  
    PHIout = exp_term * PHI  
  
    if k != 0.0:  
        self.Results['CSIMINTERN'].add( PHI * (1.0 - exp_term)  
                                         / ( Q * k * HRT ) )  
    else:  
        self.Results['CSIMINTERN'].add( PHI / Q )  
  
    # Collect start and end concentrations  
    self.Results['CSIMSTART'].add( PHI / Q )  
    self.Results['CSIMEND'].add( PHIout / Q )  
  
    return PHIout
```

The first order model ( $\Phi_{out} = \Phi_{in} \cdot e^{-kHRT}$ ) is straight forward, mainly the calculation of the  $C_{sim, internal}$  is of some complexity. Note, that the checks in the river models compute method in combination with checks on the elimination rate here ensures to prevent a division by zero.



# A GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”,

below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format,  $\LaTeX$  input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the

title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section A.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the

rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections A and A above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
- Include an unaltered copy of this License.
- Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified

Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section A above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

## COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section A is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section A. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section A) to Preserve its Title (section A) will typically require changing the actual title.

## TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.