Object Modeling System v3.0

Developer and User Handbook

Olaf David

Jack R. Carlson

George H. Leavesley

James C. Ascough II

Frank W. Geter

Kenneth W. Rojas

Lajpat R. Ahuja

Object Modeling System v3.0: Developer and User Handbook

Olaf David
Jack R. Carlson
George H. Leavesley
James C. Ascough II
Frank W. Geter
Kenneth W. Rojas
Lajpat R. Ahuja

Publication date Aug 10, 2010

Abstract

OMS is a framework for ago-environmental model development, data provisioning, testing, validation, and deployment. It provides a bridge for transferring technology from the research organization to the program delivery agency. The framework provides a consistent and efficient way to create science components, build models, calibrate and test them, modify and adjust as the science advances, and re-purpose for emerging customer requirements. OMS was first released in 2004, and subsequently integrated into the USDA Natural Resources Conservation Service technical architecture in 2008.

This Handbook targets the model developer, integrator, and user. It provides a comprehensive overview and reference about developing, integrating, running and deploying environmental simulation models using this framework. It explains the steps required to create its fundamental building blocks, the components, and integrate them into a more complex model. Model execution is demonstrated for testing, calibration, and uncertainty analysis, illustrating the flexible options available once the model is within the framework.

With OMS 3.0, framework invasiveness to the model has been minimized, improving portability, adaptability and infrastructure integration. The emphasis with this release was focused on (i) simplifying component integration, (ii) implicit auto-scaling of simulation models in multi-core and multi-processor environments, (iii) providing for modeling simulation traceability and integrity, and (iv) auto-documentation of models and simulations.

An annotation standard was developed for those features. That standard promotes straightforward integration of simulation code written in various programming languages with a low impact on the developer. The framework was also designed with strong support for multi-threading, the concurrent processing of multiple simulation tasks. The modeler and integrator does not have to know parallel programming knowledge is required to build parallelized models. OMS 3 enables seamless transitioning from multi-core modeling on a desktop to a clustered processors in a computing cloud.

The framework addresses traceability requirements of agencies with program tracking and financial management responsibilities. Agencies running simulation models can leverage OMS to create auditable simulation trails, based on Secure Hash Algorithms, a Federal Information Processing (FIP) Standard. A related features is the capability to auto-document a model and simulation structure into an open document standard such as Docbook5+.

In summary, the OMS framework supports a work flow to develop and deliver agro-environmental models to user organizations with the following primary steps: (1) develop components, (2) integrate components into models, (3) develop simulations, and (4) deploy and run simulations. The final step of running simulations is integrated into the business work flow of the user organization.

Introduction

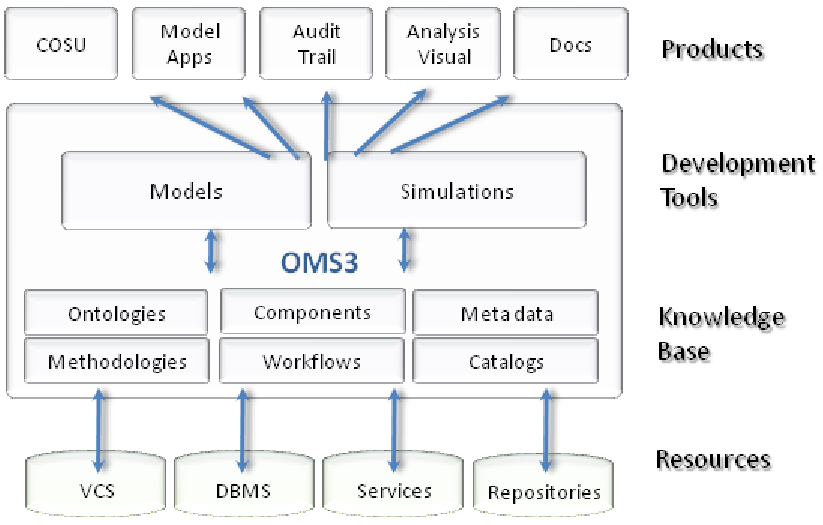
The Object Modeling System (OMS) is an integrated environmental modeling framework. Frameworks in general are helping to create and use software applications by providing common and reoccurring functionality. Web frameworks, user interface frameworks, or frameworks for database integration are examples for successful framework implementations within general software applications. They deal with complicated, mostly infrastructure aspects by abstracting it to a level that is appealing for the non-specialist.

What characterizes **modeling** frameworks? They are expected to support the modeling process, for example, straightforward model code development, seamless model access to data, and data analysis and visualization. Some modeling frameworks also focus on high performance computing and are specifically tailored for particular modeling domains such as climate modeling.

Driving forces for framework adoption within the modeling community are (1) saving time and reducing costs, (2) providing quality assurance and control, (3) re-purposing model solutions for new business needs, (4) ensuring consistency and traceability of model results, and (5) mastering computing scalability to solve complex modeling problems. At the bottom line the model developer should be able to efficiently develop and deliver a simulation model. As pointed out in [Rizzoli2005], the modeler should experience an immediate return on investment by adopting a framework designed to increase modeling productivity.

The Object Modeling System (OMS) is a integrated modeling framework designed to support the delivery of science relating to agricultural and natural resource management in programs administered by the U.S. Department of Agriculture (USDA). The OMS architecture has been designed so that it can inter operate with other frameworks supporting agro-environmental modeling in Europe, Australia, North America, and elsewhere. Its principle architecture is shown in Figure 1, “OMS3 Principle Architecture”. It very much assembles the generic architecture for environmental integrated modeling frameworks as described in [Rizzoli2008].

Figure 1. OMS3 Principle Architecture



There are four foundations identified for OMS3 (Figure 1, “OMS3 Principle Architecture”): **modeling resources**, the system **knowledge base, development tools**, and the **modeling products**. OMS3 core consists of an internal knowledge base and development tools for model and simulation creations. The system derives information out of various modeling resources, such as data bases, services, version control systems, or other repositories, transforms it into a framework knowledge bases that the OMS3 development tools use to create modeling products. Products include model applications; simulations that support calibration, optimization, and parameter sensitivity analysis; output analyses; audit trails; and documentation

Implementing this architecture may require a commitment to a structured model and simulation development process, such as the use of a *version control system* for model source code management, or a simulation run database database to store audit trails. Such features are important for institutionalized implementation of the framework, however, a single modeler may not be required to adhere to it.

1. Basic Concepts

There are a few simple concepts to master in order to create and use models in OMS3. Like other modeling frameworks such as OpenMI [Gregersen2007], CCA [Bernholdt2006], ESMF [Collins2005], and CMP [Moore2007], OMS3 adheres to the notion of **objects** as the fundamental building blocks for a model and to the principles of component based software engineering for the model development process.

Component-based software engineering (CBSE) has existed in one form or another for a number of years. The advantages of constructing *modular* software are well known within the software community. Modularity is a general concept which applies to the development of software in a fashion which allows individual modules to be developed, often with a standardized interface to allow modules to communicate. In fact, the kind of *separation of concerns* between objects in an object-oriented language is much the same concept as for modules, but on a larger scale. Typically, partitioning a system into modules helps minimize coupling, which should lead to 'easier-to-maintain' code.

Initially, software components were viewed almost exclusively as source code modules. In recent years, however, the popular use of the term software component has been with reference to so-called “binary” components. Binary component are individual software artifacts that exist in compiled form, and are typically ready for distribution. A wide variety of technologies have been developed to support the packaging of binary components.

OMS3 as a framework is object-oriented, the models within the framework are objects or components as understood in CBSE. However the design of OMS3 is unique in one important aspect. It is considered *non-invasive* and sees models and components as plain objects with meta data by means of annotations. Modelers do not have to learn an extensive object-oriented Application Programming Interface (API), nor do they have to comprehend complex design patterns. Instead OMS3 plain objects are perfect fits as modeling components as long as they communicate the location of their (i) processing logic, and (ii) data flow. Annotations do this in a descriptive, non-invasive way. Non-invasive lightweight frameworks principles based on plain objects have proven successful in other application fields [Richardson2006] and are expected to pay dividends for agro-environmental modeling.

Why is a non-invasive approach important?

- Most agro-environmental modelers, at least early in the development life cycle, are natural resource scientists with experience in programming (often self-taught), but not software architecture and design. Most modeling projects do not have the luxury employing an experienced software engineer or computer scientist. Software engineers understand and apply complex design patterns, UML diagrams, advanced object-oriented techniques such as parameterized types, or higher level data structures and composition. A hydrologist or other natural resource scientist usually lacks these skills.

  The targeted use of object-oriented analysis and design principles for modeling could be productive for a specific model having limited expectation for reuse and extensibility. However, for a framework, the extensive use of object-oriented features for models is questionable since it puts an undesirable burden on the scientist.

- The agro-environmental modeling community maintains a large number of legacy models. Some methods and equations still in use were developed as long as 60 years ago. What has changed and will continue to change is the infrastructure around them that delivers the output from these models. Smart phones and computing clouds are emerging technologies, to which lightweight, non-invasive frameworks can easily adapt.

- A lightweight framework adjusts to an existing design as opposed to define its own specification or API. The learning curve small, as there is no complex API to learn or new data types to manage. This has some very practical implications for a modeler, since there is no major paradigm shift in using existing modeling code and libraries. Committing to a non-invasive framework is more likely than for the more heavyweight counterparts, since the component integrated in the lightweight framework can still have a 'life' outside on other platforms, and can keep evolving there.

Object-oriented techniques were promoted over the last decade to be the solution for natural resource modeling. Well designed object-oriented models are hard to design if the expected outcome is expected to contain all the promises of reuse, extensibility, and flexibility. Design of complex systems requires experience, anticipation of future use cases, freedom to discard a dead-end design, and most of all: time and resources. In addition environmental modeling perspectives, concepts, and approaches vary, which is hard to capture in a single object-oriented design.

Since OMS3 is a non-invasive modeling framework, the modeler does not need an extensive knowledge of object-oriented principles to make the model-framework integration happen. Creating a modeling object is very easy. There are no interfaces to implement, no classes to extend and polymorphic methods to overwrite, no framework-specific data types to replace common native language data types[1] etc. Instead OMS3 uses meta data by means of Annotations to specify and describe "points of interest" for existing data fields and methods for the framework. Annotations are explained in detail in Appendix B. Chapters 2, 3, and 4 show their use within a model. Within a modeling object any complex internal object-oriented design can be used as needed, however, the framework does not depend on any object-oriented contract.

1.1. Model Components

Components are the main building blocks of simulation models in OMS. Traditionally, scientific applications are designed as large blocks as hand-crafted code, which usually results in a monolithic application. Such model applications are not designed to have parts of them easily re-purposed if a related application will be required in the future. A major disadvantage of building monolithic simulation models is that conceptual boundaries within the model are not captured or not there at all.

In this handbook we refer to a **Component** as a modeling entity, that implements one conceptual modeling concept, is implemented as a plain (Java) object that comes along with annotations. A component can be hierarchical, it may contains other, finer grained components contributing to the larger goal. It is a black-box that exposes its framework relevant aspects via meta data.

The component represents a sufficient level of complexity so that someone can use it in an executable simulation is called a **Model**. Therefor each component can become a model. The model is usually the top level component within a component hierarchy.

Figure 2. Principle Component Structure

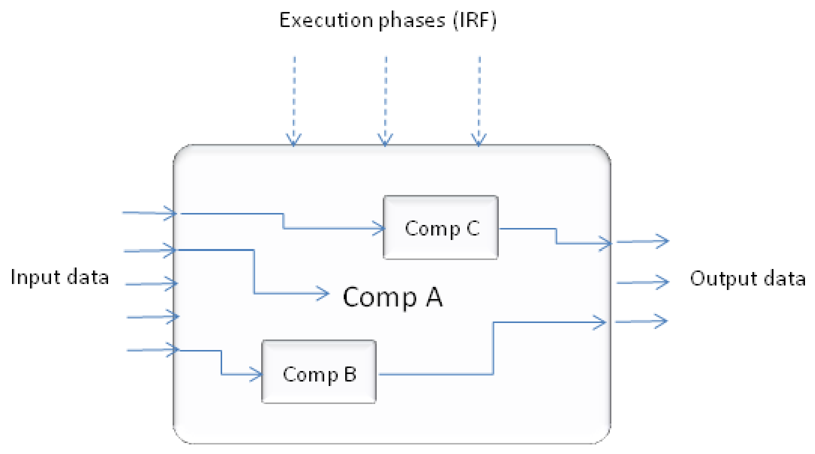


Figure 2, "Principle Component Structure" shows the principle design of a component in OMS3. Like all modeling frameworks OMS3 provides for the same features: Isolating a computational aspect in a component, facilitating

[1]It was widely observed in framework implementation including OMS 2.x and before, that component interoperability can only achieved by using framework supplied data types that usually are redundant equivalences to native language types. Such a requirement leads to even more undesired coupling between the model and application and results in a framework lock-in.

directed data flow (slots, or exchange items) and manage various execution phases of a component. As pointed out in [Peckham2008], an "Initialize #Run #Finalize" (IRF) is a common life cycle for almost all simulation models. OMS3 offers only annotations to allow a user to specify those framework required aspects on top of plain object.

The term **Component** refers to a concept in software development which extends the re usability of code from the source level to the executable. Components are software units that are context-independent both in the conceptual and technical domain. A component is a self contained software unit that is well separated from its environment.

The component approach takes object-oriented designs to the next level. While object-oriented design methods focus on abstraction, encapsulation, and localization of data and methods, they can also lead to simulation systems where objects are highly co-dependent. To remove this limitation, a more structured process was introduced in model development and applied to OMS3: component-oriented modeling. It emphasizes the component as object-oriented software which can be developed and tested independently, and delivered as a unit. It provides its services through well defined interfaces.

Component implementations in general have proven to support reuse in a more efficient way than just using object-oriented methods. There are some general benefits of using components for building complex systems.

- Components are designed with a **standard, well defined interface** in mind. Such a published interface hides the implementation of the component logic and forces an abstraction level which separates the offered contract for communication from its implementation.
- Components are **self contained units** from the conceptual and technical perspective. They can be developed and tested individually. Finally they are packaged and are delivered to be used in several applications.
- The use of components **simplifies the construction of complex systems** since they change the way these are built. As opposed to programming the entire model, it can be composed using components. Some parts of a model may originate from legacy code sources, and some may be new code. A component approach facilitates the integration of legacy and new code.

The use of components helps face the challenge building complex simulation models by reducing model software complexity and overcome the limitations of monolithic, highly coupled model implementations.

1.1.1. Model Component Base

OMS3 also introduces the concept of a model base, considered to contain two or more instances of a model designed to address the modeling needs of a customer. For example, the customer may require a model apply to business scenarios across diverse regions. Therefore the solution could involve several instances of the model, each modified, configured, and validated for a particular region.

Likewise, the customer may require a model be used in different contexts within a business function, and the solution may involve model instances that fit into the various business work flows. Knowing these requirements up front is important for architecting the model and deploying as instances aligned with business needs. The model instances should be managed as a model base. Model base components (including the model and its instances) should be managed within an OMS3 modeling project and stored in the OMS3 Component Library.

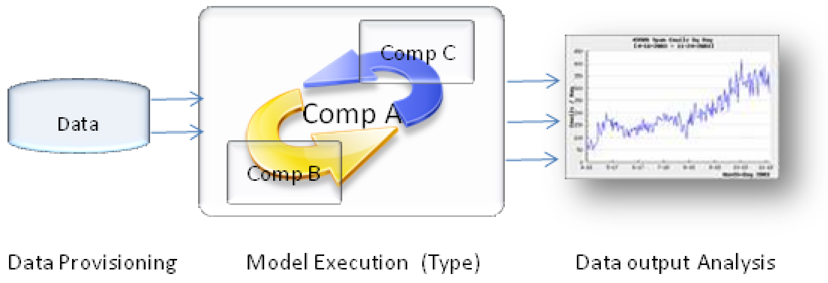
1.2. Simulations

The other fundamental concept in OMS3 are **Simulations**. Simulations are giving a model a purpose, they are the model applications. Usually a simulation consists of the

- **Model** that is to be used which is in most cases the top level component.
- **Location and time sensitive input data** such as parameter files or climate data sets
- **Output management** and analysis such as model efficiencies, statistical summaries, graphs, and plots.
- **Type of execution**, such as single execution, parameter sampling execution, an uncertainty analysis, or a forecast execution using synthetic input, etc.
- **Execution environment** such as the local computer, a remote box, or a cluster of computer at a remote location.

OMS3 defines a set of various simulation types that are discussed in Chapter ???. Figure 3, “Principle Simulation” shows the concept of a simulation.

Figure 3. Principle Simulation



How to create a simulation? OMS3 employs a simple but powerful concept called **Domain Specific Language** (DSL) to provide for a concise, robust, and flexible representation of simulations. A DSL in general is a mini-language aiming at representing constructs for a given domain. Our domain is modeling, therefore the vocabulary of this DSL very much reflects modeling concepts.A DSL is really a language extension dealing with a design pattern such as building a hierarchy of objects in a simple, descriptive way.

With a DSL, simulations can be created and executed from different tools such as IDEs, the OMS3 modeling console, the command line, or any application that embeds a OMS3 runtime.

The use of DSLs over other approaches such as XML for a structured setup of simulations and meta data has many advantages. XML tends to be verbose when supported by a schema, parsing XML does not account for programming languages data types, or conversions. DSLs instead offer an elegant and concise construction of simulations while providing for implicit data types. The specification of the modeling DSL elements is part of this handbook.

2. Audience

Modeling in general is a complex process that refers to many activities such as data management, coding, output analysis, etc. This defines the type of use for a framework such as OMS3. Scientist may developing an operational model with a software engineer, a technician is preparing data to be used in the simulation by changing its format, a technical service provider might perform model calibrations to obtain a fitting parameter set, or a stakeholder is using a canned model simulation to get some answers to a problem. Also, a third party company might want to include the framework into their own application suite and needs to integrate it very low level.

OMS3 can support many of those common activities. The table below shows different kinds of audiences specific in context to OMS as a framework supporting various modeling aspects. Other classifications exist [Rizzoli2008], Good Modeling practice).

Table 1. Framework Audience

Framework User	Description	Expertise	Activities	Product	Example
System Integrator	integrates a model or modeling framework into	IT Infrastructure architectures, system design,	Integrating the framework into large scale ap-	e.g. a web service that employs the framework	software engineer, system integrator

Framework User	Description	Expertise	Activities	Product	Example
	a larger business application	OMS3 Framework integration API	plication using IDEs		
Model Component Developer	developing software code	OMS3 Annotations and component integration API	Coding using an IDE and related build/test and version control tools	Model/component executable	Research scientist, software engineer, Domain expert
Simulation Developer	setting up a simulation for a specific use	OMS simulation definitions, Model parameterization, data formats,	Using simulation definitions for data provisioning and setup, calibration, uncertainty analysis and optimization	A runnable simulation (e.g. calibrated parameter and data set) for a given area and problem.	Research scientist, Technical Service Provider,
Simulation User	using a simulation to get an answer to a problem	knowledge about the proper use of a simulation within a provided domain context	Using the simulation with an application and selecting data sets / parameter to be used	An quantitative answer to a problem	Consultant, Farmer, Manager

The are most likely overlaps of activities, since a framework user is usually involved in many activities.

3. OMS 3.0 Feature Summary

This handbook describes the most recent incarnation of OMS version 3.0 (referred as OMS3). This version differs from previous versions in that the support for framework-based model development has widened in several respects. The modifications and enhancements respond to experience from the user community.

What are the summarized features, improvements and characteristics of OMS3?

1. OMS modeling is **component-based**.

   It aims for only minimal requirements to call a plain Java object a OMS3 component. Existing legacy classes are allowed to keep their identity, which means that once a component has been introduces into OMS3 it is still usable outside of OMS3. OMS 3.0 is **non-invasive**. It minimizes the burden on a component/model developer to build code into the framework by not imposing an API. There is almost no learning curve as existing Java/Fortran/C/C++ code does not have to be changed. The modeler does not have to learn and use framework data types, and does not have to comprehend communication patterns to parallelize the model. Most if not all previous modeling frameworks have contained these intrusive features, and OMS 3.0 represents a fundamental step to remove them.

2. OMS is based on the **Java platform**.

   However it is inter operable with C,C++, and FORTRAN on all major operating systems and architectures. **Language interoperability**. OMS 3.0 moves from a source centric Java Native Interface (JNI) strategy focused on FORTRAN to a DLL centric Java Native Access (JNA) integration that now supports all versions of FORTRAN, C, and C++ on all major architectures in 32 and 64 bit. FORTRAN and C/C++ programmers can continue to use their respective tools to create components and then use one of the Java IDEs to annotate and assemble components into a model, create simulations for testing and validation, and package model instances for deployment.

3. Components always execute **multi-threaded**.

The default execution is multi-threaded. Sequential execution is just a specific case of multi-threaded execution where the data flow requires the sequential execution of components. If data flow allows it, components are being executed in parallel. No explicit thread coding is needed to make this happen. OMS models are **data flow driven**. The execution of components is driven by data flow dependencies. There is no explicit/manual control of an execution sequence of components.

4. Simulations as defined as **DSLs**.

   OMS 3.0 adds a new tool set for model calibration, sensitivity and uncertainty analysis (such as GLUE, SCE, MOCOM, and others) to the package suite. A consistent user experience in simulations is provided also for existing types such as Ensemble Streamflow Prediction (ESP) and LUCA calibration.

5. **Runtime flexibility** for simulation execution

   Models can now be executed in different environments that scale from a notebook to a computing cluster or even a cloud such as Amazon's Elastic Computing Cloud (EC2).

6. Wide range of **System Integration** options and **development flexibility**.

   OMS3 can be integrated into almost any infrastructure at different framework integration levels. Models can now be developed with any Integrated Development Environment (IDE), that supports at least Java. The constraint to use the custom OMS IDE platform (with version 2.2) has been removed. Models can now be developed using all major Java IDEs such as Netbeans, Eclipse, or IntelliJ.

How does is compare to its previous version OMS 2.2? It is easier to integrate OMS3 into an existing development process, it scales into multi-core, multiprocessor environments, it makes model development easy by its non-invasive nature. It lowers the burden for a modeler to take advantage of a modeling framework.

4. "Hello Modeler" - A first example

Let's create a very simple model that demonstrates all the basics involved with OMS3. It simulates nothing numerically (we get to this later), it is rather the modeler's variant of the usual "`Hello World`" example, which we will call "Hello Modeler." Later, the examples will become more comprehensive. This example shows the two fundamental parts of modeling in OMS: Creating a

1. **component and therefor a model** and the executable code, and

2. **a simulation**, combining the model with input data sets and performing a simulation.

4.1. A First Model Component

For the Java modelers components are developed as Java classes (For FORTRAN and C/C++ modelers, Section ???).

We create a new Java class named `Component` and save it as `Component.java`. The source file needs to be compiled with `javac` using an IDE or just the command line compiler. Section ??? explains the details for setting up a development environment and the `classpath`. Now, we have created the model source code and have compiled it.

As seen here, a component is a piece of plain Java code that is annotated with OMS specific meta data. Meta data annotations (shown in bold) are a very easy to add and turn plain objects into components to be used within the modeling framework.

Example 1. First Component (Component.java)

```
package helloworld;

import oms3.annotations.*;              // 1.

/**
 * First component
 */
public class Component {

   @Role(Role.PARAMETER)                // 2.
   @In public String message;           // 3.

   @Execute                             // 4.
   public void run() {
       System.out.println(message);
   }
}
```

1. Adding the `oms3.annotation` package to the component associates it with the OMS framework. This package defines OMS annotation types.

2. The `@Role` annotation gives its following field declaration the 'Parameter' identity.

3. The OMS `oms3.annotations` package defined `@In` annotation tags the field, that provides input to the component, a message as a plain String in the case above. Note that the field needs to be public like the class.

4. The annotation `@Execute` indicates the entry point for model execution. The run method now gets called when OMS will execute the component. The method name is irrelevant here, the annotation `@Execute` points to it.

The rest of the class is plain Java code. The class has to be public as shown above.

4.2. A First Simulation

As a next step we need to create the simulation to parameterize and run the just created model. Create a text file called `hello.sim` with the following content. We define all the resource elements for the simulation, which are assembled in a hierarchical way.

Example 2. First Simulation (hello.sim)

```
sim(name:"Hello") {                            // 1.
   resource "$work/dist/*.jar"                 // 2.
   model(classname:"helloworld.Component") {   // 3.
      parameter {                              // 4.
            message " Hello Modeler ..."       // 5.
      }
   }
}
```

1. The element `sim` defines a simple simulation that runs a model in a single run. It has the name '`Hello`" as shown. In curly brackets there are all the resources for this simulation.

2. First, the `resource` element lists all files, belonging to this simulation, all `jar` files in the dist folder in the work directory. The value of work might be passed as a system property. The jar files listed here by wild card expansion will be put into the model's `classpath`.

3. The model resource element specifies the class that represents the top level component. The class name is full qualified, it must include the package name. The class `helloworld.Component` was developed in the previous section.

4. The model contains the `parameter`, which should be passed from the simulation to the model on startup. In out example the `@In` field `message` gets the value `" Hello Modeler ..."` as input. The name as listed here in this simulation must exactly match the name of the field in the model.

The `sim` element is a part of the OMS3 DSL for modeling. This DSL provides a simple descriptive syntax for constructing model applications for different model to calibrate parameters, evaluate uncertainty and parameter sensitivity, run model ensembles in a (cloud) cluster, or just simply execute.

With the simulation developed, we are ready to run the model. The OMS3 console provides the easiest way to load the simulation, execute it and look at the result. The following chapters and sections will discuss more options to run a simulation.

Start the console by clicking the Launch button as it provided at

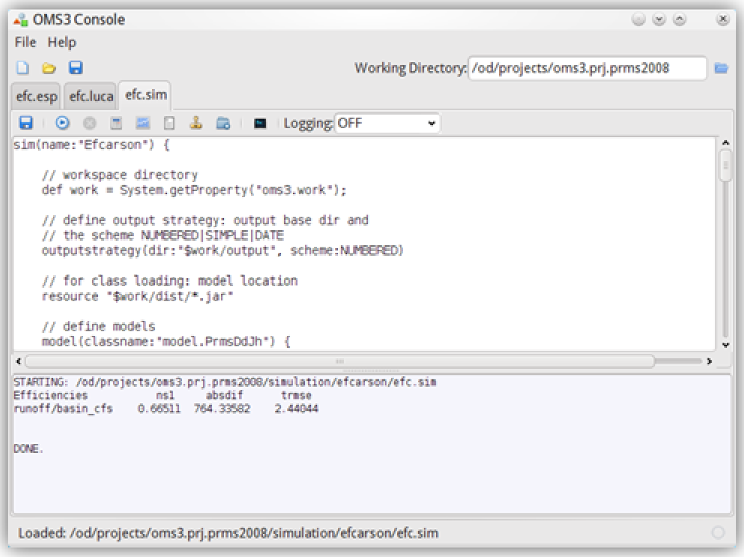
```
http://oms.javaforge.com
```

Note

You need to have some recent Java 1.6 Runtime Environment installed to launch the application. Download it from http://java.sun.com/javase/downloads.

With the console running, open the file `hello.sim` (you can also use it to write and create it in the first place). It shows as tab in the upper part of the screen.

Figure 4. Running the Simulation 'Greeting' using the OMS3 Console



Working Directory: /od/projects/oms3.prj.prms2008

Logging: OFF

```
sim(name:"Efcarson") {

    // workspace directory
    def work = System.getProperty("oms3.work");

    // define output strategy: output base dir and
    // the scheme NUMBERED|SIMPLE|DATE
    outputstrategy(dir:"$work/output", scheme:NUMBERED)

    // for class loading: model location
    resource "$work/dist/*.jar"

    // define models
    model(classname:"model.PrmsDdJh") {
```

```
STARTING: /od/projects/oms3.prj.prms2008/simulation/efcarson/efc.sim
Efficiencies          nsl     absdif      trmse
runoff/basin_cfs    0.66511  764.33582   2.44044


DONE.
```

Loaded: /od/projects/oms3.prj.prms2008/simulation/efcarson/efc.sim

Now hit the Run button to execute the simulation. The expected output appears in the lower part of the screen. The string '`Hello Modeler`' is passed from the simulation to the model component and is printed out at the lower part of the console.

Congratulations, the first model and simulation in OMS3 is implemented.

Bibliography

[Argent2004] Argent, R.M. An overview of model integration for environmental applications—components, frameworks and semantics Environmental Modelling & Software, Volume 19, Issue 3, Pages 219-234, Mar 2004

[Bernholdt2006] Bernholdt D.E. and Allan B.A. and Armstrong R. and Bertrand F. and Chiu K. and Dahlgren T.L. and Damevski K. and Ewasif W.R. and Epperly T.G.W and Govindaraju M. and Katz D.S. and Kohl J.A. and Krishnan M. and Kumfert G. and Larson J.W. and Lefantzi S. and Lewis M.J. and Malony A.D. and McInnes L.C. and Nieplocha J. and Norris B. and Parker S.G. and Ray, J. Shende and S. Windus, T.L. and Zhou, S. "A Component Architecture for High Performance Scientific Computing", Journal of High Performance Computing Applications, ACTS Collection Special Issue, May 2006

[Collins2005] Collins, N., G. Theurich, C. DeLuca, M. Suarez, A. Trayanov, V. Balaji, P. Li, W. Yang, C. Hill, and A. da Silva. Design and Implementation of Components in the Earth System Modeling Framework. International Journal of High Performance Computing Applications, Volume 19, Number 3, pp. 341-350. 2005

[David2002] David O., S.L. Markstrom, K.W. Rojas, L.R. Ahuja and I.W. Schneider, The Object Modeling System. In: L.R. Ahuja, L. Ma and T.A. Howell, Editors, Agricultural System Models in Field Research and Technology Transfer, Lewis Publishers, Boca Raton (2002), pp. 317–330.

[Gregersen2007] Gregersen, J.B., Gijsbers, P.J.A., and Westen, S.J.P., (2007). Open Modelling Interface. Journal of Hydroinformatics, 9 (3), 175-191. 2007

[Moore2007] Moore, A.D., D.P. Holzworth, N.I. Herrmann, N.I. Huth, M.J. Robertson, The Common Modelling Protocol: A hierarchical framework for simulation of agricultural and environmental systems. Agricultural Systems, Volume 95, Issues 1-3, Dec 2007, Pages 37-48

[Peckham2008] Peckham, S: CSDMS Handbook of Concepts and Protocols: A Guide for Code Contributors, http://csdms.colorado.edu/wiki/Help:Tools_CSDMS_Handbook, 2008

[Richardson2006] Richardson, C: POJOs In Action. Manning Publications. Jan 2006

[Rizzoli2008] Rizzoli, A.E., Leavesley, G.H., Ascough II, J.C., Argent, R.M. Athanasiadis, I.N., Brilhante, V.C., Claeys, F.H., David, O., Donatelli, M., Gijsbers, P., Havlik, D., Kassahun, A., Krause, P., Quinn, N.W., Scholten, H., Sojda, R.S., and Villa, F. 2008. Chap. 7: Integrated modelling frameworks for environmental assessment and decision support. In: Environmental Modelling and Software and Decision Support – Developments in Integrated Environmental Assessment (DIEA), Vol. 3, A.J. Jakeman, A.A. Voinov, A.E. Rizzoli, and S.H. Chen (Eds.), pp. 101-118. Elsevier, The Netherlands.

[Rizzoli2005] Rizzoli, A.E., Svensson, M.G.E., Rowe, E.C., Donatelli, M., Muetzelfeldt, R., van der Wal, T., van Evert, F.K., Villa, F.:Modelling Framework (SeamFrame) requirements. SEAMLESS report no. 6, Dec 2005

Chapter 1. Installation and Setup

OMS3 is based on the Java Platform, therefor it is usable in all major operating systems. Examples are Windows, Linux, and MacOS, or OpenSolaris. Any environment supporting Java 1.6 can run OMS3. Different Java versions are available for 32 and 64 bit architectures and processor types. All of those Java versions are suitable to run OMS3 and its models. Please be aware that models using native systems components from a DLL or shared object are not binary compatible across platforms, however pure Java models are.

1. Required Software

Download and install Java 1.6. The JDK 6 (Java SE Development Kit) is required for OMS3 model development, the Java 1.6 Runtime Environment is sufficient. To obtain latest JDK, download it from:

```
http://java.sun.com/javase/downloads
```

Follow the installation instructions for your operating system during installation. Make sure the commands `java` and `javac` are accessible in the PATH when using the command line interface.

2. Install and Setup OMS3

There are different options for installing OMS3 which are very much driven by its use case scenario. The OMS Console allows to run en existing model, but does not support model development like an IDE. However an IDE will give you the option to develop components by editing, compiling and packaging them.

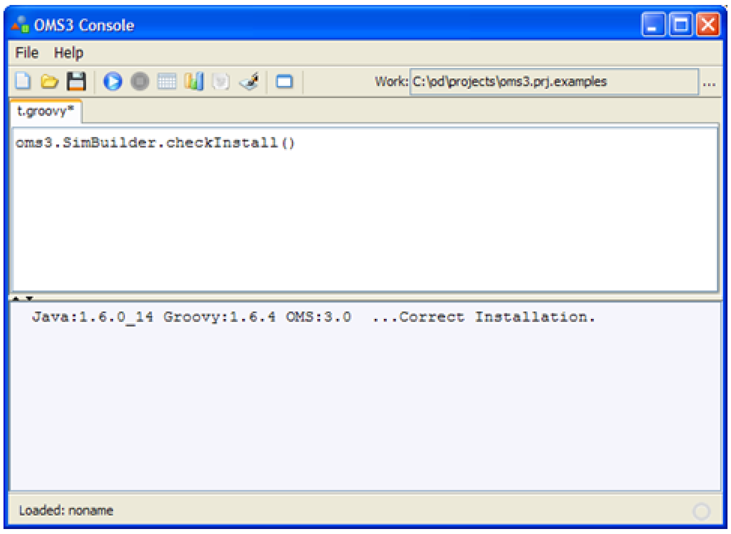
2.1. Modeling Console

Install the JRE or JDK, launch the OMS3 Console Launch from:

```
http://oms.javaforge.com
```

You can test the successful installation of all the packages by executing this line in the OMS 3 Console:

Figure 1.1. Modeling Console



```
oms3.SimBuilder.checkInstall()
```

```
Java:1.6.0_14 Groovy:1.6.4 OMS:3.0  ...Correct Installation.
```

The `oms3.SimBuilder` class has the method `checkInstall()` that verifies the presence and correct version of required software such as Java. In the OMS3 Console, add the same command above and hit `Ctrl+R` (Run).

2.2. Integrated Development Environments (IDE)

You may also download a free Java IDE such as the Netbeans IDE, Eclipse or IntelliJ. Those IDEs might also bundle Groovy language support out of the box (check the IDE, and download Groovy from http://groovy.codehaus.org. Any environment that supports Java at least will do it.

For any IDE configuration, the OMS jar needs to be downloaded. It is available as a zip file, containing the oms library (`oms3-all.jar`), the sources and the API documentation .

Download the OMS3 distribution file `oms-3.x.zip`

```
http://oms.javaforge.com/downloads/oms3
```

To use OMS, you need to add the file `oms3.x-all.jar` to your IDEs `CLASSPATH`. There are different ways to do this depending on the development tool you are using. Please follow the IDE instructions on how to add external libraries to a development project.

2.3. Command Line Interface (CLI)

The OMS3 command line interface allows the execution of a simulation at the command line. It is part of the `oms3-all.jar` and can be easily accessed by executing the jar file itself:

```
$ java -jar oms-all.jar
usage: java -jar oms-all.jar [-r|-e|-d|-a|-s] <simfile>
 CLI access to simulations.
           -r   run the simfile
           -e   edit parameter in simfile
           -d   document the simfile
           -a   run the simfile analysis
           -s   create SHA digest
$
```

Note

The command line executes `oms-all.jar`, located in the same directory. Also, the library `groovy-all-1.6.x.jar` needs to be in the same directory (download the groovy binary distribution from http://groovy.codehaus.org, the `embeddable` directory contains this library) .

To run a simulation use a command such as:

```
$java -jar oms-all.jar -Dwork="/tmp/prms2008" -r efc.sim
```

It is also possible to specify the CLI class directly to account for different versions of the libraries:

```
$java -cp "oms-all.jar;groovy-all-1.6.5.jar" oms3.CLI -r efc.sim
```

By providing the `-D` argument to the Java command line, system properties can be set for the simulation as shown above.

3. References

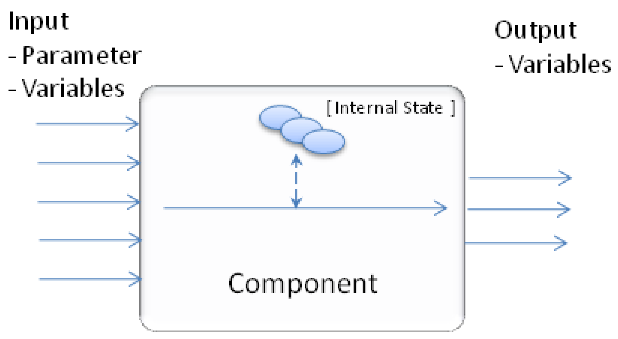
- Java Development Kit version 6 [http://java.sun.com/javase]
- Netbeans IDE [http://netbeans.org]
- Eclipse IDE [http://eclipse.org]

- Groovy language specification and Development Kit [http://groovy.codehaus.org]

Chapter 2. Developing Components

This chapter is designed as a step by step tutorial. It guides a user through the process of creating and testing of components, their integration into more complex components (models), the data provisioning of models, and finally their deployment into various environments.

Figure 2.1. General Component Schematic



A component representing a certain conceptual function in a simulation is the foundation for each model. A hydrological model for example usually needs components for (1) handling input/output, (2) representing processes of a hydrological system such as precipitation, interception and runoff, and (3) realizing general data processing functions such as reading climate data or parameter sets. Ideally, the key to a proper model design based on components is a clean "separation of concerns" in a model. Modeling in OMS and most other frameworks must answer the question: “what qualifies a part of a model to become a component?”

- A component realizes a certain and mostly **one conceptual function** in a model. It represents a physical process, a management action, a data gathering step, or the presentation of results to the user interface. Such functions need to be identified and separated from each other. Each of these will result in a component.

- An identified component **can be fully described** regarding its function, data requirements and data offerings. Therefore, the specification and later implementation of a component will be done with respect to its anticipated simulation context, but a tight dependency to this context should be avoided. Later during development, the component will be tested standalone using a test harness to validate the design specification.

- Components that model important functions and processes that recur across models and modeling projects should be **designed for portability and re-use**. Some additional investment is required to produce this design, but payoff can be substantial. Widely used components would be similar to widely referenced science in technical papers.

- Conceptually, models and sub-models (or modules) are **aggregated components**, and conversely components often are models in some form. Therefore modeling architecture is a question of granularity and depends on the business needs of the customer. The model can be large, designed to address many business requirements comprehensively, or it can be small, a relatively simple service deployed as part of a business application work flow.

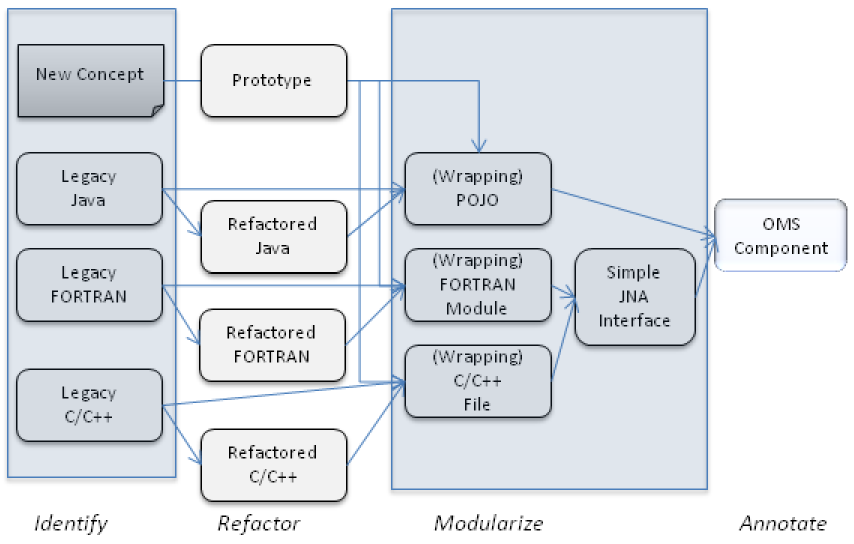
A model also should be architected so that new science can be incorporated through time. This is best achieved by architecting the model to contain well documented components that can be removed and replaced with updated science. This also is a question of granularity. Sometimes it will be necessary to replace a module, other times a simple component. And the design should take into account that models often should be built with a common code base that permits contributions and updates from several modelers.

By analyzing simulation models a classification of potential components can be made.

Scientific components	... Implement methods and equations to estimate some physical phenomenon. Examples would be a component estimating amount of water evaporated from a certain land cover into, a component predicting the soil loss due to wind erosion, etc. Such components usually apply some mathematical function.
Scientific utility components	... Support the analysis of models by providing statistical analysis methods such as descriptive statistics, frequency analysis, etc. Distribution generator components are used to provide data to scientific components.
Control components	... Are responsible for managing the execution of a model. A Runge-Kutta Integration component, a Time management, or a Convergence criteria component are examples for this.
Data Input/Output Components	... Are providing data to other components in a simulation model. Such components could handle data transfer from databases or files to the model. Visualization components like graphs or spreadsheets are also falling under this category.

Components are the basic building blocks of models in OMS. Components can be simple or very complex, depending on the scope and the problem domain. The Figure below shows different paths to create a component the can be used within a OMS model.

Figure 2.2. Component Development



There are 4 identified phases of development that might not all apply in real world. It depends on the existence or quality of the existing code, the skills of the developer, and the available tools to support and automate this process.

The first two phases (Identify and Refactor) focus on the identification and improvement of existing code, where as the two last steps adjust the code to OMS.od/projects/oms3.prj.

1) Identify needed simulation approaches.	In this first step the simulation code should be identified.
2) Refactor existing code or prototype new solutions as needed.	Refactoring refers to the concept of improving the design of existing code without changing its behavior. Refactoring addresses the fact that code is

	(usually) written without extensive reuse outside of its original scope in mind. There are recommendations, methods and tools available to support such a process.
3) Modularize the code.	This step closely follows the previous one but aims at creation of self contained modules or software units. The result the of the process is a generic component free from hidden dependencies.
4) Annotate the code.	Finally the modeler annotates the component, the final step to making it OMS-compliant. Annotation provides the modeler, the modeling team, and external users meta data about the component to aid in the modeling process. Annotations also become important when the component or model is used in customer business applications that require validation and certification of the science.

At the end of this process you have produced a component that can be used within OMS, and other frameworks.

Note

All native components remain intact, and still can be used in their original context (if refactored right).

The next sections will introduce component development of Java components step by step. Section ??? will demonstrate adjusting existing legacy components so that they can be used with OMS.

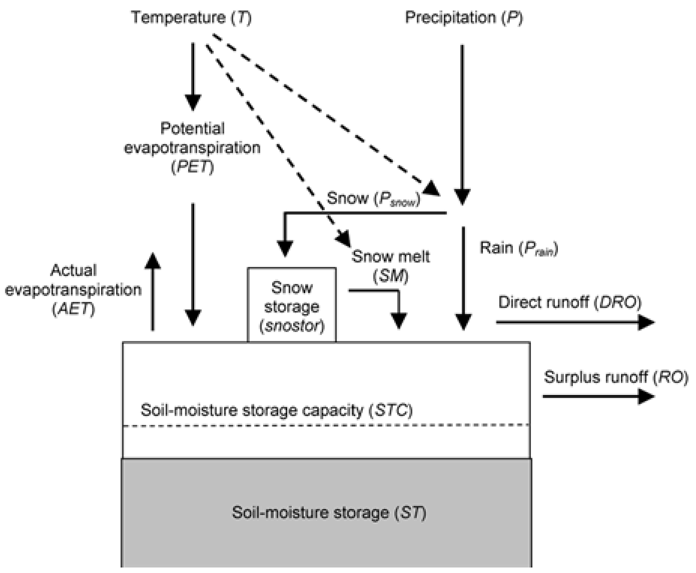
1. Example: Developing Components for a Monthly Water Balance Model

Monthly water-balance models have been used as a means to examine the various components of the hydrological cycle (for example, precipitation, evapotranspiration, and runoff). Such models have been used to estimate the global water balance to develop climate classifications to estimate soil-moisture storage, runoff, and irrigation demand ; and to evaluate the hydrological effects of climate change [McCabe2007].

The model, referred to as the Thornthwaite water-balance program, can be used as a research tool, an assessment tool, and as a tool for classroom instruction.

The water-balance model (Figure 2.3, “Thornthwaite Monthly Water Balance Model Schematic”) analyses the allocation of water among various components of the hydrological system using a monthly accounting procedure based on the methodology originally presented by Thornthwaite [Thornthwaite1948],[Mather1978]. Inputs to the model are mean monthly temperature (T, in degrees Celsius), monthly total precipitation (P, in millimeters), and the latitude (in decimal degrees) of the location of interest. The latitude of the location is used for the computation of day length, which is needed for the computation of potential evapotranspiration (PET). The model is referred to as the Thornthwaite model. A discussion of the individual process components of the water balance can be found in [McCabe2007]

Figure 2.3. Thornthwaite Monthly Water Balance Model Schematic



This model may serve as an example to create an component based implementation. The processes are simple and straightforward, however its structure and general setup is typical for many hydrological and environmental models. The following sections will explain the component creation of based on the processes, their integration into a full model, and finally the provisioning of data, simulation setup and output analysis.

2. Developing a PET Java Component

The very basic building block of a simulation model is a component as stated earlier. It implements usually one purpose within a larger model. The concepts are already stated as physical processes of the model. Hence, monthly PET computation should be implemented as a component.

Monthly PET is estimated from mean monthly temperature (T) and is defined as the water loss from a large, homogeneous, vegetation-covered area that never lacks water [Thornthwaite1948],[Mather1978] . Thus, PET represents the climatic demand for water relative to the available energy. In this water balance, PET is calculated by using the Hamon equation [Hamon1961]:

Equation 2.1. Potential Evapotranspiration (Hamon)

$$\mathrm{PET}_{\mathrm{Hamon}} = 13.97 \times d \times D^2 \times W_t$$

where $\mathrm{PET}_{\mathrm{Hamon}}$ is PET in millimeters per month, d is the number of days in a month, D is the mean monthly hours of daylight in units of 12 hours, and W_t is a saturated water vapor density term:

Equation 2.2. Saturated Water Vapor Density

$$W_t = \frac{4.95 \times e^{0.062 \times T}}{100}$$

where T is the mean monthly temperature in degree Celsius [Hamon1961].

Step 1: Coding the Equation

As a first task the core equation code necessary for the computation of `pet` is created. The equations above are translated into valid JAVA statements. All variables that are either required as input or provided as output are shown in bold.

Example 2.1. Hamon equations (code fragment)

```
...
int[] DAYS = {
     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31        // 1.
};

...
double Wt = 4.95 * Math.exp(0.062 * temp) / 100.0;         // 2.
double D2 = (daylen / 12.0) * (daylen / 12.0);             // 3.
double d = DAYS[month]
double pet = 13.97 * d * D2 * Wt;                          // 4.
...
```

1. An int array holds the number of days per month, used in 3.

2. Compute `Wt`, the `Math` class provides the `exp()` method.

3. Compute `daylen` input in units of 12 hours.

4. The final `pet` calculation.

Note that the data type `double` is a 8-byte floating point data type in Java

The code above is valid Java code implementing the needed equations for Hamon, however is has to be structured as valid JAVA source code.

Step 2: Creating the Class

A JAVA class has to be created, its content is shown below. Use a Java IDE or a text editor to create this source. Note that the class `HamonET` has to be stored in a file `HamonET.java` within the folder `thornthwaie` (package name) according to the Java language specification. Common Java IDE provide very good support for handing those tasks.

Example 2.2. HamonET Class (HamonET.java)

```
package thorthwaite;                                        // 1.

import java.util.Calendar;                                  // 2.
import java.lang.Math;

public class HamonET {                                      // 3.

   static final int[] DAYS = {                              // 4.
       31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   };

   public double temp;                                      // 5.
   public double daylen;
   public Calendar time;

   public double pet;

   public void compute() {                                  // 6.
       if (temp <= -1.0) {
           pet = 0.0;
           return;
       }
       int month = time.get(Calendar.MONTH);                // 7.
       double Wt = 4.95 * Math.exp(0.062 * temp) / 100.;
       double D2 = (daylen / 12.0) * (daylen / 12.0);
       pet = 13.97 * DAYS[month] * D2 * Wt;
       if (pet < 0.0) {
           pet = 0.0;
       }
   }
}
```

1. The package `thornthwaite` provides a 'name space' for this class. A name space is an arbitrary named container providing for related classes. An alternative name could be '`et`' to hold related evapotranspiration classes.

2. The `import` statements are introducing core Java utility classes into HamonET.

3. The `class` declaration defines the HamonET class, which has to be `public`.

4. The `DAYS` array is declared within the class as '`static final`' which makes its content not modifiable and visible for all instances of `HamonET`.

5. The field declaration section of the class lists the Input/Output fields. Note that time is declared as a `Calendar` object, a class provided by the Java API.

6. The method compute encapsulated the Hamon equations. It is declared `public` and `void`, which means no return type. There are also no arguments for this method.

7. The actual month is obtained from the time object. Note that the return value is zero-based, which makes `month` usable as index for the `DAYS` array.

Up to this point we have created a class that is a valid Java class. This is also being referred as a POJO (Plain old Java Object).

Step 3: Creating the OMS3 Component

Just a very little work is required, to turn the `HamonET` Java class into an OMS3 component. For a developer it mainly means that the existing elements that are important for (i) data flow and (ii) code processing as developed in the previous step have to be *annotated*. No other changes are required in this example. Annotations are provided by the OMS3 core library.

Example 2.3. Annotated Component for Execution (HamonET.java)

```
package thornthwaite;

import oms3.annotations.*;                                     // 1.
import java.util.Calendar;
import java.lang.Math;

 public class HamonET {
    // number of days per months
    final static int[] DAYS = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };

    @In public double temp;                                    // 2.
    @In public double daylen;
    @In public Calendar time;

    @Out public double pet;                                    // 3.

    @Execute                                                   // 4.
    public void compute() {
         if (temp <= -1.0) {
            pet = 0.0;
            return;
        }
        int month = time.get(Calendar.MONTH);
        double Wt = 4.95 * Math.exp(0.062 * temp) / 100.;
        double D2 = (daylen / 12.0) * (daylen / 12.0);
        pet = 13.97 * DAYS[month] * D2 * Wt;
        if (pet < 0.0) {
            pet = 0.0;
        }
    }
 }
```

1. Add the import statement for including all OMS3 annotations into this class. The package `oms3.annotations` contains them all. Section ??? discusses them in detail.

2. Annotate the fields which serve as input to all equations in HamonET with `@In`. Each field has to have its own annotation which preceded the field. Note that the field can be declared public, however it is not required.

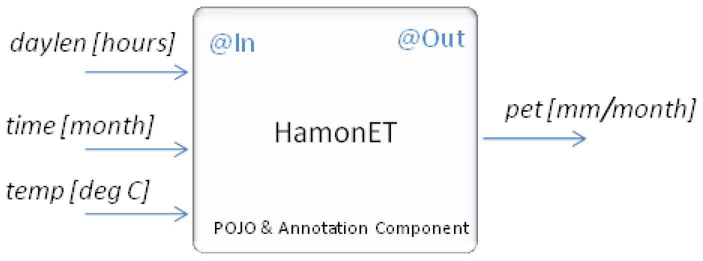
3. Annotate the fields which serve as output to all equations in HamonET with `@Out`.

4. Annotate the main computational method of this class with `@Execute`. As a requirement, this method must have no arguments, must be public, and no arguments.

Now we turned the `HamonET` class into a OMS3 component. It is in fact still a POJO that still can be used as such outside of the framework, as long as the OMS3 annotations are put into the `CLASSPATH`.

The conceptual schematic of the HamonET component shows Figure ???

Figure 2.4. HamonET Component Schematic



The component is now fully functional in OMS3 and ready to go into a model for execution. However, OMS3 offers more annotations that enrich the component for

1. Documentation and archival,

2. Robustness improvement due to data verification, and

3. Later Traceability of simulation results.

The following adds the annotations for those aspects.

Step 4: Improving the OMS3 Component

The `oms3.annotations` package contains a rich set of annotations that are optional for core component execution but should be good practice in order to provide for a well rounded set of context information. Since annotations are being used, this context information is very much attached to the source code. Therefore, it will be always in sync with the source.

Context information can be attached to the (i) whole component and (ii) the `In/Out` fields as shown below.

Example 3.4. Complete Annotated Component (HamonET.java)

```
package thornthwaite;

import oms3.annotations.*;
import java.util.Calendar;
import java.lang.Math;

@Description                                                        // 1.
   ("Hamon Potential Evapotranspiration." +
   "Climatic demand for water relative to the available energy," +
   " after Hamon.")
@Author
   (name= "Jo Scientist", contact= "jos@research-org.edu")
@Keywords
   ("Hydrology, Potential Evapotranspiration")
@Bibliography
   ("Hamon, W.R., 1961, Estimating potential evapotranspiration. " +
    "Journal of the Hydraulics Division, " +
    "Proceedings of the American Society of Civil Engineers, "
    "v.87, p.107–120.")
@SourceInfo
   ("$HeadURL: http://svn.javaforge.com/svn/oms/branches/ +
         oms3.prj.thornthwaite/src/tw/HamonET.java $")
@Status
   (Status.DRAFT)
@VersionInfo
   ("$Id: HamonET.java 319 2009-08-20 17:34:32Z odavid $")
@License
   ("http://www.gnu.org/licenses/gpl-2.0.html")

public class HamonET {
   // number of days per months
   final static int[] DAYS = {
       31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   };

   @Description("Temperature")                                    // 2.
   @Unit("C")
   @In double temp;

   @Description("Day length in hours.")
   @Range(min=9, max=15)
   @In double daylen;

   @Description("Current time")
   @In Calendar time;

   @Description("Potential ET")
   @Unit("mm/month")
   @Range(min=0.0)
   @Out public double pet;

   @Execute
   public void compute() {
        if (temp <= -1.0) {
           pet = 0.0;
           return;
       }
       int month = time.get(Calendar.MONTH);
       double Wt = 4.95 * Math.exp(0.062 * temp) / 100.;
       double D2 = (daylen / 12.0) * (daylen / 12.0);
       pet = 13.97 * DAYS[month] * D2 * Wt;
       if (pet < 0.0) {
           pet = 0.0;
       }
   }
}
```

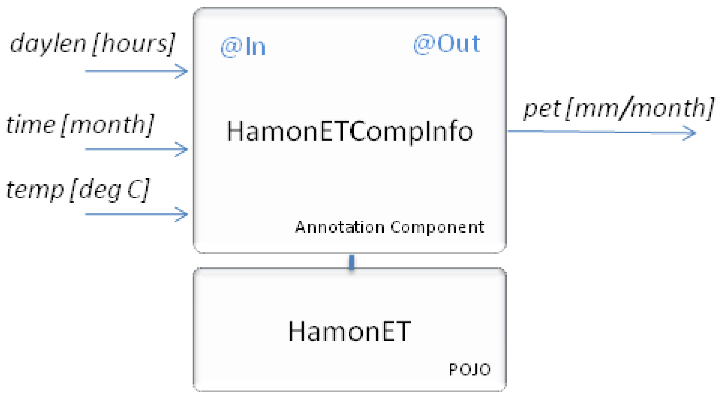
1. Component annotations for documentation, source code repository references, literature references, development status information and licensing.
2. Field annotations for documentation, unit specification and range constraints.

All annotations are explained in detail in Appendix ??.

The component as shown in Step 4 is very well annotated. All code and meta information resides in one source file. This has the great advantage that a metadata/documentation context is always directly attached to the source code. Such an approach is always preferred. I some cases it is necessary to separate the metadata/documentation aspect from the component source. If the component is only available in its compiled form (as POJO), no source code is available, it can still be annotated and used as a OMS3 component.

Lets suppose we have the Java class as shown in step 2 only available as compiled `.class` or `.jar` file (executable binary format). By creating an additional class named `HamonETCompInfo.java`, the compiled `HamonET.class` file can be annotated too. (the `CompInfo` file name suffix indicates the annotation extension of the same class without this suffix). The class `HamonETCompInfo.java` is referred as **Annotation Component** (Figure ???).

Figure 2.5. Annotation Component



The annotation component class must be an abstract Java class. It includes only the language constructs from the component, that are relevant to the framework, such as only the `In` and `Out` fields and the tagged methods. Those methods must be declared `abstract` too, hence they do have no implementation body.

Example 2.5. Pure Annotation Component (HamoETCompInfo.java)

```
package thornthwaite;

import oms3.annotations.*;
import java.util.Calendar;
import java.lang.Math;

@Description
   ("Hamon Potential Evapotranspiration." +
   "Climatic demand for water relative to the available energy, "+
   "after Hamon.")
@Author
   (name= "Jo Scientist", contact= "jos@research-org.edu")
@Keywords
   ("Hydrology, Potential Evapotranspiration")
@Bibliography
   ("Hamon, W.R., 1961, Estimating potential evapotranspiration. " +
    "Journal of the Hydraulics Division, " +
    "Proceedings of the American Society of Civil Engineers, " +
    "v.87, p.107–120.")
@SourceInfo
   ("$HeadURL: http://svn.javaforge.com/svn/oms/branches/" +
        oms3.prj.thornthwaite/src/tw/HamonET.java $")
@Status
   (Status.DRAFT)
@VersionInfo
   ("$Id: HamonET.java 319 2009-08-20 17:34:32Z odavid $")
@License
   ("http://www.gnu.org/licenses/gpl-2.0.html")

public abstract class HamonETCompInfo {                              // 1.

   @Description("Temperature")
   @Unit("C")
   @In double temp;

   @Description("Day length in hours.")
   @Range(min=9, max=15)
   @In double daylen;

   @Description("Current time")
   @In Calendar time;

   @Description("Potential ET")
   @Unit("mm/month")
   @Range(min=0.0)
   @Out public double pet;

   @Execute
   public abstract void compute();                                          HamonETCompInfo  // 2.

}
```

1. The Annotation Component is declared `abstract` and its name is prefixed with `CompInfo`.
2. The compute method is also declared `abstract`, no implementation.

The Component Annotation approach has the advantage to keep the Component 100% as a POJO component with no dependency to the framework annotations. However, it means an additional burden to the developer to keep POJO Component and Annotation Component in sync during development.

Bibliography

[McCabe2007] McCabe, G.J., and Markstrom, S.L., 2007, A monthly water-balance model driven by a graphical user interface: U.S. Geological Survey Open-File report 2007-1088, 6 p.

[Hamon1961] Hamon, W.R., 1961, Estimating potential evapotranspiration: Journal of the Hydraulics Division, Proceedings of the American Society of Civil Engineers, v. 87, p. 107–120.

[Mather1978] Mather, J.R., 1978, The climatic water balance in environmental analysis: Lexington, Mass., D.C. Heath and Company, 239 p.

[Thornthwaite1948] Thornthwaite, C.W., 1948, An approach toward a rational classification of climate: Geographical Review, v. 38, p. 55–94.

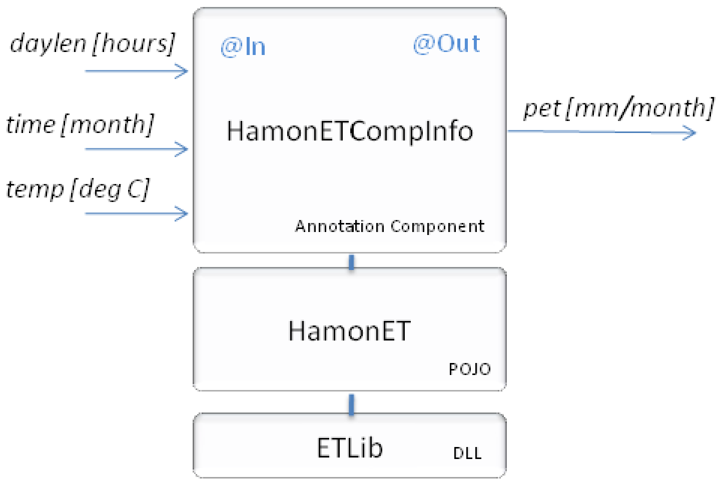
3. A FORTRAN PET Component

Up to this point we discussed the creation of components in the Java programming language. It is possible, however, to create components that originate from languages such as C, C++, and FORTRAN, popular and widely used native languages within the scientific community.

The concept of embedding native code is derived from JNA (Java Native Architecture), an open source library that tremendously simplifies the use of native code by providing a transparent and easy access to Dynamically Linked Libraries (DLL) under Windows or Shared Libraries under Linux/UNIX/MAC OS X. The term DLL refers to those type which represent the same concept. Appendix ??? discusses all details of DLL creation, integration, and use for the C,C++, and FORTRAN programming languages.

In general OMS3 can directly interact with DLLs on all major platforms directly without the need of creating any glue source code between Java and a DLL.

Figure 2.6. Native HamonET Component



As an example, the `HamonET` component now gets implemented using the FORTRAN programming language. Figure ??? shows the involved software units. The component accesses the the DLL (`ETLib.dll`), that is being created from the FORTRAN source code `hamon.f90`. It should be noted that this example used the annotation component approach, the POJO and annotation component can be combined too for native components.

Example 2.6. FORTRAN Component (hamon.f90)

```
! File:   hamon.f90
! Author: od
!
FUNCTION potET(daylen, temp, days) BIND(C, name='hamon')        ! 1.
  REAL*8,VALUE :: daylen,temp                                   ! 2.
  INTEGER*4,VALUE :: days
  REAL*8 :: potET
  REAL*8 :: Wt,D2

  Wt = 4.95 * exp(0.062 * temp) / 100.0
  D2 = (daylen / 12.0) * (daylen / 12.0)
  potET = 0.55 * days * D2 * Wt
  if (potET <= 0.0) then
      potET = 0.0
  endif
  if (temp <= -1.0) then
      potET = 0.0
  endif
  potET = potET * 25.4
END
```

The FORTRAN 90 source as shown above contains some F2003 enhancements which simplify the language interoperability, mostly C interoperability. Most compilers support those features

1. The function declaration ends with the `BIND()` construct, providing for easier C interoperability, mostly taking care of the name underscoring inconsistencies within object files and DLLs across compilers. It allows to define an name alias for other programs to calling this function. In the case above, the FORTRAN function `potET` can be called as `hamon`.

2. The function parameter are declared as 'value parameter', again a F2003 extension supporting C calling conventions.

The rest of the function is just plain FORTRAN implementation if the Hamon equations. As a next step, the file hamon.f90 will be compiled and linked into a DLL, `ETLib.dll` on Windows, and `libETLib.so` on Linux/Unix. The instructions for different compilers on different operations systems can be found in the Appendix ???.

Now the `HamonET` component gets created to use the DLL. Note that if you use a DLL, a developer still needs to create a Java component containing the annotation meta data in order to provide execution and data flow information.

Example 2.7. Component Binding of a DLL

```
package thorthwaite;

import java.util.Calendar;
import oms3.util.Libraries;                                          // 1.

public class HamonET {

   @DLL("ETLib")                                                     // 2.
   interface Et extends com.sun.jna.Library {                        // 3.
       Et lib = Libraries.bind(Et.class);                            // 4.

       double hamon(double daylen, double temp, int days);           // 5.
   }

   static final int[] DAYS = {
       31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   };

   public double temp;
   public double daylen;
   public Calendar time;

   public double pet;

   public void compute() {
       int month = time.get(Calendar.MONTH);
       pet = Et.lib.hamon(daylen, temp, DAYS[month]);                // 6.
   }
}
```

1. OMS contains a support library for handling DLLs. Its a part of the `util` package.

2. The `@DLL` annotation refers to the external DLL name in its argument. It has to annotate the interface that follows the following line.

3. The interface `Et` extends the JNA Library interface.

4. the `Libraries.bind()` call binds the DLL `ETLib.dll` to the interface `ET_FTN`. The static variable lib will reference an an instance which is bound to the native DLL.

5. The `hamon` method listed here is a Java surrogate for the FORTRAN `hamon` method in the DLL. A call to this method would result in calling the FORTRAN function. Note that the name, argument types and return type have to match in order to map.

6. The call performed in compute used the lib reference within `Et` to pass in the actual arguments as Java variables witch are at the receiving end arrive as FORTRAN objects.

At model runtime the it has to be ensured, that the DLL is accessible with the environment path, or a system property `jna_library_path` has to be to point to the DLL's directory. See for details in Appendix ???

4. A C PET Component

The same PET component can also be implemented in C or C++, as the program listing shows below. Default C calling conventions for function arguments and function naming are being used, Also, type names are similar to Java and show somehow the origins of the Java programming language.

Example 2.8. C Component (hamonc.c)

```
/*
 * File:   hamonc.c
 */
double hamon(double daylen, double temp, int days) {
   double Wt = 4.95 * exp(0.062 * temp) / 100.;
   double D2 = (daylen / 12.0) * (daylen / 12.0);
   double potET = 0.55 * days * D2 * Wt;
   if (potET <= 0.0) {
       potET = 0.0;
   }
   if (temp <= -1.0) {
       potET = 0.0;
   }
   potET *= 25.4;
   return potET;
}
```

The code above (hamonc.c) should be compiled into a DLL or shared library using a C or C++ compiler. Its use in a OMS3 component is identical to its FORTRAN counterpart as described in the previous section. The only change might be the DLL name as specified in the `@DLL` annotation.

5. Component Method Annotation

As seen in the developed example, the `compute()` method was annotated with `@Execute` annotation to point the framework to the entry method for a component's execution. This method provides the core purpose and equations of the Hamon class. For the framework the name of this entry method is irrelevant, as long as it has the expected method signature. The method has to be `public`, `void` return type and no arguments, the method name does not matter.

All together there are three types of annotations that indicate execution entry points of a component:

`@Initialize`
: Annotates a component's method that initializes the component. Some components need to setup and handle an internal state in order to allow for their execution. **Note that no input to the component (`@In`) can be used during initialization**. Those values are not present at initialization time. The initialization method gets called only once on model startup. It is also called after instantiation and before this and any other component's execute method. Examples might be allocation of memory, opening of data base connections, file handles, or the creation of a user interface element. This annotation is optional.

`@Execute`
: This annotation annotates the method implementing the core functionality of the component. This method is required. Within this method usually all `@In` tagged fields are accessed, equations are applied and `@Out` fields are being assigned to results. It can be safely assumed that all `@In` tagged fields contain valid values.

`@Finalize`
: Finalize is called after the last `@Execute` of this and any other component in the model. It should be used to gracefully end the model run by cleaning up resources, such as closing open file streams or data base connections, deallocation of memory if native code is involved. This annotation is optional.[1]

All of the annotated methods must have the same signature: `public void <name>()`. Any other signature is invalid. The methods can also declare Exceptions being thrown. A component must have at least the `@Execute` annotation.

The example below shows the use of the method annotations in detail.

Example 2.9. Unconditional Initialization

```
 public class ClimateInput {

    DBConnection db;
    Iterator<String[]> inp;

    @Out public double temp;
    @Out public double precip;
    @Out public boolean moreData;

    @Initialize
    public void init() throws Exception {
        db.open(System.getProperty("db.connect");
        inp = db.iterator();
    }

    @Execute
    public void exec() throws Exception {
        if (inp.hasNext()) {
            String[] row = inp.next();
            temp = Double.parseDouble(row[2]);
            precip = Double.parseDouble(row[3]);
        }
        moreData = inp.hasNext();
    }

    @Finalize
    public void done() {
        db.close();
    }
}
```

The Climate input example above used all three method annotations. A data base connection gets established during initialization, within execute data is fetched from it row by row on each call. At the end of execution the database connection is closed.

The connect information is provided as a static system property in this case. Since the initialization does not depend on any state variable of the component this method is called *unconditional initialization*

5.1. Conditional Initialization

Another practical method for component initialization is shown in the example listing below. Unlike unconditional initialization, the *conditional initialization* can take input values of the component into account. Hence, it has to be called from within `@Execute`, since it needs valid `@In` values. The `@Initialize` tagging is not needed, the `@Ininit()` method can be declared private, since it only called from within this component (1).

Example 2.10. Conditional Initialization

```
public class Climate {

   @In  public File climateInput;
   @Out public double temp;
   @Out public double precip;
   @Out public boolean moreData;
   @Out public Calendar time = new GregorianCalendar();

   /** Row Input iterator*/
   Iterator<String[]> inp;
   /** data formatter */
   private SimpleDateFormat f;

   private void init() throws Exception {         // 1.
       CSTable table = DataIO.table(climateInput, "Climate");
       f = new SimpleDateFormat(table.getColumnInfo(1).get("Format"));
       inp = table.rows().iterator();
   }

   @Execute
   public void execute() throws Exception {
       if (inp == null) {                         // 2.
           init();
       }
       if (inp.hasNext()) {
           String[] row = inp.next();
           time.setTime(f.parse(row[1]));
           temp = Double.parseDouble(row[2]);
           precip = Double.parseDouble(row[3]);
       }
       moreData = inp.hasNext();
   }

   @Finalize
   public void done() {
       DataIO.dispose(inp);
   }
}
```

This methodology provides more flexibility, because it is more dynamic within the execution. However, an checking on an internal variable is usually required to indicate the need for the `init()` call, hence called conditional initialization (2).

6. References

- Java Native Architecture (https://jna.dev.java.net)
- GCC (http://gcc.gnu.org)
- Intel Compiler (http://software.intel.com/en-us/intel-compilers)

Chapter 3. Component Integration

So far, we talked about individual components, their internal structure, annotated elements such as fields and methods, and meta data annotations for components. In this chapter we will explain the assembly of multiple components into coarser grained components, that becomes eventually the model.

1. Component vs. Model

[tbd]

2. Compound Java Components

A Compound is an aggregate of simple components. It can be viewed as a facade for its internal components that are usually considered simpler. Figure ??? shows the schematic of a geometry compound component, that contains three internal components. They are connected to provide data for computing the surface area of a cylinder. They are also connected to the compounds Input and Output. If the internals of the `CylinderCompound` would be omitted, it would look like a simple component with height and radius as input and surface as output. It can certainly be used this way in another compound.

[tbd]

2.1. Connecting Components

The `connect()` method allows to connect the output field of a component (tagged as `@Out`) with an input field of another component (tagged with `@In`). Both components are internal to this compound. Their field types have to be compatible.

```
public void out2in(Object from, String c1Out, Object to, String c2In);
```

`from` - component1 c2

`c1Out` – output field name of component 1

`to` – component 2

`c2In` – input field name of component 2

If c1Out and c2In are the same name, a shortcut can be used:

```
public void out2in(Object c1, String c1Out, Object ... c2);
```

`c1` - component1 c2

`c1Out` – output field name of component 1

`c2` – component 2

2.2. Input and Output mapping

The mapIn() method connects a compounds @In field with an internal component's @In field. The mapOut() methods works similar for `@Out` fields. Unlike the connect() method above, the map???() methods require both field to have the same data flow tag. The types of those fields have to be compatible.

```
public void in2in(String in, Object c2, String c2In)
```

`in` - in field of this component

`c1Out` – output field name of component 1

`c2` – component 2

If both input fiels do have the same name a shortcut can be used:

```
public void in2in(String in, Object ... c)
```

`in` - in field of this component

`c` – internal components to map in to

3. Temporal Iterations

[tbd]

4. Spatial Iterations

[tbd]

Chapter 4. Simulations

In this chapter the structure, setup, and execution of model simulations are discussed.

A simulation within OMS is defined as the application of a model (or component) with a concrete data set to predict the actual behavior of a system or the environment. A developed model component is needed as well as a dataset providing input data for the model. Input data can be supplied directly to the model.

The implemented simulation concept leverages the concept of Domain Specific Languages (DSL) as introduced in the Groovy language. Introducing groovy and DSL is outside of the scope of this handbook, only the implementation for simulation building will be discussed. There is no need to master groovy beyond the described concepts. There are some general technical requirements for developing simulations.

- In order create a new or run an existing simulation the jar file `groovy-all.jar` has to be in the CLASSPATH of your IDE or other runtime environment or you just install the Groovy package (http://groovy.codehaus.net).
- The simulation file, which will be explained in detail below has to have the extension `*.groovy`.

At least, a simulation can be executed using the command line like:

```
$ java -cp "./mycomponents.jar" sim.groovy
```

1. Data Input/Output format

This chapter covers data input and output handling. Although this library is a part of the OMS3 core package structure it is neither depending on OMS3, nor OMS3 is depending on it. They just play well together. Design motivations for this package were (i) the support of typical scientific Data IO such as tables and properties (ii) human readability (it should not be too verbose), (iii) it has to support meta data, (iv) it should be consumable by other tools and (v) it should allow the definition of a simple API, to programmatically read and write data.

There two type of informations that adhere to this are:

- **Tables**, containing tabular information
- **Properties**, referring to key/value property data

The Data IO format was commonly defined for both, tables and properties. The format is based on CSV structure that has been extended with some meta tags. A file might contain any number of tables and properties.

Both types of information can be mixed in the same file and may occur multiple times. The definitions for tables and properties are similar, both support meta data.

- The data file complies fully to the CSV standard.
- The file name extension is csd, standing for "comma separated data". It might be zipped, and would then have the extension csz.
- A csd file might contain a table or property section, or multiple of those, or a mixture of both.
- A # symbol at the beginning of the line indicates a comment line.
- Empty lines are ignored.

1.1. Keywords

Keywords are used to indicate properties and tables in the file.

Table 4.1. CSV Tags

Keyword	Name	Description
@T	Table	Defines a new table
@H	Header	Starts a header in a table
@S	Section	Starts a new property section
@P	Property	Starts a new property

Note

All of those keywords can be followed by optional meta data.

Note

Keywords are case insensitive (`@T` is equal to`@t`).

1.2. Meta data

Meta data may always follow the property and table markups. There is one meta data entry by line. Such an entry may have a key/value pair (separated by a comma), or a single key with no value indicating the presence of a meta data entry.

The property section example below, shows section level meta data supporting the whole "Parameter set" such as data, or `createdBy`, as well as key value pair property meta data such as description or single value properties such as public. It might be good practice to quote meta data values in general to account for potential commas, however it is not required.

Table 4.2. Meta Data Examples (@S and @T)

Name	Description	Example
`CreatedBy`	data set creation date	`CreatedBy, "JCarlson"`
`CreatedAt`	user who created the data set	`CreatedAt, "May 1st, 2008"`
`Description`	brief data set description	`Description, "EFC climate file"`
`VersionInfo`	Version information (use in conjunction with VCS)	`VersionInfo, "$Id:"`
`SourceInfo`	Source information (use in conjunction with VCS)	`SourceInfo, "$HeadURL:"`

1.3. Properties

Properties are key/value pairs (KVP) that are aggregated in a section. There could be meta data for the whole section `@S` and also for each property `@P`. The example below shows a property section.

Property Example:

```
@S, "Parameter"
CreatedAt, "Jan 02, 1980"
CreatedBy, Joe

# Single Properties
@P, coeff, 1.0
description, "A coefficient"
public

@P, start, "02-10-1977"
description, "start of simulation"
```

A proper section starts with the `@S` keyword, followed be the name of the property section. It is followed by optional meta data. Meta data keys/values can be arbitrary, and may occur at any number. A single property starts with the property keyword `@P`, followed by the property name and the property value. Optional meta data may also follow a single property. The property section ends at the beginning of the next section or table or the end of the file.

1.3.1. Property Key/Value Substitution

Properties support internal key/value substitution. This feature helps organizing property sets more efficiently, An example is shown below. A directory property `idir` is defined and internally used by multiple files.

```
...
 # Input file folder (variable)
@P, idir,                      "ccreek"
 Description, "Data directory"

@P, ahumFileName,              "${idir}/ahum.dat"
 Description, "Absolute Humidity Data"

@P, gwFileName,                "${idir}/hgeo.par"
 Description, "Hydrogeology Data"
...
```

The expression `${<prop_key>}` will be replaced with `<prop_value>`, if there is somewhere else within the same property set a property defined as `@P, prop_key, prop_value`.

1.4. Tables

Tables consists of columns and rows, and optional table meta data. Columns may have a type and optional meta data. Meta data is organized as pair key, value. A table requires two key words, `@T` (Table) and `@H` (Table header). The `@T` keyword tags the start of a table definition, the `@H` tag starts a column definition.

Tables can be generated using any text editor. Spreadsheet tools usually do allow the export into a CSV file.

Table Example:

```
# table example
@T, "Example DataSet"
CreatedAt, 5/11/06
CreatedBy,  JackC
# Now, there is header information
@H,     time,b,c
Type,   Date,Real,Real
Format, yyyy-MM-dd,#0000.00,#000.0000
,2006-05-12,0000.00,001.1000
,2006-05-13,0001.00,002.1000
,2006-05-14,0002.00,003.1000
,2006-05-15,0003.00,004.1000
,2006-05-16,0004.00,005.1000
,2006-05-17,0005.00,006.1000
,2006-05-18,0006.00,007.1000
```

A Table consists of three main sections:

1. The table header, indicated by @, followed by the name of the table. The next lines may have table level meta data, one meta data entry per line. Meta data is optional.
2. The table header is followed by the column header, indicated by the `@H` keyword. Next to this all the column names are listed. The next lines may contain column meta data, starting with the key, followed by the values for each column (Example above shows Type and Format for the columns).
3. Data rows start with a ',' as the first character; values are comma separated.

A minimal table with no optional meta data looks like this:

```
@T, example data table
```

```
 @H a, b, c
 , 1,2,3
 , 4,5,6
... more data
```

1.5. Using the DataIO API

[TBD]

2. Concepts and Common Elements

This Section introduces the concepts and common elements of simulations in OMS.

2.1. SimBuilder

The common start and entry point for developing a simulation is the class `SimBuilder`. This class is part of the OMS simulation package.

`SimBuilder` provides for two main features

1. The **Creation** of different kinds of simulations by using a very easy language structure

2. The **Execution** of a simulation once its built successfully.

. There are at least two parts that are common to every simulation as shown below:

A first Simulation:

```
sb = new oms3.SimBuilder()                 // (1)
sb.sim(name:"SimpleTWModel for EF") {      // (2)
   // define the model
   model(classname:"tw.Thornthwaite") {
       // add parameter
       parameter {
           climateFile  "c:/od/projects/ngmf.models/src/tw/climate.cst"
           outputFile   "output.csv"
           runoffFactor 0.5
           latitude     35.0
           smcap        200.0
       }
   }
}
```

(1) Creating the Builder	This statement instantiates a builder object using its class `oms3.SimBuilder`; In the example above it is names `sb`, but this name is just arbitrary.
(2) Creating the Simulation from the builder	Use the `SimBuilder` instance to create the simulation. The example above used the `simu` method to create a basic simulation (`simu` is explained in detail below)

The `SimBuilder` class can be instantiated with the following properties:

level (java.util.logging.Level)	The logging `level` that will be used on simulation building and be passed on to the simulation instance. Valid logging levels are defined in `java.util.logging.Level` as `ALL, FINEST, FINER, FINE, CONFIG, INFO, WARNING, SEVERE,` or `OFF`. Setting the level to `CONFIG` will for example report the building of the simulation as it performs. If not provided, logging is set to `OFF`.

The following example illustrated the proper setting of those properties.

```
sb = new oms3.SimBuilder("CONFIG")
```

```
sim = sb.sim(name:"SimpleTWModel for EF") {
  // define the model
  //...
}

sim.run()   // call run(), since no autorun!
```

Setting the properties above may help inspecting the simulation build process as it progresses without automatic execution.

OMS supports different flavors of model simulations that are all constructed in a similar way from the `SimBuilder` class. This allows constructing and performing basic simulations, model calibrations, forecasting methods, uncertainty and sensitivity analysis, or just plain model testing in an easy and consistent way. The following types of simulations can be created using `SimBuilder`.

Table 4.3. Simulation Types

Simulation	Description
`sim`	Basic Simulation
`esp`	Ensemble Streamflow Prediction
`luca`	Model calibration using the LUCA method
`sce`	Shuffled Complex Evolution
`mocom`	Multi-objective complex evolution procedure
`glue`	Generalized Likelihood Uncertainty Estimation
`dds`	Dynamic Dimensioned Search
`rsens`	Relative Sensitivity
`test`	Automated model testing

The following sections introduce elements that are commonly used across all simulation types, followed by the simulations them selfs.

Common Simulation Element Structure

All simulations adhere to the same formal structure as used in the first simulation:

```
// comment
 <element>(<key:value>, <key,value>, ...) {
  <element>(<properties like above>) {
     // more sublements..
  }
  <element>(<properties like above>) {
     // more sublements or just elements with value
     <element> <value>
  }
  <element>(<properties like above>)
  <element> {
     <element ..
  }
  // more subelements
}
```

- There is one root element that is usually the simulation.
- Elements might have properties, provided in parenthesis after the element name. If there are no properties the parenthesis can be omitted.
- Properties are a list of comma separated tuple of `<key:value>` pairs.
- Elements might have sub elements within curly brackets. If there are no sub elements, the curly brackets can be omitted.

- Elements can have just a value following the element name separated by space .
- Comments can be single lined ('`// ...`') or can span multiple lines ("`/* ... */`") such as in C++, Java, or Groovy.

2.2. Model (`model`)

The model element is a part of every simulation and describes a model/component to be used. It is really top component class that is represented here. The component can be any Class that has at least the `@Execute` annotation indicating the execution entry point.

The example below defines a model by using the class "`tw.Thornthwaite`".

```
model(classname:"tw.Thornthwaite") {
    // optional parameter definitions
}
```

The model might have optional parameter subelements. If no parameter should be defined, the subelement body can be omitted:

```
model(classname:"tw.Thornthwaite")
```

Specification

Name `model` - specifies a model for a simulation

Properties

Name	Description	Type	Required
`classname`	the classname of the model.	String	Y

Note

The classpath of the simulation should include all required classes for the model.

Subelements

Name	Description	Type	Default	Occurrences 1)
`parameter`	the model parameter set	-	-	*

Parent(s) all simulation types

Notes

- 1) Occurrences: 1 - exact one time; + - one or more time; ? - zero or one time; * - zero or more time
- The class as specified in `classname` and every other class has to be found in the classpath of a simulation. Setting the classpath can be done using command line arguments on simulation execution, using environment variables, or other methods.

2.2.1. Parameter (`parameter`)

The model parameter element allows the specification of input values for a model. It can reference a external file (csd) that contains the model parameter as specified in Section ???. This element also allows the direct specification of parameter as sub-element. An example:

```
model(classname:"tw.Thornthwaite") {
   // parameter
   parameter(file:"params.csd") {
       climateFile  "c:/od/projects/ngmf.models/src/tw/climate.cst"
       outputFile   "output.csv"
```

```
        runoffFactor 0.5
        latitude     35.0
        smcap        200.0
    }
}
```

the parameter 'file' property takes a file name. Note that this file can be an absolute file path or a relative one, that will relate to the base directory of the simulation. The parameter element can also have the parameter names/ values as sub-elements as shown above. Names are matching the `@In` tagged fields of the model that is named in the surrounding model element. Values are space separated from their keys and have valid Java/Groovy data types such as Strings, Numbers, Files, etc. Those data types have to match the data type of the corresponding `@In` field in the model. However, the system will convert the values into the proper field data type, if the value is provided as String. (The `climateFile` value in the example above is of type `File`, but for convenience purposes it is provided as String.

If both, the `file` property is provided and there are also parameter values given, and both the file and the subsection specifies the same parameter, then the sub element will overwrite the file specified parameter.

The following use of the parameter element is possible. Multiple parameter elements can help splitting parameter sets in groups and allow for redefinition.

```
model(classname:"my.model") {
   // parameter defintion
   parameter(file:"params.csv")          // parameterfile only
   parameter(file:"params-dates.csv")    // parameterfile only
   parameter(file:"params-files.csv") {  // parameterfile and explicit
       testdir "/tmp/test"
   }
   parameter {                           // only explicit parameter
       coeff 2.34
   }
}
```

More general about parameter value reading and setting order: A parameter at a higher line number will overwrite the same one at a lower line number. It is not relevant if it comes from a file or is specified explicitly.

Specification

Name `parameter` - describes a model parameter

Properties

Name	Description	Type	Required
`file`	the parameter file	String (csd - file)	N

Sub elements

Name	Description	Type	Default	Occurrences
`<pname> <pvalue>`	single model parameter	<component field type>	-	*

Parent(s) `model`

Notes

- A parameter element must have either a file property or at least one single parameter sub-element or both, but cannot have non of both.

2.2.2. Resources (`resource`)

Every simulation has to manage resources such as a model executable, DLLs, parameter files, climate data input, documentation, etc. The `resource` element allows the listing of those resources. There are several uses for the resource listings in a simulation.

- All jar files listed in a resource element are added to the `classpath` for JAVA model execution. Jar files can be referenced as local files or URLs, if the model should be loaded from a remote location. If no Jar files are present, the model will use the default class path for the application.

- All files regardless of which type are used for digest computation to ensure comprehensive hashing of all simulation resources.
- Other tools for remote execution within a cluster can use the resource listing to copy thiose files to other machines.

A `resource` section of a simulation might look like:

```
work = "home/prj"
sim(name:"ceap") {
    //
    resource "$work/dist/oms3.prj.ceap.jar"
    resource "$work/dist/oms3.prj.ceap-lib.jar"
    resource "$work/input/climate.csv"
}
```

The `resource` values always follow the resource keyword. It also shows the use of string replacement in order to reference a common root directory. Alternatively the files above can be provided as a list to one resource element. (Note the required brackets and parenthesis). Both notations do have the same semantics.

```
 ...
 resource (["$work/dist/oms3.prj.ceap.jar",
            "$work/dist/oms3.prj.ceap-lib.jar",
            "$work/input/climate.csv"])
 ...
```

Specification

Name `resource` - list all relevant simulation resource files

Value

	Description	Type	Required
`<single file>`	a file belonging to this simulation	String (file or URL)	y
`<file list>`	all file belonging to this simulation	List of Strings (file or URL)	y

Parent(s) `sim, model`

Notes

- A `resource` element must have either have a single file value or a list of files.
- The `resource` element should at least list all `.jar`, `.exe`, `.dll` files that are needed for execution.
- Listed files should provide the full path.
- The resource element of a `sim` element and a `model` element are shared.

2.2.3. Logging (`logging`)

The logging sub-element is an optional part of a model element. It controls the logging levels for single components or for the whole model. In order to use the logging feature, components have to obtain and use a logger accordingly.

A logger is an object that allows output handling based on logging levels. Such levels usually indicate the severeness of a message. The Java logging infrastructure supports per default 7 log levels, ranging from `FINEST` (the lowest priority or importance) to `SEVERE` (the highest importance). In addition there is a level `OFF` to turn off logging at all. If a log level is provided, all log message of the same or higher priority are passed to the system and printed out.

The examples below shows the use of the logging. The logging element is part of the model element. It lists the component class names and their associated log levels for a simulation run.

```
 model(classname:"my.model") {
    // logging definition
    logging {
        "StreamFlow" "INFO"
        "GwFlow"     "CONFIG"
    }
}
```

The component `StreamFlow` in '`my.model`' is assigned the Log level `INFO`, the `GwFlow` component will have a finer grained `CONFIG` log level. The default log level for all other components in the model is `WARNING`.

```
model(classname:"my.model") {
    logging (all:"INFO"){
        "StreamFlow" "FINEST"
    }
}
```

Now the default log level for all model components is set to INFO, StreamFlow has the most verbose log level.

```
model(classname:"my.model") {
    logging (all:"OFF")
}
```

The logging element above turns off all logging for the whole model. In such a configuration, even severe problems within components are not reported. This statement should be used with care.

Specification

Name `logging` - assigns log levels to components

Properties

Name	Description	Value	Required
`all`	The base log level for all components in the model.	`'OFF'`, `'SEVERE'`, `'WARNING'`, `'INFO'`, `'CONFIG'`, `'FINE'`, `'FINER'`, `'FINEST'`	N (default: `WARNING`)

Sub-elements

Name	Description	Type	Default	Occurrences
`<comp name> <log level>`	single component log level	`<String> ['OFF', 'SEVERE', 'WARNING', 'INFO', 'CONFIG', 'FINE', 'FINER', 'FINEST']`	-	*

Parent(s) `model`

Notes

- If there is no logging element with a model element, all components will default to the `WARNING` level.

2.2.3.1. Setting up logging in a component

To use the logging features within a simulation as described in the previous section, the component has to be setup properly. It has to obtain and use a logger object:

Usually, the logger object is obtained as a static reference somewhere in the declaration part of a component:

```
import java.util.logging.*;                 // 1.
...
public class Ddsolrad {
```

```
static final Logger log =
   Logger.getLogger("oms3.model." + Ddsolrad.class.getSimpleName()); //2.
...

 @Execute
 public void exec() {
   ...

    if (log.isLoggable(Level.INFO)) {           // 3.
        log.info("Solrad " + basin_potsw);      // 4.
    }
 }
 ...
}
```

1. Import the logging classes from the `java.util` package.
2. Obtain the logger using the `Logger.getLogger()` call. Declare this reference `static` to share it across all instances of this class and `final` to make it a constant. The argument must start with the String "oms3.model." and must to end with the component's simple class name. Use `getClassName()` as shown above to obtain this name from the class, instead if manually adding this to the logger name String. Refactoring tools will respect this and will change the logger name properly if needed.
3. At any location within the component methods the logger can be used. As shown above, 'guarded logging' is recommended. It is a pattern that checks to see if a log statement will result in output before it is executed. This will reduce the memory fragmentation and garbage collection by avoiding the creation of unnecessary strings if log levels are disabled. The statement here checks if logging is enabled at the `INFO` level and above.
4. The statement issues the logging message at the '`INFO`' level. Use the methods `servere()`, `warning()`, `info()`, `config()`, `fine()`, `finer()`, and `finest()` accordingly.

The use of Logging in component provides great flexibility for diagnostics and messaging, that is efficient and configurable from within a simulation.

2.3. Simulation Output Strategy (`outputstrategy`)

A simulation usually produces output files such as times series predicted runoff, sediment yield, etc. The `output-strategy` element of a simulation manages the storage of the output based on different strategies. However it does not manage the files or the values them self. It provides for a consistent method and strategy dealing with subsequent simulations.

An example simulation might use a output element:

```
sb = new oms3.SimBuilder()
sb.sim(name:"SimpleModel") {

   outputstrategy(dir:"c:/tmp/out", scheme:NUMBERED)

   // define the model
   model(classname:"tw.Thornthwaite") {
       // add parameter
       parameter {
           climateFile  "c:/od/projects/ngmf.models/src/tw/climate.cst"
       }
   }
}
```

The types of supported output strategy schemes are:

SIMPLE — The simulation creates a folder to hold the model output files. Each new simulation run will overwrite existing files with the same name. The simulation output folder is always:

```
<output dir> + <sim name>
```

For the example above the output would always go into "`c:/tmp/out/SimpleModel`"

NUMBERED The simulation creates a new folder for each simulation run. A new simulation will not overwrite the output from the previous one. The last simulation always has the highest number folder. The simulation output folder is:

```
<output dir> + <sim name> + <simulation run number>
```

For the example above the output of the 5th run would go into "`c:/tmp/out/SimpleModel/0005`"

TIME The simulation creates a new folder for each simulation run. A new simulation will not overwrite the output from the previous one. The last simulation output is always in the folder named with the simulation start time. The simulation output folder is:

```
<output dir> + <sim name> + <simulation start time>
```

For the example above the output a run would go into "`c:/tmp/out/SimpleModel/2009-04-05T12:04`"

In the future there might be more strategies for output handling.

Specification

Name `output` - describes simulation output management

Properties

Name	Description	Type	Required
`dir`	the output base dir, must exist.	`String`	N (default: `java.tmp.dir`)
`strategy`	the output strategy to be used for this simulation	`'SIMPLE' | 'NUMBERED' | 'TIME'`	N (default: `SIMPLE`)

Parent(s) `simu`

Notes

- If there is no output specified in sim, the defaults for both, dir and strategy apply.
- The output strategies are defined in `oms3.SimConstants`
- Output strategy cannot be combined.

2.4. Model Efficiencies (`efficiency`)

Model efficiencies are commonly used to quantify the prediction performance of a simulation model by computing some aggregate based on observed and simulated values of the same model property.

Several model efficiencies are available, see table below.

Table 4.4. Efficiencies

Name (KEY)	Description	Equation
`ABSDIF`	Absolute difference	$\mathrm{ABSDIF} = \sum_{t=1}^{m} \left\| \frac{Q_{t,o} - Q_{t,s}}{Q_{t,o}} \right\|$
`LOGABSDIF`	Log of the absolute difference	$\mathrm{LogABSDIF} = \sum_{t=1}^{m} \left\| \ln Q_{t,o} - \ln Q_{t,s} \right\|$
`NS`	Nash-Sutcliffe	$\mathrm{NS} = 1 - \frac{\sum_{t=1}^{m} (Q_{t,o} - Q_{t,s})^2}{\sum_{t=1}^{m} (Q_{t,o} - Q_o)^2}$

Name (KEY)	Description	Equation
`LOGNS`	Log of Nash-Sutcliffe	$\text{LogNS} = 1 - \frac{\sum_{t=1}^{m} \lvert \ln Q_{t,o} - \ln Q_{t,s} \rvert}{\sum_{t=1}^{m} \lvert \ln Q_{t,o} - \ln Q_o \rvert}$
`LOGNS2`	Log of Nash-Sutcliffe (Pow 2)	$\text{LogNS}^2 = 1 - \frac{\sum_{t=1}^{m} \left(\ln Q_{t,o} - \ln Q_{t,s}\right)^2}{\sum_{t=1}^{m} \left(\ln Q_{t,o} - \ln Q_o\right)^2}$
`IOA`	Index of Agreement	$\text{IOA} = 1 - \frac{\sum_{i=1}^{m} \lvert Q_{i,o} - Q_{i,s} \rvert}{\sum_{i=1}^{m} \lvert Q_{i,s} - Q_o \rvert + \lvert Q_{i,o} - Q_o \rvert}$
`IOA2`	Index of Agreement (Pow 2)	$\text{IOA}^2 = 1 - \frac{\sum_{i=1}^{m} \left(Q_{i,o} - Q_{i,s}\right)^2}{\sum_{i=1}^{m} \left(Q_{i,s} - Q_o\right)^2 + \left(Q_{i,o} - Q_o\right)^2}$
`R2`	Goodness of fit	$R^2 = \left(\frac{\sum_{i=1}^{n} \left(X_i - X\right)\left(Y_i - Y\right)}{\sqrt{\sum_{i=1}^{n} \left(X_i - X\right)^2} \sqrt{\sum_{i=1}^{n} \left(Y_i - Y\right)^2}}\right)^2$
`GRAD`	Linear Regression Gradient	$\text{GRAD} = \frac{\sum_{i=1}^{n} \left(X_i - X\right)\left(Y_i - Y\right)}{\sum_{i=1}^{n} \left(X_i - X\right)^2}$
`WR2`	Weighted Correlation Coefficient	$\text{WR}^2 = \begin{cases} \lvert \text{GRAD} \rvert R^2, & \text{GRAD} <= 1 \\ \frac{1}{\lvert \text{GRAD} \rvert} R^2, & \text{GRAD} > 1 \end{cases}$
`DSGRAD`	Double Sum Analysis Gradient	[TBD]
`AVE`	Absolute Volume Error	$\text{AVE} = \lvert \sum_{i=1}^{m} Q_{i,s} - Q_{i,o} \rvert$
`RMSE`	Root Mean Square Error	$\text{RMSE} = \sqrt{\frac{1}{m} \sum_{t=1}^{m} \left(Q_s - Q_o\right)^2}$
`PBIAS`	Percent BIAS	$\text{PBIAS} = 100 \frac{\sum_{i=1}^{n} \left(Q_{i,o} - Q_{i,s}\right)}{\sum_{i=1}^{n} Q_{i,o}}$
`PMCC`	Pearson product-moment correlation coefficient	$\text{PMCC} = \frac{\sum_{i=1}^{n} \left(X_i - X\right)\left(Y_i - Y\right)}{\sqrt{\sum_{i=1}^{n} \left(X_i - X\right)^2} \sqrt{\sum_{i=1}^{n} \left(Y_i - Y\right)^2}}$
`TRMSE`	Transformed Root Mean Square Error	$\text{TRMSE} = \sqrt{\frac{1}{m} \sum_{t=1}^{m} \left(Z_s - Z_o\right)^2}$, $Z = \frac{\left(1+Q\right)^{\lambda} - 1}{\lambda}$
`ROCE`	Runoff Coefficient Error	$\text{ROCE} = \left\lvert \frac{Q_s}{P} - \frac{Q_o}{P} \right\rvert$

The simulation below shows the use of the efficiency element in a simulation. Multiple efficiencies can be computed at once. Just combine those by using the '+' operator as shown.

```
sim(name:"Efcarson") {
   // define the model
   model(classname:"model.PrmsDdJh") {
       // ... parameter here
   }
   efficiency(obs:"runoff[0]", sim:"basin_cfs", methods:NS+NS2+ABSDIF+TRMSE)
}
```

Executing the simulation will produce additional table output for the requested efficiencies:

```
 Efficiencies           ns1        ns2     absdif      trmse
runoff/basin_cfs    0.66512    0.82971  764.30044    2.44043
```

Specification

Name `efficiency` - model efficiency computation

Properties

Name	Description	Type	Required
`obs`	a output field that provides observed values	String	Y
`sim`	a output field that provides simulated values	String	Y
`precip`	precipitation values	String	Y (only for `ROCE`, ignored otherwise)
`method`	efficiency method(s) to compute	KEYs (can be combined)	Y
`file`	the output file	String	N (if missing output goes to the console, otherwise to the specified file located in the output folder.)

Parent(s) `sim`

Notes

- Multiple keys can be combined using the + operator, the output will be a combined table.
- Multiple efficiencies can use the same file for output. They get appended.

2.5. Summary Output (`summary`)

The summary element provides ad-hoc statistics for selected model (state) variables. This element is a part of a simulation. Statistical moments are computed over an aggregation period that can be selected. The period can be daily, weekly, monthly, yearly, or the entire simulation. A summary is always specified for *one* variable at a time. That variable *must* be output of one component in the model.

The examples below shows the use of the summary element within the '`SimpleModel`' simulation.

```
sim(name:"SimpleModel") {

   // define the model
   model(classname:"tw.Thornthwaite") {
       ...
   }
   summary(time:"time", var:"basin_ro", statistics:MAX, file:stats.txt)
}
```

The maximum value of the output variable basin_ro gets computed over the total simulation period, and the output will be stored in the file stats.txt, located in the simulation run output folder.

```
sim(name:"SimpleModel") {

   // define the model
   model(classname:"tw.Thornthwaite") {
       ...
   }
   summary(time:"time", var:"runoff[4]", statistics:MEAN+MIN+LAG1, period:YEARLY)
}
```

The runoff array element #4 will be aggregated over one year and its minimum, mean and autocorrelation will be printed to the console.

Table 4.5. Statistical Moments

Moment	Description
`MEAN`	$\mathrm{MEAN} = \frac{1}{N}\sum_{i=1}^{N} x_i$
`MAX`	$\mathrm{MAX} = \max_i(x_i)$
`MIN`	$\mathrm{MIN} = \min_i(x_i)$
`COUNT`	$\mathrm{COUNT} = \mathrm{count}\, x_i$
`RANGE`	$\mathrm{RANGE} = \max_i(x_i) - \min_i(x_x)$
`MEDIAN`	$\mathrm{MED} = \begin{cases} Y_{(N+1)/2} & \text{,if N is odd} \\ \frac{1}{2}\left(Y_{N/2} + Y_{1+N/2}\right) & \text{,if N is even} \end{cases}$
`STDDEV`	$\mathrm{SD} = \sqrt{\frac{N}{i=1}\sum_{i=1}^{N}(x_i - x)^2}$
`VAR`	$\mathrm{VAR} = \frac{1}{N}\sum_{i=1}^{N}(x_i - x)^2$
`MEANDEV`	$\mathrm{MD} = \frac{1}{N}\sum_{i=1}^{N}\lvert x_i - x\rvert$
`SUM`	$\mathrm{SUM} = \sum_{i=1}^{N} x_i$
`PROD`	$\mathrm{PROD} = \prod_{i=1}^{N} x_i$
`Q1`	First Quartile [Tbd]
`Q2`	Second Quartile (MEDIAN) [TBD]
`Q3`	Third Quartile [TBD]
`LAG1`	LAG-1 autocorrelation [TBD]

Specification

Name `summary` - ad-hoc summary statistics

Properties

Name	Description	Type	Required
`time`	the time field to be used to compute the aggregation period	String	Y
`var`	a output field that provides the values	String	Y
`statistics`	Statistical moment(s) to compute	*Statistical Moments* KEY	Y
`period`	aggregation period	`'DAILY'` | `'WEEKLY'` | `'MONTHLY'` | `'YEARLY'` | `'TOTAL'`	N (defaults to `'TOTAL'`)
`file`	the output file	String	N (if missing output goes to the console, otherwise to the speci-

Name	Description	Type	Required
			fied file located in the output folder.)

Parent(s)	`sim`
Notes	• Multiple keys for statistics can be combined using the + operator, the output will be a combined table. • Multiple statistics can use the same file for output. They get appended. • Aggregation periods cannot be combined. • Variable names may refer to scalars or array elements using the Java style (e.g. 2D array element: `ro[1][0]`)

2.6. Dynamic Output (output)

Optional to a simulation, the output element can be used to capture and store output field values in an CSV output file. This is an alternative to a component-based solution, where the a dedicated output component is a integrated part of the model. Both approaches do have pro and cons.

Model output component	*Pros* - Can write to any data store; essential model feature, hight performance *Cons* - No ad-hoc change of output variables, change needs model recompilation.
Simulation output element	*Pros* - Output variables can be changed or disabled altogether without model recompilation, very flexible. Well suited for component output ad-hoc inspection *Cons* - Slower in performance that dedicated output component. Only CSV data file output supported for the output element

Note that both types of output definition can co-exist in one simulation.

The examples below show the use of the output element within a simulation.

```
sim(name:"Efcarson") {
   // ...Efficiencies
   model(classname:"prms2008.PrmsDdJh") {
      // ...
   }
  output(time:"date", vars:"basin_gwflow_cfs,basin_cfs,runoff[0]",
          fformat="7.3f", file:"out1.csv")
  // ...
}
```

The output is driven by `date`, an out field in the model. The variables to be captured in an output file called '`out1.csv`' are listed in vars. Note that an array element `runoff[0]` is a part of this. The numerical output format for floating point variables is "`7.3f`" (7 digits total, 3 decimal rounding). The output file gets stored to the output folder as defined in `outputstrategy`. For the configuration above it will have the following content:

```
@T, "Efcarson"
 Created, "Tue Sep 15 14:31:36 MDT 2009"
@H, date, basin_gwflow_cfs, basin_cfs, runoff[0]
 Type, Date, Double, Double, Double
,1980-10-01 12:00:00,    116.453,    116.453,     84.000
,1980-10-02 12:00:00,    114.974,    117.640,     82.000
,1980-10-03 12:00:00,    113.514,    113.514,     80.000
,1980-10-04 12:00:00,    112.072,    112.072,     80.000
,1980-10-05 12:00:00,    110.649,    110.649,     80.000
```

```
...
```

Note: The name of the simulation will be used as table name (EFcarson).

Specification

Name `output` - simulation defined output

Properties

Name	Description	Type	Required
`time`	the time field to be used to trigger output on a time change.	String	Y
`vars`	a list of output fields that provides the values on each time step	String, (field names separated by ',', ';', or ':'	Y (at least one field name)
`fformat`	format for floating point values	String	N (default '10.3')
`dformat`	format for integer values	String	N (default '10')
`file`	the output file	String	N (if missing output goes to the console, otherwise to the specified file located in the output folder.)

Parent(s) `sim`

Notes

- If multiple output elements are used, each should have its own unique file name. It is also recommended to use a file for output when specifying multiple outputs in a simulation.

2.7. Analysis (`analysis`)

An analysis elements provides for post run analysis by means of plotting/graphing features. It is an optional part of a simulation. Performing an analysis will usually result in graphs. The following basic types of analysis plots are available

- Time series plots
- Flow duration plots
- Scatter plots

In addition for Ensemble Streamflow Prediction:

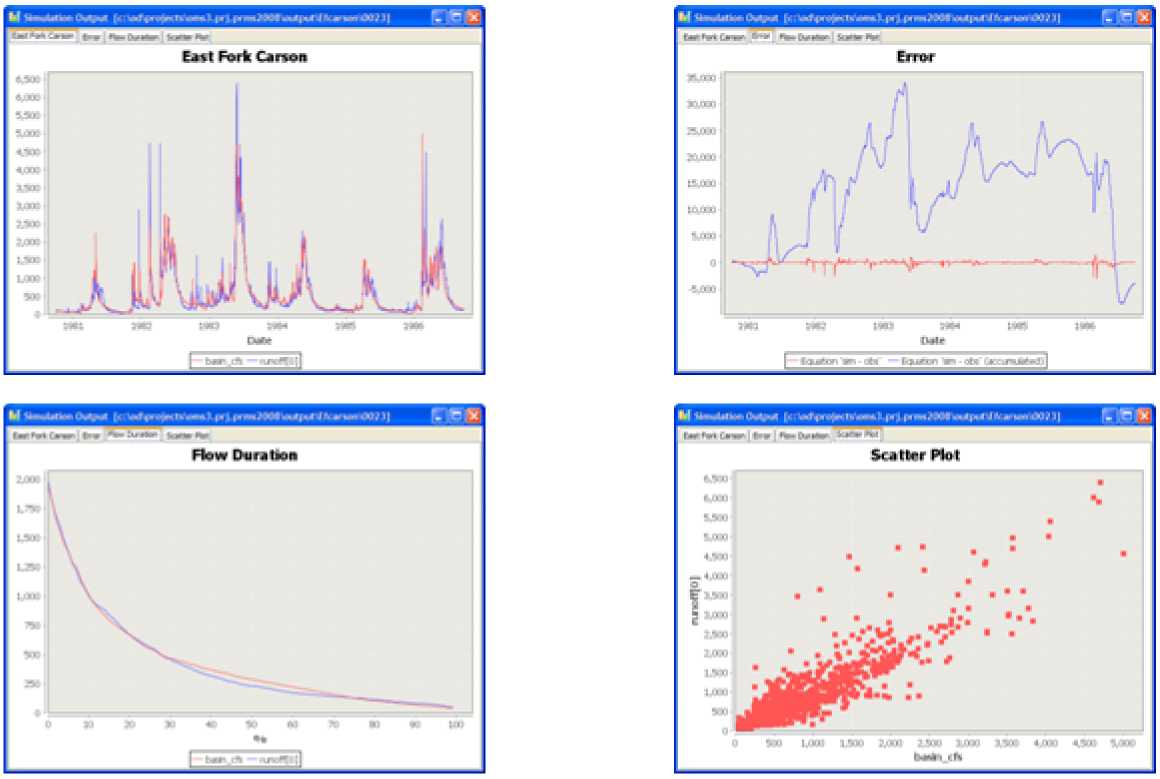
- Esp trace analysis plots

An analysis can contain any number of those plots as sub-elements as shown below.

```
sim(name:"Efcarson") {
  ...
  analysis(title:"Simulation Output") {
       tsplot(title:"East Fork Carson") {
            x(file:"%last/out1.csv", column:"date")
            y(file:"%last/out1.csv", column:"basin_cfs")
            y(file:"%last/out1.csv", column:"runoff[0]")
        }
        tsplot(title:"Error") {
            x(file:"%last/out1.csv",  column:"date")
            expr(eq:"sim - obs") {
                sim(file:"%last/out1.csv", column:"basin_cfs")
                obs(file:"%last/out1.csv", column:"runoff[0]")
            }
            expr(eq:"sim - obs", acc:true) {
                sim(file:"%last/out1.csv", column:"basin_cfs")
                obs(file:"%last/out1.csv", column:"runoff[0]")
            }
        }
        flowduration {
            y(file:"%last/out1.csv", column:"basin_cfs")
            y(file:"%last/out1.csv", column:"runoff[0]")
        }
        scatterplot {
            x(file:"%last/out1.csv", column:"basin_cfs")
            y(file:"%last/out1.csv", column:"runoff[0]")
        }
    }
    ...
}
```

The configuration as shown above will result in output graphs as showed in Figure ???. The whole analysis will appear as a separate window, each plot will have its own tab and graph. The Figure ??? actually shows the same window with different tab being activated.

Figure 4.1. Example Analysis


Simulation Output [c:\od\projects\oms3.prj.prms2008\output\efcarson\0023]
East Fork Carson
Error
Flow Duration
Scatter Plot
Date
basin_cfs
runoff[0]
Equation 'sim - obs'
Equation 'sim - obs' (accumulated)
%
basin_cfs
runoff[0]

There are some general rules for referencing data sets that apply to all analysis types as described in the following sub-sections.

2.7.1. Referencing data sets

All plots are referencing data sets that are stored as CSV tabular data. To identify a column a (i) file name, (ii) table name, and (iii) column name have to be provided. However, the analysis can handle some shortcuts in context of the simulation. There are some examples:

```
x(file:"c:/tmp/SIM/0003/out1.csv", table"efc", column:"runoff")
```

A column is referenced fully by its file, table, and column name. The file name is absolute.

```
x(file:"c:/tmp/SIM/%last/out1.csv", table"efc", column:"runoff")
```

A column is referenced fully by its file, table, and column name. The file name is absolute but references the last simulation run. The meaning of 'last' depends on the chosen output strategy.

```
x(file:"%last/out1.csv", table"efc", column:"runoff")
```

Now the file reference is in context to the simulation. It points to a file in the last output folder for this simulation.

```
x(file:"%last/out1.csv", column:"runoff")
```

If the table name is not provided, the name of the simulation is assumed. The last variant is preferred, since it provides the most flexibility for referencing data path independent.

OMS3 defines 3 'variables' to refer to s simulation run context. They can be used in a `file` value as shown above. Note, that for a `SIMPLE` output strategy, they all refer to the same output folder.

`%first`	The first simulation output in this run sequence. (For numbered outputs is the folder with the lowest number, for timed output the oldest simulation time)
`%previous`	The previous previous simulation output in this run sequence. (For numbered output this is the folder with the highest number - 1, for timed output the second recent simulation time)
`%last`	The last simulation output in this sequence (For numbered output this is the folder with the highest number, for timed output the most recent simulation time)

Using those variables has several benefits. A analysis configuration needs to created once and can always reference the most recent output of a simulation after each run (use `%last`). I addition, someone can always compare the last run's output with the previous output and analyse the effect of model parameter changes. In another scenario a modeller might want to compare the last run's output against a baseline data set, that is being referenced with a full qualified absolute path name. Again, not analysis file needs to be adjusted after each run. `%first`, `%previous`, and `%last` are supporting therefore the modeller's work flow.

Specification

Name `axis (x,y, ...)` - column reference

Properties

Name	Description	Type	Required
`file`	CSV file name	String	y
`table`	table name.	String	n (default simulation name)
`column`	column name.	String	y

Parent(s) `timeseries, flowduration, scatter`

Notes

- The y axis must be referencing a column with numerical values.
- Columns can have different number of rows.

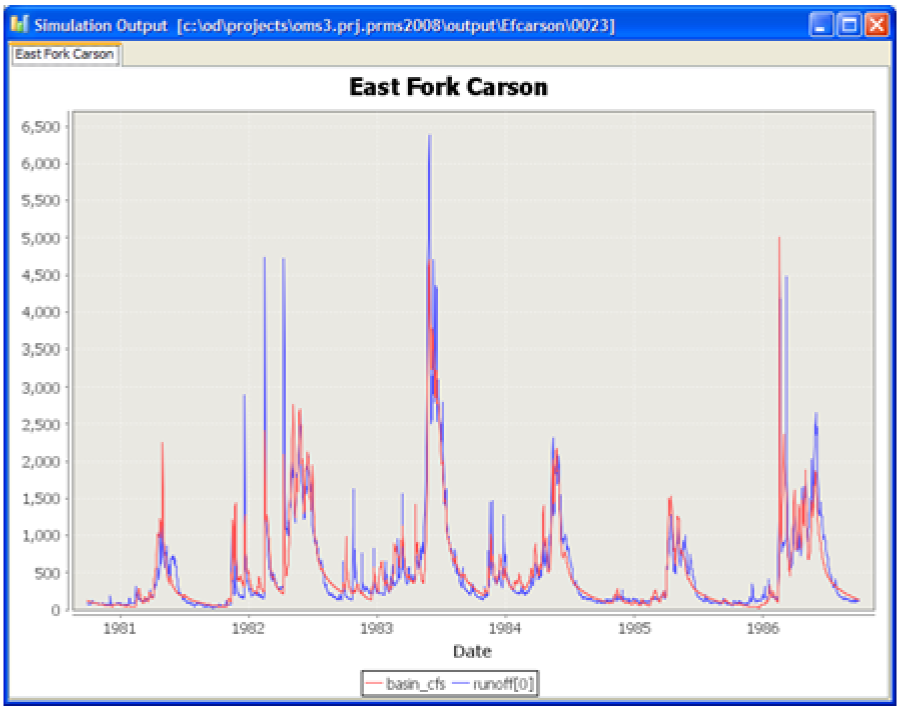
2.7.2. Time Series

A simple time series plot can be configured using the `timeseries` element within an analysis. It takes one x sub element referring to the time series column and a variable number of y elements for the data graphs.

```
analysis {
 timeseries(title:"East Fork Carson") {
     x(file:"%last/out1.csv", column:"date")
```

```
        y(file:"%last/out1.csv", column:"basin_cfs")
        y(file:"%last/out2.csv", column:"runoff[0]")
    }
    ...
}
```

The example above defines the x axis as the date column of the last run that produces out1.csv. The two y data sets (basin_cfs, and runoff[0]) are obtained from different files. The visual output might look like the screen shot below.



Specification

Name `timeseries` - time series chart

Properties

Name	Description	Type	Required
`title`	chart title.	String	N

Sub elements

Name	Description	Type	Default	Occurrences
x	x axis, independent var	Date column		1
y	y axis	value column		+

Parent(s) `analysis`

Notes

- The x axis must be referencing a column with dates.
- All columns must have the same number of rows.

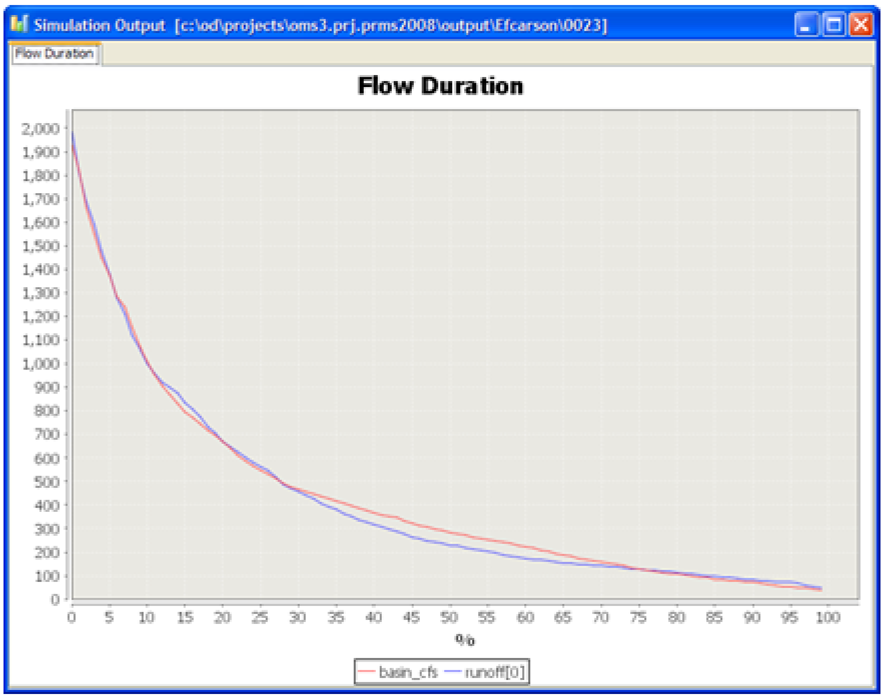
2.7.3. Flow Duration

Flow duration is a plot showing the percentage of time that stream flow is likely to equal or exceed some specified value of interest. It can be used to show the percentage of time river flow can be expected to exceed a design flow of some specified value, or to show the discharge of the stream that occurs or is exceeded some percent of the time.

Within an analysis, the `flowduration` element allows the creation of such graphs. It takes y axis elements as sub elements from which the flow duration gets computed.

```
analysis {
  flowduration {
      y(file:"%last/out1.csv", column:"basin_cfs")
      y(file:"%last/out1.csv", column:"runoff[0]")
  }
  ...
}
```

This example defines the flow duration with two graphs. The visual output is shown below.



A flow duration curve is a plot of discharge vs. % of time that a particular discharge was equaled or exceeded. The area under the flow duration curve (with arithmetic scales) gives the average daily flow, and the median daily flow is the 50% value.

Specification

Name `flowduration` - flow duration chart

Properties

Name	Description	Type	Required
`title`	chart title.	String	N

Sub elements

Name	Description	Type	Default	Occurrences
y	y axis	value column		+

Parent(s) `analysis`

Notes

- The y axis must be referencing a column with numerical values.
- Columns can have different number of rows.

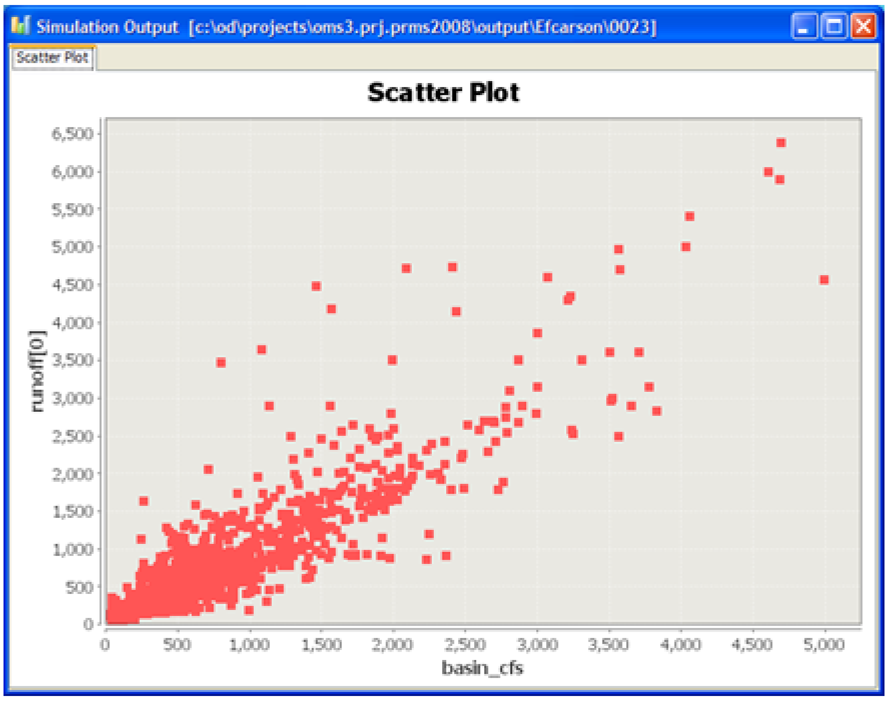
2.7.4. Scatter

Scatter plots show the relationship between two variables by displaying data points on a two-dimensional graph. They are useful in the early stages of analysis when exploring data before actually calculating a correlation coefficient or fitting a regression curve. For example, a scatter plot can help one to determine whether a linear regression model is appropriate.

The `scatter` element of an analysis provides for an easy creation of a scatter plot:

```
analysis {
  scatter {
    x(file:"%last/out1.csv", column:"basin_cfs")
    y(file:"%last/out1.csv", column:"runoff[0]")
  }
  ...
}
```

Provide x and y axis information accordingly. The setup above indicates the correlation of a simulated and observed property such as runoff



Name	`scatter` - scatter plot

Properties

Name	Description	Type	Required
`title`	chart title.	String	N

Sub elements

Name	Description	Type	Default	Occurrences
`x`	x axis variable	value column		1
`y`	y axis variable	value column		1

Parent(s) `analysis`

Notes

- The x and y axis must be referencing a column with numerical values.
- Columns must have the same number of rows.

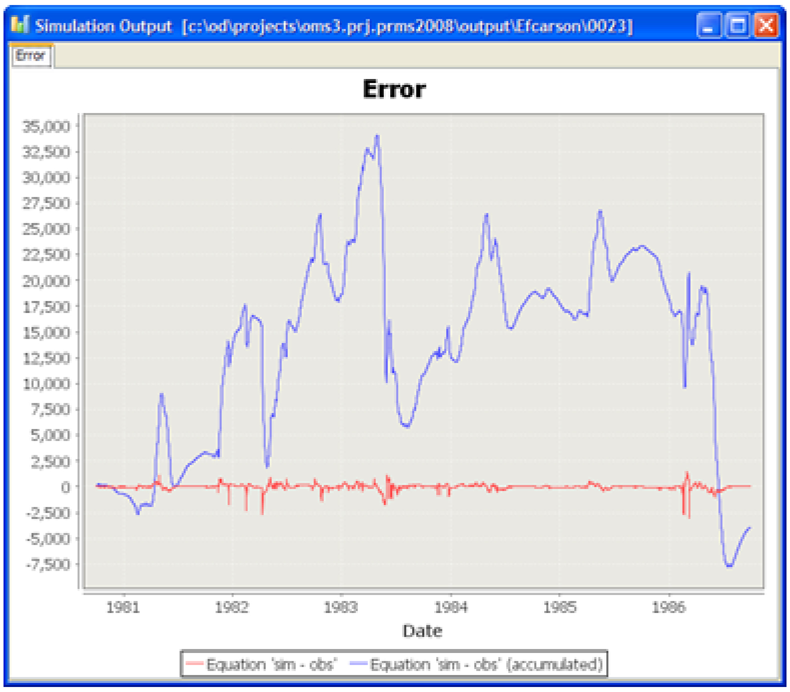
2.7.5. Computed Ad-hoc graphs

The previous examples are fully based on column data obtained from data files. However sometimes an derived data set should be used instead on an ad-hoc base, for example to plot the sum of the difference of two data sets without creating computing this value in the model or manipulate the date sets with other tools. Another example would be the ad-hoc creation of mass balance term.

The `calc` element was introduced to support the creation of arbitrary derived data sets for all analysis types above. It allows the specification of an user defined equation, that operates on all elements of data columns. An example:

```
analysis {
   timeseries(title:"Error") {
     x(file:"%last/out1.csv",  column:"date")
     calc(eq:"sim - obs") {
         sim(file:"%last/out1.csv", column:"basin_cfs")
         obs(file:"%last/out1.csv", column:"runoff[0]")
     }
  ...
}
```

The `calc` element in the example above is used as y axis. It has two major parts that relate to each other. First, the `eq` attribute takes a user defined term as string argument, in this case: `"sim - obs"`. The names are user defined, and they have to match the names that are used for column data that is enclosed in the `calc` element. "`sim(...)`, and '`obs(...)`' are two definitions of data columns with arbitrary names. As a rule: Any name being used in `eq` should occur as element within `calc`. Any complex term can be created in `eq` and `calc` can contain an arbitrary number of column references. As a result, the `timeseries` plot will have the element-wise diff of basin_cfs and runoff shown as y axis.



The graph above shows two calculated data sets. The second definition accumulates the calculated value of an element over the data set. The `accu` flag is set to true.

```
analysis {
    timeseries(title:"Error") {
           x(file:"%last/out1.csv",  column:"date")
           calc(eq:"sim - obs") {
               sim(file:"%last/out1.csv", column:"basin_cfs")
               obs(file:"%last/out1.csv", column:"runoff[0]")
```

```
            }
            calc(eq:"sim - obs", acc:true) {
                sim(file:"%last/out1.csv", column:"basin_cfs")
                obs(file:"%last/out1.csv", column:"runoff[0]")
            }
    }
}
```

Specification

Name `calc` - calculate data sets ad-hoc

Properties

Name	Description	Type	Required
`title`	chart title.	String	n
`eq`	the equation	String	y

Sub elements

Name	Description	Type	Default	Occurrences
`<name>`	data set variable	value column		+

Parent(s) `timeseries, scatter, flowduration`

Notes

- The sub elements must be referencing a column with numerical values.
- Columns must have the same number of rows.
- names for sub elements must be unique.

3. Basic Simulation (`sim`)

A standard simulation is the basic method to setup and run a model. It is being created using the `sim` element of the `SimBuilder` class. Define the package name and list imported packages as shown below first. The package `oms3` contains `SimBuilder` and needs to be imported at a minimum (1).

```
sim(name:"SimpleTWModel for EF") {
   // define the model
   model(classname:"tw.Thornthwaite") {
       // add parameter
       parameter {
           climateFile  "c:/od/projects/ngmf.models/src/tw/climate.cst"
           outputFile   "output.csv"
           runoffFactor 0.5
           latitude     35.0
           smcap        200.0
       }
   }
}
```

Line (2) creates the `SimBuilder` object that is being used in (3) to construct the simulation type. (Other simulations as described in the following sections are using different names here. The `sim` object has the property `name` to identify it and its purpose. In general properties are key/value pairs, separated by colons. Multiple properties are separated by comma, all properties are embedded in parenthesis following the model.

Specification

Name `sim` - defines a basic simulation.

Properties

Name	Description	Type	Required
`name`	the name of the simulation	String	Y

Sub elements

Name	Description	Type	Default	Occurrences 1)
`model`	the model to execute	n/a		1
`outputstrategy`	output management	n/a	`StandardOutput`	?
`efficiency`	model efficiencies to be computed during simulation	n/a		*
`summary`	statistics summary to be computed during simulation	n/a		*
`resource`	simulation resource definition	n/a		*
`analysis`	post run analysis	n/a		1

Notes [TBD]

4. Ensemble Streamflow Prediction (`esp`)

ESP is a simulation type for Ensemble Streamflow Prediction. It implements is a modified version of the National Weather Service's ESP procedure (Day, 1985). ESP uses historic or synthesized meteorological data as an analogue for the future. These time series are used as model input to simulate future conditions.

The typical application of ESP is streamflow forecasting . The initial hydrological conditions of a watershed, for the start of a forecast period, are assumed to be those simulated by the model for that point in time. Typically, multiple hydrographs are simulated from this point in time forward, one for each year of available historic data. For each simulated hydrograph, the model is re-initialized using the watershed conditions at the starting point of the forecast period. The forecast period can vary from a few days to an entire year. A frequency analysis is then performed on the peaks and/or volumes of the simulated hydrograph traces to evaluate their probabilities of exceedance.

Initialization Period	Provide a start and end time for ESP initialization. This is period over which the model will be run prior to the forecast period. It should be long enough to run the model through one or more wetting and drying cycles.
Forecasting Period	Provide the end date for your forecasting period. (Note: The start date for the forecasting period follows the end date of the initialization period.)
Historical Years	Provide the historical years to be used for forecasting traces.
Model parameter files	Provide all the parameter files for the model. One of those files must have the ESP property set and content as described in the next section.

```
esp(name:"EFCarson") {

   // define output strategy: output base dir and
   // the strategy NUMBERED|SIMPLE|DATE
   outputstrategy(dir: "$work/output", scheme:NUMBERED)

   // for class loading: model location
   resource "$work/dist/*.jar"

   // define model
   model(classname:"model.PrmsDdJh") {
        // parameter
        parameter (file:"$work/data/efcarson/params.csv") {
            inputFile  "$work/data/efcarson/data.csv"
            outFile    "out.csv"
            sumFile    "basinsum.csv"
            out        "summary.txt"

            startTime "1983-10-01"
            endTime   "1984-09-30"
        }
    }

    // number of forecast days
    forecast_days 15

    // historical years for to be used for traces
    // years are inclusive
    first_year 1981
    last_year  1983
}
```

Specification

Name esp - defines a Ensemble Streamflow Prediction (ESP) simulation.

Properties

Name	Description	Type	Required
`name`	the name of the simulation	String	Y

Sub elements

Name	Description	Type	Default	Occurrences
`model`	the model to execute	-		1
`outputstrategy`	output management	-	`StandardOutput`	?
`resource`	simulation resource definition	-		*
`forecast_days`	number of forecast days	int	-	1 (if forecast_days not provided)
`forecast_end`	forecasting end date	ISO Date String		1 (if forecast_days not provided)
`first_year`	first historical year	int	-	1
`last_year`	last historical year	int	-	1

Notes

- Either the forcast_days or forecast_end has to be provided, if both are missing or both are provided an error message will be given and the simulation stops.

[TBD]

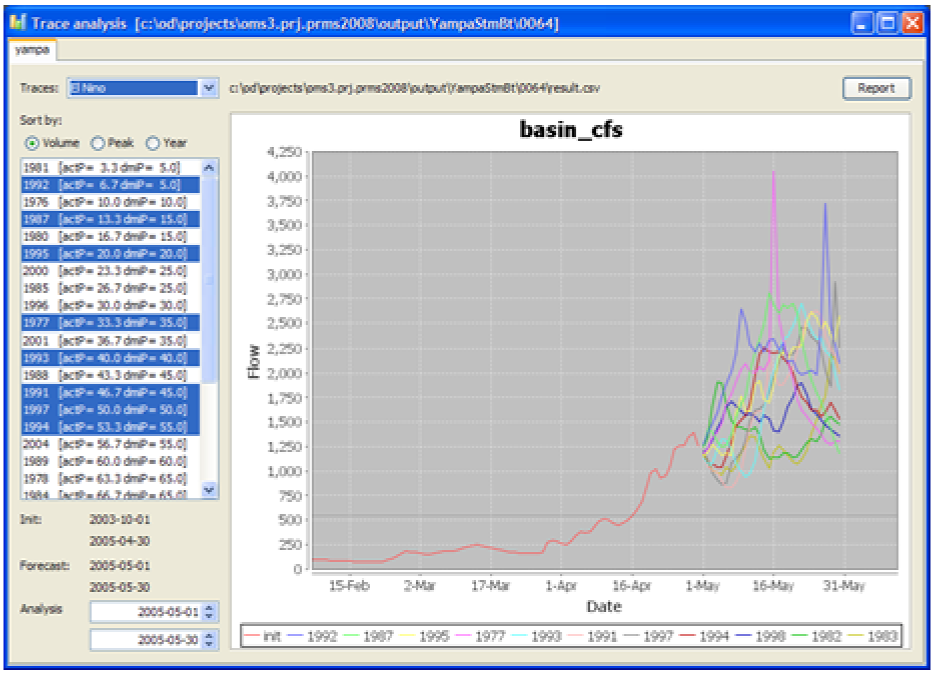
4.1. ESP Trace Analysis

The ESP procedure uses historical meteorological data to represent future meteorological data. Alternative assumptions about future meteorological conditions can be made with the use of synthesized meteorological data.

A few options are available in applying the frequency analysis. One option assumes that all years in the historic database have an equally likely probability of occurrence. This give equal weight to all years. Years associated with El Nino, La Nina, ENSO neutral, Pacific Decadal Oscillation (PDO) less than -0.5, PDO greater than 0.5, and PDO neutral have also been identified in the ESP procedure, and the years in these groups can be extracted separately for analysis. Alternative schemes for weighting user-defined periods, based on user assumptions or a priori information, are also being investigated.

[TBD] ...

Figure 4.2. ESP trace analysis


Trace analysis [c:\od\projects\oms3.prj.prms2008\output\YampaStmBt\0064]

yampa

Traces: El Nino   c:\od\projects\oms3.prj.prms2008\output\YampaStmBt\0064\result.csv   Report

Sort by: [✓ Volume] Peak Year

```
1981  [actP=  3.3 dmP=  5.0]
1992  [actP=  6.7 dmP=  5.0]
1976  [actP= 10.0 dmP= 10.0]
1987  [actP= 13.3 dmP= 15.0]
1980  [actP= 16.7 dmP= 15.0]
1995  [actP= 20.0 dmP= 20.0]
2000  [actP= 23.3 dmP= 25.0]
1985  [actP= 26.7 dmP= 25.0]
1996  [actP= 30.0 dmP= 30.0]
1977  [actP= 33.3 dmP= 35.0]
2001  [actP= 36.7 dmP= 35.0]
1993  [actP= 40.0 dmP= 40.0]
1988  [actP= 43.3 dmP= 45.0]
1991  [actP= 46.7 dmP= 45.0]
1997  [actP= 50.0 dmP= 50.0]
1994  [actP= 53.3 dmP= 55.0]
2004  [actP= 56.7 dmP= 55.0]
1989  [actP= 60.0 dmP= 60.0]
1978  [actP= 63.3 dmP= 65.0]
1984  [actP= 66.7 dmP= 65.0]
```

Init: 2003-10-01
2005-04-30
Forecast: 2005-05-01
2005-05-30
Analysis 2005-05-01
2005-05-30

The trace analysis is a part if the analysis element, and is simple to configure.

```
esp(name:"Yampa") {

  ...
  analysis(title:"Trace analysis") {


       // relative path name, last output
       esptraces(title:"yampa", dir:"%last", var:"basin_cfs")
   }

}
```

Name `esptraces` - analysis an ensemble run.

Properties

Name	Description	Type	Required
`title`	the title of the graph	String	n
`dir`	the esp output directory, containing all the trace outputs and the file '`result.csv`'.	String	y
`var`	the variable to trace	String	y

Notes • ..

4.2. References

Day, G.N., Extended streamflow forecasting using NWSRFS: J. Water Resour. Plan. and Manag. Am. Soc. Civ. Eng., 111, 157, 1985.

5. Luca Calibration (luca)

Luca (Let us calibrate) is a multiple-objective, stepwise, automated procedure for model calibration. The calibration procedure uses the Shuffled Complex Evolution global search algorithm to calibrate any OMS3 model. Luca defines a OMS simulation type for building and performing a procedure to calibrate parameters for a (hydrological) model. It integrates the following components:

- Multiple-objective, step-wise calibration
- Shuffled Complex Evolution (SCE), a global-search parameter optimization; and
- OMS model interoperability.

5.1. Shuffled Complex Evolution (SCE)

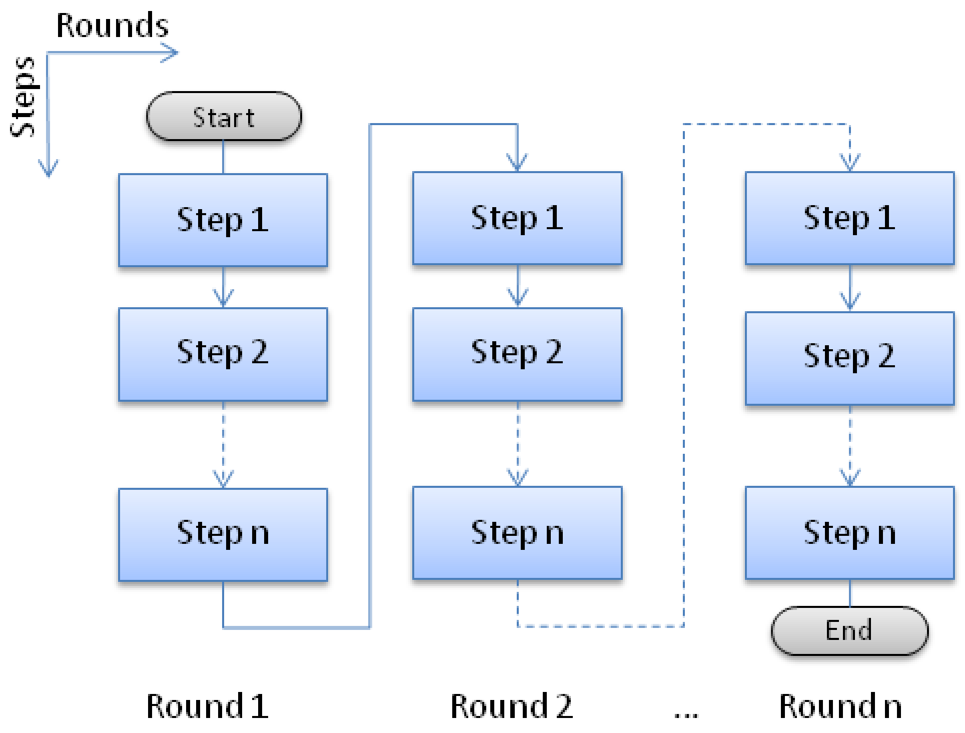
The purpose of Shuffled Complex Evolution (SCE) is to calibrate model parameters so that the model, which requires those parameters, gives better results. SCE consists of the following steps:

1. **Generating points**. The set of parameters to be calibrated is considered as a point in N dimension space where N is the number of parameters. SCE generates many points, in which each parameter has a random value within its lower and upper bound values.
2. **Assigning criterion values.** The model is run with every point (a set of parameters) generated in SCE Step 1 as an input. An objective function that determines how close the simulation results are to observed values is used to calculate a criterion value for each point.
3. **Creating complexes.** The points are divided into smaller groups called complexes such that points of good and bad criterion values are equally distributed.
4. **Complex evolution.** Each complex is evolved in the following way: Several points are selected from the complex to construct a sub-complex. In the sub-complex, a new point is generated, and a point that has a bad criterion value is replaced with this new point. This evolution step is repeated several times with different random points in a sub-complex.
5. **Combining complexes.** All points in the complexes are combined together to be one group.
6. **SCE Steps (3) – (5) are called a shuffling loop**. It is repeated until the results of the complex evolution meet one of the following end conditions:
   - The number of model executions reaches the maximum number of model execution
   - The percent change in the best criterion value of the current shuffling loop and that of several shuffling loops before is less than a specified percentage.

The points converge into a very small region, which is less than 0.1% of the space within the lower and upper bounds of parameters. The number of complexes used in SCE Step 3 decreases by 1 for every shuffling loop. This decrease stops when the number of complexes reaches the **minimum number of complex required**. The output is the parameter file containing the point (a parameter set) that has the best criterion value.

Luca Rounds and Steps

Figure 4.3. Rounds and Steps in Luca



In the multi-step calibration technique, a step and a round are defined as follows:

Steps	A step is associated with a parameter set, which contains one or more parameter values.
Rounds	A round consists of one or more steps.

[TBD]

Specification - `luca{}`

Name `luca` - defines a Luca calibration simulation.

Properties

Name	Description	Type	Required
`name`	the name of the simulation	String	y

Sub elements

Name	Description	Type	Default	Occurrences
`model`	the model to execute	model {}		1
`outputstrategy`	output management	outputstrategy {}	`StandardOutput`	?
`resource`	simulation resource definition	model {}		*

Name	Description	Type	Default	Occurrences
`calibration_start`	start date of calibration	ISODate String	-	1
`rounds`	number of rounds	int	1	?
`step`	calibration step definition	step {}	-	+

Notes

- ....

Specification - `step{}`

Name `step` - defines a single Luca calibration step.

Properties

Name	Description	Type	Required
`name`	the name of the step	String	n

Sub elements

Name	Description	Type	Default	Occurrences
`parameter`	parameter to calibrate	-	-	1
`optimization`	optimization definition	-	-	1
`max_exec`	maximum # executions in one step	int	10000	?
`init_complexes`		int	-	?
`points_per_complex`		int	-	?
`points_per_subcomplex`		int	-	?
`evolutions`		int	-	?
`min_complexes`		int	-	?
`shuffling_loops`		int	5	?
`of_percentage`		double	0.01	?

Notes

- If the name of the step is missing, it will be numbered instead.

Name `optimization` - defines optimization parameter.

Properties

Name	Description	Type	Required
`simulated`	the simulated variable name	String	y
`observed`	the observed variable name	String	y

Sub elements

Name	Description	Type	Default	Occurrences
`of`	objective function definition	of {}	-	+

Notes

- ....

Name `of` - defines an objective function.

Properties

Name	Description	Type	Required
`method`	the objective function	`NS`|`RMSE`|`ABSDIF`|`LOGABSDIF`|`PMCC` 1)	y
`timestep`	the time step for simulated and observed values	`DAILY`	n (default: `DAILY`)
`weight`	the objective function weight	double (`0 - 1.0`)	n 2)

Notes

- 1) If the method name is not one of the constants above, it is assumed to be the name of a user defined Java class that (i) implements the `oms3.ObjectiveFunction` interface, and (ii) is available on the `CLASSPATH`. This was a modeler can implement custom objective function(s) and use the in a simulation.
- 2) If the weight is not specified, all provided objective functions will be equally weighted. If specified, it has to be specified for **all** objective functions. The user has to ensure that the weights sum up to `1.0` for all objective functions.

Specification - step

Name `parameter` - defines model parameter to calibrate

Properties

Name	Description	Type	Required
`lower`	the lower boundary	double	y
`upper`	the upper boundary	double	y
`strategy`	the calibration strategy	`MEAN`|`VALUES`|`BINARY`	n (default: MEAN)

Notes

- Only MEAN is implemented at the moment.

5.2. References

SCE Related Papers

Duan, Q., Sorooshian, S. and Gupta, V.K., (1992). Effective and efficient global optimization for conceptual rainfall-runoff models. Water Resources Research 28 (4), 1015-1031.

Duan, Q., Sorooshian, S. and Gupta, V.K., (1993). A Shuffled Complex Evolution approach for effective and efficient global minimization. J. of Optimization Theory and its Applications, 76 (3), 501-521.

Duan, Q., Sorooshian, S. and Gupta, V.K., (1994). Optimal use of the SCE-UA global optimization method for calibrating watershed models. Journal of Hydrology, 158 265-284

Step-Wise, Multiple-Objectiveu Calibration Related Papers

Hay, L.E., Leavesley, G.H., Clark, M.P., Markstrom, S.L., Viger, R.J., and Umemoto, M. (2006). Step-wise, multiple-objective calibration of a hydrological model for a snowmelt-dominated basin. Journal of the American Water Resources Association.

Hay, L.E., Leavesley, G.H., and Clark, M.P., (2006). Use of Remotely-Sensed Snow Covered Area in Watershed Model Calibration for the Sprague River, Oregon. Joint 8th Federal Interagency Sedimentation Conference and 3rd Federal Interagency Hydrologic Modeling Conference, Reno, Nevada, April, 2006.

Others

Leavesley, G.H. and L.G. Stannard, (1995). The precipitation-runoff modeling system- PRMS. In: Computer Models of Watershed Hydrology, Water Resources Publications, Highlands Ranch, CO, edited by V.P Singh, Chapter 9, 281-310.

Leavesley, G.H., Restrepo, P.J., Markstrom, S.L., Dixon, M., and Stannard, L.G., (1996). The modular modeling system - MMS: User's manual: U.S. Geological Survey Open File Report 96-151, 200 p.

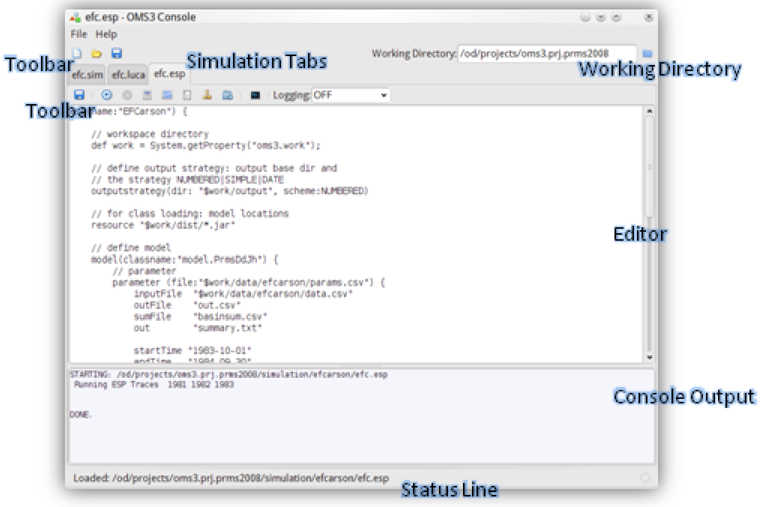
6. Automated Model and Component Testing (`test`)

[TBD]

Chapter 5. The Modeling Console

The OMS3 Modeling Console is a graphical user interface for the OMS3 Modeling Framework. It provides simple access to framework core features such as simulation management, output analysis, or documentation generation. Using the OMS3 Console is one way to interact with the framework. Others methods are the integration of OMS3 into Integrated Development Environments (IDEs) or custom applications for model applications. The main purpose of the OMS3 Console is to allow a modeller a straightforward and simple tool to develop a a model and a simulation, run the simulation, provide for parameter editing and offer ad-hoc post run analysis and visualization. Figure ??? provides an overview of the OMS3 Console.

Figure 5.1. The OMS3 Console



```
name:"EFCarson") {

    // workspace directory
    def work = System.getProperty("oms3.work");

    // define output strategy: output base dir and
    // the strategy NUMBERED|SIMPLE|DATE
    outputstrategy(dir: "$work/output", scheme:NUMBERED)

    // for class loading: model locations
    resource "$work/dist/*.jar"

    // define model
    model(classname:"model.PrmsDdJh") {
        // parameter
        parameter (file:"$work/data/efcarson/params.csv") {
            inputFile  "$work/data/efcarson/data.csv"
            outFile    "out.csv"
            sumFile    "basinsum.csv"
            out        "summary.txt"

            startTime "1983-10-01"
```

```
STARTING: /od/projects/oms3.prj.prms2008/simulation/efcarson/efc.esp
 Running ESP Traces  1981 1982 1983

DONE.
```

Loaded: /od/projects/oms3.prj.prms2008/simulation/efcarson/efc.esp

The console can be started directly from the OMS download site at `http://oms.javaforge.com`, by clicking the **Launch** button. It is provided as a Java Webstart application that integrates itself into the client's desktop after installation. As the only prerequisite, a user is required to install the JDK version 1.6 or greater on the client.

The OMS3 Console is a self signed Webstart application. On initial launch it will prompt the user to run the console, even without certificate validation. Accept this and click on **Run** the console anyway.

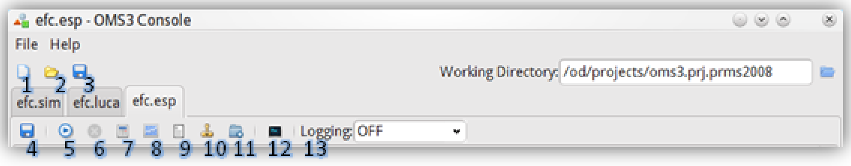
The Console's principal user interface is shown in Figure ???. Many of the user interface elements operate as known from many other applications (File|Open, File | Save, ...), the specific console user interface elements are highlighted in Figure ???

Working Directory	The working directory sets the base directory and system property `oms3.work` for a simulation. It defines a workspace for the simulation. If this folder contains the file `oms3.conf`, its content gets loaded and applied for all simulation. For example if the file oms3.conf contains a list of open files, they will be opened after setting this directory. The directory can be set using the button next to the directory name.
Simulation tabs	Each tab contains one simulation script, tool bar, and associated console output. The console might have multiple simulations open at one time, however they must originate from the same workspace. Each simulation tabs operates independent from each other, means, you can execute one simulation, while editing another one in a different tab.
Toolbars	The console has two tool bars. A global tool bar manages a set of simulations. In addition, each simulation has its own tool bar, allowing for different operations such as executing, documenting, etc a simulation. The tool bar action are explained in detail below.

Editor	The simulation editor allows creating and editing a simulation file.
Console Output	This read/only output area, shows all standard and error output from the simulation run.
Status Line	Some informative message during console use.

Each simulation tab contains its own private tool bar (Figure ???). The tool bar actions are always directed towards the current simulation script in the editor

Figure 5.2. Tool Bars



1. Creates a new, empty simulation.
2. Opens an existing simulation from `<working directory>/simulations`.
3. Saves all open simulations.

4. Saves the script to a file. If the file is new, it will prompt for a name.
5. Runs the simulation script.
6. Interrupts and stops a running simulation. If no simulation is running this button is disabled.
7. Opens the parameter editor with parameter being loaded from the current simulation. The parameter editor is explained below.
8. Executes the `analysis` part of a simulation. This will usually result in an new window containing graphs and plots. The analysis window is explained in more detail below.
9. Creates Docbook5 documentation of the simulation and stores it into the current output folder.
10. Creates a `SHA` digital signature of the simulation and prints the simulation fingerprint to the output window.
11. Opens the last output folder using the operating system's file explorer.
12. Clears the console output for this simulation.
13. Logging setting. Define the log level here for your simulation, this will result in more or less verbose output during simulation execution.

Each tab has a context menu that is accessible with a right mouse click. It allows saving, saving under a different name, closing, and other operations with respect to the selected tab.

1. OMS3 Project Workspace

An OMS3 workspace is a recommended but not required directory layout. It is used to store and operate all resources that belong to a simulation such as model components, climate data files, parameter files, simulations, output data, documentation. It represents best practice file management and is result of many realized simulation projects within OMS3. It also complements the folder layout of many Integrated Development Environments, so an IDE and OMS3 can share the same project or working directories.

The working directory contains at least the folders as shown below with suggested content. This is sufficient to use the model for simulation runs.

```
<Working Directory>
|   oms3.conf
+-- simulations      (*.sim, *.luca, *.esp, ...)
+-- data             (*.csv)
+-- output           (*.csv)
+-- dist             (*.jar, *.dll, *.exe)
```

The `<Working Directory>` is the root of the project workspace. In the console, it is being set using the user interface. This folder name is passed to the simulation as a Java system property `"oms3.work"`, that can be read by the simulation and the model components. The directories have the following content and meaning:

`simulations`	contains simulation files or other scripts. Simulation files execute a model with input data and produce output. Simple simulations, or scripts for model calibration, uncertainty or sensitivity analysis can be stored here.
`data`	contains input data to the model, usually climate data or parameter sets. Those file can be OMS3 `csv` data files or any other data file format.
`output`	this folder will contain the simulation output data after each run. Usually it will have sub-folders, that have the name of the simulation and further sub folders for each run if output is versioned. See `<outputstrategy>` for further details.
`dist`	The dist folder contains all executable code for a simulation run. These are usually platform independent componet/model `*.jar` file(s), or platform dependent DLLs (`*.dll`) or executables (`*.exe`), and others.

All the folders above may have sub-folders to store data/code that is for example organized by watershed or other application area.

It is recommended that the project workspace contains the folders as shown above. The folder organize the different file types of a project. The project root folder can contain the file `oms3.conf`.

The workspace can also accommodate for component code development. Add for example a source (`src`) folder that will contain the component / model source files. The build process (e.g. managed by an IDE) will build the executable and store it in the `dist` folder.

```
<Working Directory>
|   oms3.conf
+-- simulations      (*.sim, *.luca, *.esp, ...)
+-- data             (*.csv)
+-- output           (*.csv)
+-- dist             (*.jar, *.dll, *.exe)
+-- src              (*.java, *.f90, *.c, ..)
```

`src`	The src folder contains sources, if the workspace is also being used to develop/test components and models. Source files are Java files (`*.java`), FORTRAN files (`*.for,..`), or C/C++ files (`*.c, *cpp,...`), just to mention some examples.

In addition to the listed workspace folders, there can be other sub folders needed by IDEs and other development tools. Note that of a folder name is case sensitive when used in a simulation script.

1.1. Workspace configuration file '`oms3.conf`'

The file oms3.conf is an optional file located in a project working directory. It can be used to provide for project specific settings to be used by the OMS3 console. The console is usually creating such a file when a working directory is set. A user can also edit this file and all custom setting for model execution. The file oms3.conf is a text file that has "key-value-pairs" of configuration information, separated by lines.

`oms3.conf` Example:

```
#OMS3 Console Project Configuration
#Mon Apr 09 10:11:09 MDT 2010
open.files=simulation/efcarson/efc.sim;simulation/efcarson/efc.esp
jvm.options=-Xms128M -Xmx256M
```

The example above shows the some project settings. The entry `open.files` lists all files relative to the working directory that should be opened when the workspace is set in the console. The files names are separated by semi-colon.

The entry`jvm.options` contains as value additional options for the JVM to be used when executing a simulation. This can be any valid JVM argument. Usually this option tweaks memory provisioning for the simulation, or garbage collection adjustments. The example above provides 128 MB of memory initially to the simulation and sets the maximum available memory to 265 MB. The entry `jvm.options` should be managed manually by the user.

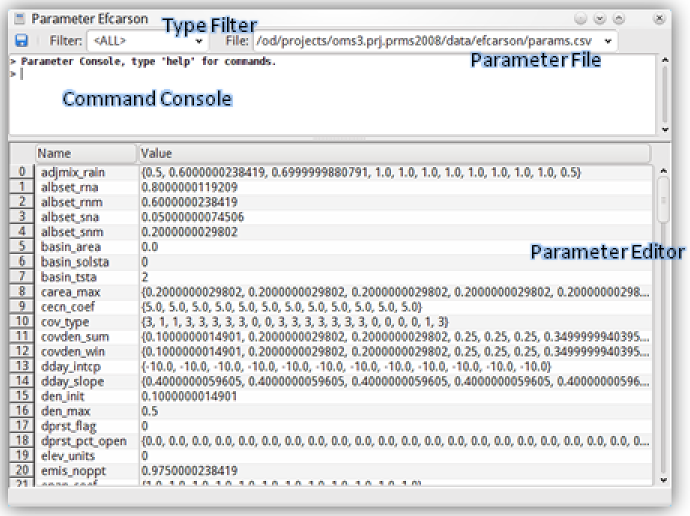
2. Parameter Editor

The Parameter Editor is a visual tool for parameter editing that is embedded in the OMS3 console. It can be started by clicking on the appropriate button in the tool bar of a simulation tab. It allows for a presentation and filtering of parameter values, bulk editing of values, statistics, and basic spread sheet like operations.

The editor works on parameter files that adhere to the OMS3 `csv` parameter file format. Such files can be easily edited with other external tools such as Excel, however the parameter editor provides an easy to use interface to manipulate parameter values numerically.

Note that only parameter files that are references in a simulation using the `.. parameter(file:'<name>') ...` construct can be edited this way. Parameter that are in-lined within the parameter section cannot be presented and edited here.

Figure 5.3. Parameter Editor



```
> Parameter Console, type 'help' for commands.
> |
```

	Name	Value
0	adjmix_rain	{0.5, 0.6000000238419, 0.6999999880791, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5}
1	albset_rna	0.8000000119209
2	albset_rnm	0.6000000238419
3	albset_sna	0.05000000074506
4	albset_snm	0.2000000029802
5	basin_area	0.0
6	basin_solsta	0
7	basin_tsta	2
8	carea_max	{0.2000000029802, 0.2000000029802, 0.2000000029802, 0.2000000029802, 0.2000000298...
9	cecn_coef	{5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0}
10	cov_type	{3, 1, 1, 3, 3, 3, 3, 3, 0, 0, 3, 3, 3, 3, 3, 3, 0, 0, 0, 0, 1, 3}
11	covden_sum	{0.1000000014901, 0.2000000029802, 0.2000000029802, 0.25, 0.25, 0.25, 0.3499999940395...
12	covden_win	{0.1000000014901, 0.2000000029802, 0.2000000029802, 0.25, 0.25, 0.25, 0.3499999940395...
13	dday_intcp	{-10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0}
14	dday_slope	{0.4000000059605, 0.4000000059605, 0.4000000059605, 0.4000000059605, 0.4000000596...
15	den_init	0.1000000014901
16	den_max	0.5
17	dprst_flag	0
18	dprst_pct_open	{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0...
19	elev_units	0
20	emis_noppt	0.9750000238419

Figure ??? shows the Parameter Editor with parameter content. The layout of the window is organized as follows.

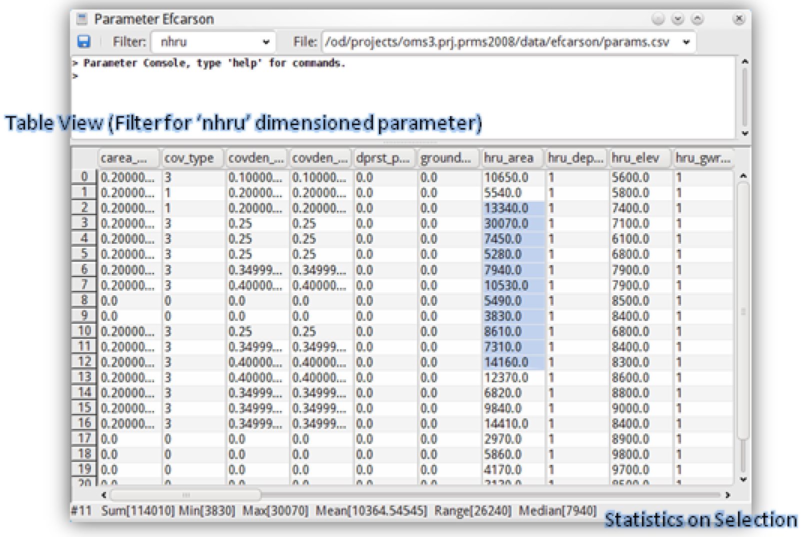
Parameter File
: This combo box shows the active parameter file for editing. When opened it lists all parameter files as specifies in the simulation file. Changing the combo box selection will cause the while view to change content.

Filter
: The Filter combo box allows to switch views on parameters. They can be filtered by dimension, by scalars, or all parameter can be presented as sown in this figure. The selection change in this combo box will cause the editor values to change.

Command Console
: The command line on this console allows the (bulk) manipulation of parameter data shown below. This command line has a history, a help function and offers a powerful method of editing parameter values. It is further explained below.

Parameter Editor
: This table shows the parameter names and values. The layout might change when different filter are applied. It has in-place editing capabilities, by clicking on a cell, it value can be edited and changed.

Figure 5.4. Parameter Editor with applied filter



Parameter Efcarson

Filter: nhru   File: /od/projects/oms3.prj.prms2008/data/efcarson/params.csv

```
> Parameter Console, type 'help' for commands.
>
```

	carea_...	cov_type	covden_...	covden_...	dprst_p...	ground...	hru_area	hru_dep...	hru_elev	hru_gwr...
0	0.20000...	3	0.10000...	0.10000...	0.0	0.0	10650.0	1	5600.0	1
1	0.20000...	1	0.20000...	0.20000...	0.0	0.0	5540.0	1	5800.0	1
2	0.20000...	1	0.20000...	0.20000...	0.0	0.0	13340.0	1	7400.0	1
3	0.20000...	3	0.25	0.25	0.0	0.0	30070.0	1	7100.0	1
4	0.20000...	3	0.25	0.25	0.0	0.0	7450.0	1	6100.0	1
5	0.20000...	3	0.25	0.25	0.0	0.0	5280.0	1	6800.0	1
6	0.20000...	3	0.34999...	0.34999...	0.0	0.0	7940.0	1	7900.0	1
7	0.20000...	3	0.40000...	0.40000...	0.0	0.0	10530.0	1	7900.0	1
8	0.0	0	0.0	0.0	0.0	0.0	5490.0	1	8500.0	1
9	0.0	0	0.0	0.0	0.0	0.0	3830.0	1	8400.0	1
10	0.20000...	3	0.25	0.25	0.0	0.0	8610.0	1	6800.0	1
11	0.20000...	3	0.34999...	0.34999...	0.0	0.0	7310.0	1	8400.0	1
12	0.20000...	3	0.40000...	0.40000...	0.0	0.0	14160.0	1	8300.0	1
13	0.20000...	3	0.40000...	0.40000...	0.0	0.0	12370.0	1	8600.0	1
14	0.20000...	3	0.34999...	0.34999...	0.0	0.0	6820.0	1	8800.0	1
15	0.20000...	3	0.34999...	0.34999...	0.0	0.0	9840.0	1	9000.0	1
16	0.20000...	3	0.34999...	0.34999...	0.0	0.0	14410.0	1	8400.0	1
17	0.0	0	0.0	0.0	0.0	0.0	2970.0	1	8900.0	1
18	0.0	0	0.0	0.0	0.0	0.0	5860.0	1	9800.0	1
19	0.0	0	0.0	0.0	0.0	0.0	4170.0	1	9700.0	1

#11 Sum[114010] Min[3830] Max[30070] Mean[10364.54545] Range[26240] Median[7940]

Figure ??? shows the parameter editor with an applied dimension filter. The table presents all parameter that are bound to a specific dimension. Selecting cells in the table will print basic statistics about those on the bottom of the window. Any area can be selected, a column selection will print statistics about the whole column.

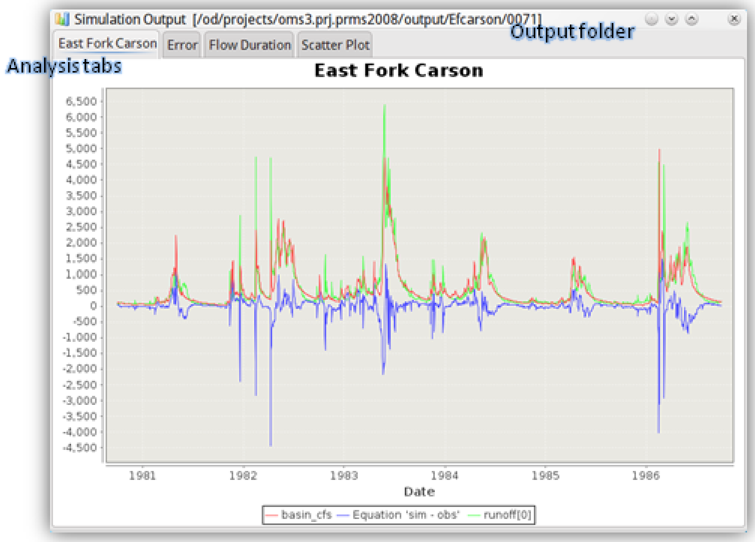
2.1. Parameter Editor Console

The Parameter Editor contains a command line console at the upper part of the window. It allows the

3. Analysis Output

The analysis output window provides ad-hoc result information as defined in the analysis section of a simulation. It is executed by pressing the Analysis Button in the tool bar in a simulation tab. A analysis window might look like Figure ???

Figure 5.5. Analysis Output



The tile bar shows the output folder that is used to perform the output data analysis. Depending on the setting in `<outputstrategy>`, such output can be versioned or unversioned. Either way, the **last** output folder of a simulation run is presented.

The `analysis` section of a simulation files allows the definition of plots such as `timeseries`, `error` plots, `scatter` plots, `esp-trace` analysis plots, and others. Each of these plots will appear as a tab in the analysis window, the plot title shows as a tab name.

A part of the analysis section used to create the plots in Figure ??? is shown below.

```
...
analysis(title:"Simulation Output") {
      timeseries(title:"East Fork Carson", view: COMBINED) {
          x(file:"%last/out1.csv", column:"date")
          y(file:"%last/out1.csv", column:"basin_cfs")
          calc(eq:"sim - obs") {
              sim(file:"%last/out1.csv", column:"basin_cfs")
              obs(file:"%last/out1.csv", column:"runoff[0]")
          }
          y(file:"%last/out1.csv", column:"runoff[0]")
      }
      timeseries(title:"Error", view: MULTI ) {
        ...
```

It is clear how the `timeseries` section content maps to the visual view. Note the the variable `%last` references the last output folder of a simulation run. The plot also has a calculated column.

Note that an analysis section of a simulation is only being used when invoked. A regular simulation run does not perform the data analysis.

Chapter 6. Advanced Techniques

In this chapter, several advances framework aspects

1. Simulation Traceability and Audit Support

Managing and tracking the simulation process is as just as important as the simulation principles and methods themselves. Authorities who use simulations for predictions, and forecasts that effect the public are required to manage the trail of resources that where used to produce a particular prediction. At any given time and on (legal) requests a simulation output must be reproduced by recalling and recreating the simulation conditions at that time.

Simulation traceability is a core feature of OMS3. There are several aspects that should be combined to provide for a secure solution for output audit trails.

- A *Version Control System* (VCS) for software version tracking must be used to host and version all model resources such as source components, parameter and data files, and the simulation file in a repositories. All files must carry the `VersionInfo` and `SourceInfo` annotation (for source code), and the corresponding meta data entries for `csv` files. If supported, the VCS should be setup to support keyword substitution on source files.
- Any OMS simulation can generate a *Secure Hash Algorithm digest (SHA digest)* of all involved simulation resources. As a default algorithm SHA-256 is being used, however is could be replaced (by setting the system property '`oms3.digest.algorithm`') with a stronger hash function (SHA-384 or SHA-512). A simulation digest is therefor a secure hash that is unique to a simulation configuration. It is being computed before the simulation starts.

  The is being applied in different ways:

  - If computed (`simu` parameter is set `digest:true`) the simulation sets the system property `oms3.digest` to the digest value. Any component in the model can now access this system property, for example an output component might carry this value into a generated output file.

    An fragment of an output component writing the Digest as text to a file `file` might look like this:

    ```
    @Intitialize
    void init() {
        ...
        String v = System.getProperty("oms3.digest");
        if (v != null) {
            file.println(" Digest," + v);
        }
        ...
    }
    ```

  - The simulation method `digest()` can be invoked to create a digest record creating the `tuple (<digest>, <simulation resources>)`. As default, this call prints the digest and the `SourceInfo`/`VersionInfo` records of all components/datafiles of that simulation to the console. Such information might be further processed for inclusion in a secure store (e.g. data base):

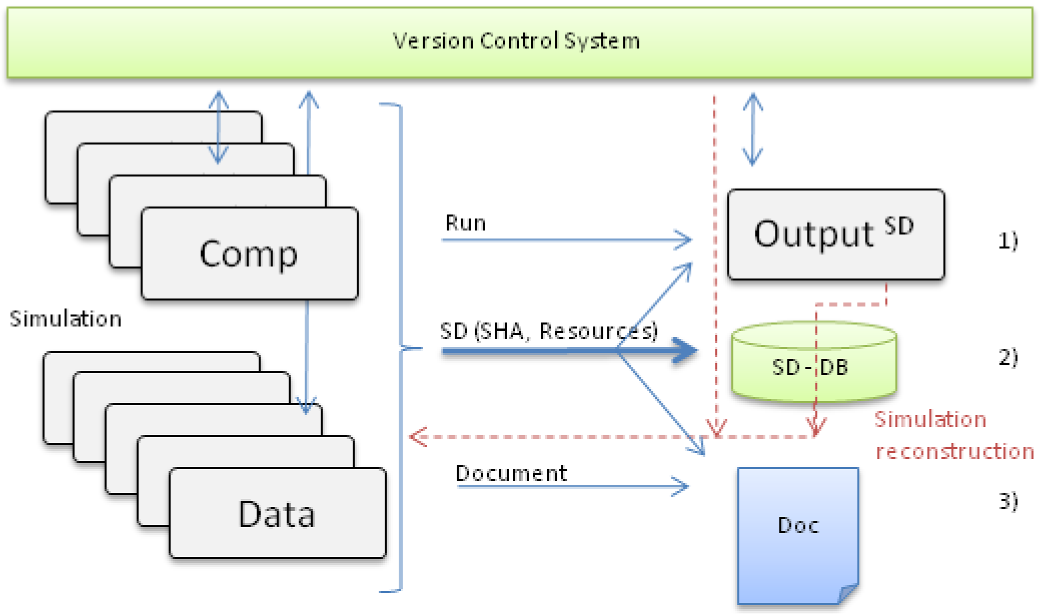
    ```
    println sim.digest()

    6b2d424e12626c65b6f9e3136de735703d405d4f207395797d9f9b4487a23e7f
       tw.Climate &  &
       tw.Daylen &  &
       tw.HamonET &  &
       tw.Output &  &
       tw.Runoff &  &
       tw.Snow &  &
       tw.SoilMoisture &  &
    >
    //// update the above!!!!!!
    ```

The example above shows the simulation digest record for the `Thornthwaite` model. The first line contains the digest, followed by the simulation resources.

- There a other infrastructure pieces required such as a data base for digest storage and retrieval.

The principal architecture for simulation traceability in OMS3 is shown in the Figure ???. It illustrates the integration with a version control system for resource tracking and a data base for digest tuple tracking.

Figure 6.1. Simulation Traceability Schematic



OMS3 provides a **core** simulation digest infrastructure that can be complemented with 'inverse' tools allowing for example (i) the automatic recreation of simulation configurations based on output data sets using simulation digest records, or (ii) the validation of output records against a set of 'certified' simulations in a master data base. As a result any model integrated in OMS3 can benefit from this integrity ensuring feature.

1.1. References

- SHA Hash functions [http://en.wikipedia.org/wiki/SHA_hash_functions]
- JCA Reference guide [http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html]

2. Digitally Signing Models

Once a model is deployed as a jar file, it can be digitally sign with an electronic *signature*. A digital signature ensures the integrity of the developed simulation. Once signed a simulation jar cannot be altered: Components, default parameter settings, and the model cannot be switched or patched. The signature protects the investment to develop a complex simulation setup. However you can overwrite public parameter values, an can control output generation.

2.1. Creating a self signed certificate

You have to have install the Java SDK from http://java.sun.com/j2se/downloads.html; the tools have to be in your path

Step 1 - Create a key:

```
 keytool -alias keyname -genkey
```

This will create a new keystore (usually `$HOME/.keystore`) if not present or add to it.

Step 2 - Sign the jar file using the key - Sign sim.jar:

```
 jarsigner sim.jar keyname
```

2.2. Importing an issued certificate

An issued certificate from a certification authority can be imported:

```
$ keytool -import -alias od  -file OD.cer
```

2.3. Validating the Integrity of a Simulation

Once a Simulation jar is signed it gets verified on execution:

```
$ java -jar EFC.jar
```

If verification fails, execution will abort. You can also verify the Simulation Jar with the `jarsigner` tool. You can also verify the Simulation Jar with the jarsigner tool

```
$ jarsigner -verify  EFC.jar
jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six months.

Re-run with the -verbose and -certs options for more details.
```

To get more details on certain component signatures:

```
$ jarsigner -verbose -certs -verify  EFC.jar
...

smk     1536 Thu May 24 10:17:22 MDT 2007  gov/usgs/prms/PrecipKrig.class

      X.509, CN=Olaf David, OU=CSU, O=CSU, L=FC, ST=CO, C=US (od)
      [certificate will expire on 1/14/09 10:44 AM]

smk     1593 Thu May 24 10:17:22 MDT 2007 gov/usgs/prms/Obs.class

      X.509, CN=Olaf David, OU=CSU, O=CSU, L=FC, ST=CO, C=US (od)
      [certificate will expire on 1/14/09 10:44 AM]
...

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope
```

These examples show different levels of protecting a model jar using digital signatures

The Jar deployment of OMS model is easy to perform and has the following features and benefits. Simulations can be packaged into an executable simulation jar. The simulation jar contains all code, data, and resources as defined within the IDE to run the simulation. To run a simulation only a Java Runtime Environment and the Simulation Jar are needed, no OMS installation is required. Simulation jars are sealed and can be digitally signed. Therefore a deployed simulation jar is secure and cannot be compromised. Simulation jars carry all information about the origin of resources that make up this simulation such as the version of components, data sets and the model. All sources can be traced back, if version info is present.

3. Documenting Simulations

Component Documentation can be generated from sources or binary classes.

```
package prms2008;

import java.util.Calendar;
import java.util.logging.*;
import oms3.annotations.*;
import static oms3.annotations.Role.*;

@Description
    ("Potential ET - Jensen Haise." +
    "Determines whether current time period is one of active" +
    "transpiration and computes the potential evapotranspiration" +
    "for each HRU using the Jensen-Haise formulation.")
@Author
    (name= "George H. Leavesley", contact= "ghleavesley@colostate.edu")
@Keywords
    ("Evapotranspiration")
@Bibliography
    ("Leavesley, G. H., Lichty, R. W., Troutman, B. M., and Saindon, L. G., 1983, "+
     "Precipitation-runoff modeling " +
     "system--user's manual: U. S. Geological Survey Water Resources Investigations " +
     "report 83-4238, 207 p.")
@VersionInfo
    ("$Id: PotetJh.java 390 2009-09-01 19:56:07Z ghleavesley $")
@SourceInfo
    ("$URL: http://svn.javaforge.com/svn/oms/branches/oms3.prj.prms2008/src/prms2008/PotetJh.java
@License
    ("http://www.gnu.org/licenses/gpl-2.0.html")
@Documentation
    ("file:/C:/od/projects/ngmf.models/src/prms2008/PotetJh.xml")
@Status
    (Status.TESTED)

public class PotetJh  {

    // private fields
    double[] tmax_sum;

    // "Indicator for whether within period to check for beginning of transpiration, 0=no, 1=yes.
    int[]    transp_check;
    int[]    transp_end_12;

    // Input params
    @Role(PARAMETER)
    @In public int nhru;
    @Role(PARAMETER)
    @In public int nsol;

    @Role(PARAMETER)
    @Description("HRU area ,  Area of each HRU")
    @Unit("acres")
    @In public double[] hru_area;

    @Role(PARAMETER)
    @Description("Monthly air temp coefficient - Jensen-Haise Monthly air " +
        "temperature coefficient used in Jensen -Haise potential evapotranspiration " +
        "computations, see  PRMS manual for calculation method")
    @Unit("per degrees")
    @In public double[] jh_coef;

    @Role(PARAMETER)
    @Description("HRU air temp coefficient - Jensen-Haise Air temperature " +
        "coefficient used in Jensen-Haise potential  evapotranspiration " +
        "computations for each HRU.  See PRMS  manual for calculation method")
    @Unit("per degrees")
    @In public double[] jh_coef_hru;
    @Role(PARAMETER)
    @Description("Units for measured temperature Units for measured " +
            "temperature (0=Fahrenheit; 1=Celsius)")
    @In public int temp_units;

    @Role(PARAMETER)
    @Description("Month to begin testing for transpiration Month to begin " +
        "summing tmaxf for each HRU; when sum is  >= to transp_tmax, transpiration begins")
    @Unit("month")
    @In public int[] transp_beg;

    @Role(PARAMETER)
    @Description("Month to stop transpiration period Month to stop transpiration " +
        "computations;  transpiration is computed thru end of previous month")
    @Unit("month")
    @In public int[] transp_end;

    @Role(PARAMETER)
    @Description("Tmax index to determine start of transpiration Temperature " +
        "index to determine the specific date of the  start of the transpiration " +
        "period.  Subroutine sums tmax  for each HRU starting with the first " +
        "day of month  transp_beg.  When the sum exceeds this index, transpiration begins")
    @Unit("degrees")
    @In public double[] transp_tmax;


    // Input vars
    @Description("The computed solar radiation for each HRU. [solrad]")
    @Unit("calories/cm2")
    @In public double[] swrad;

    @Description("Average HRU temperature. [temp]")
    @Unit("C")
    @In public double[] tavgc;

    @Description("Average HRU temperature. [temp]")
    @Unit("F")
    @In public double[] tavgf;

    @Description("Maximum HRU temperature. [temp]")
    @Unit("C")
    @In public double[] tmaxc;

    @Description("Maximum HRU temperature. [temp]")
    @Unit("F")
    @In public double[] tmaxf;

    @In public double deltim;
    @In public int active_hrus;
    @In public int[] hru_route_order;
    @In public double basin_area_inv;
    @In public int newday;
    @In public int route_on;
    @In public Calendar date;

    // Output vars
    @Description("Switch indicating whether transpiration is occurring  " +
        "anywhere in the basin (0=no; 1=yes)")
    @Out public int basin_transp_on;

    @Description("Switch indicating whether transpiration is occurring (0=no; 1=yes)")
    @Out public int[] transp_on;

    @Description("Potential evapotranspiration on an HRU")
    @Unit("inches")
    @Out public double[] potet;

    @Description("Basin area-weighted average of potential et")
    @Unit("inches")
    @Out public double basin_potet;

    @Out public double basin_potet_jh;

    private void init() {
       // init ...
    }

    @Execute
    public void execute() {
       // execute code ...
    }
}
```

The following section shows the documentation as it is created from the PotetJH.java class as a docbook5 section.

4. Native Language Interoperability

The use of modeling code written in languages other that Java is supported in OMS. This enables legacy code written in 'scientific' simulation languages such as FORTRAN, C, C++ to be used within an OMS model. All native interoperability as described below is accomplished using the Java Native Architecture (JNA) , an open source library that emphasizes an easy integration of Dynamic Linkable Libraries into Java.

JNA has been originally developed to allow for easy Java - C/C++ communication. It does not burden the developer with traditionally JNI (Java Native Interface) management and other intermediate files/APIs. In contrast to JNI which supports static interoperability, JNA uses dynamic dispatching at runtime to connect to native DLLs directly from Java. JNA's design aims to provide native access in a natural way with a minimum of effort. No boilerplate or generated code is required. While some attention is paid to performance, correctness and ease of use take priority.

4.1. FORTRAN 90/95

This section will introduce the use of JNA (Java Native Architecture) for direct Java/FORTRAN interoperability.

Examples for C/C++ are discussed in detail at the JNA website, however the use of FORTRAN within the scientific community is as much as important. It can be achieved with the 'out-of-the-box' JNA library. The objective of this section is to show how to craft, compile, and link FORTRAN code to be accessible directly from Java using JNA.

4.1.1. General Setup

The following FORTRAN example function takes two arguments and returns their product.

```
! Multiplication function that binds to the C language as 'foomult'
FUNCTION mult(a, b) BIND(C, name='foomult')
   ! both arguments are passed by value
   INTEGER,VALUE :: a,b
   INTEGER :: mult

   mult = a * b
END FUNCTION mult
```

- It uses the `BIND` keyword to provide for a C name binding. In Java/JNA this function can be called under that name.
- The function parameter are declared as value parameter. If omitted, `a` and `b` would be passed in by reference.

The FORTRAN function above can be referenced and used in Java using JNA. OMS provides on top of JNA a small convenience library that makes handling of DLLs even easier with respect to runtime binding and deployment. The use of this library is not required, however it makes the integration straightforward and simplifies deployment.

```
import oms3.annotation.*;

@DLL("F90Dyn")
```

```
interface F95Test extends com.sun.jna.Library {
   // java interface method to FORTRAN
   int foomult(int a, int b);
}

// Bind 'F90Dyn.dll' to the interface 'F95Test'
F95Test lib = Libraries.bindLibrary(F95Test.class);
```

- The FORTRAN function resides in file `libF90.dll`, that is accessible in the `jna.library.path`.
- The static call `Libraries.bindibrary` belongs to the JNA API and binds all interface methods as specified in `F95Test` to their counterparts in `libF90.dll`.
- The Java interface function maps to the name as specified in `BIND`. This solves naming problems that results from different handling of symbol names in object files/dlls with respect to underscoring. Using `BIND` is highly recommended, since it ensures a consistent external name for the function/subroutine regardless of the compiler being used and its location within a module.
- Since function arguments are passed in by value, regular native `int` types can be used within the Java interface method. However, assigning new values within the FORTRAN function to a and b won't be propagated to the caller. Use the 'Call by reference' method if this is desired.

The method can now be called like this:

```
...
int result = lib.foomult(20, 20);
assert result ==  400;
...
```

For more details on compiling/linking see further below.

4.1.2. Scalar Arguments by Value

```
! Multiplication function that binds to the C language as 'foomult'
FUNCTION mult(a, b) BIND(C, name='foomult')
   ! both arguments are passed by value
   INTEGER,VALUE :: a,b
   INTEGER :: mult

   mult = a * b
END FUNCTION mult
```

```
import oms3.annotation.*;

@DLL("F90Dyn")
interface F95Test extends com.sun.jna.Library {
   // java interface method to FORTRAN
   int foomult(int a, int b);
}

// Bind 'F90Dyn.dll' to the interface 'F95Test'
F95Test lib = Libraries.bindLibrary(F95Test.class);
```

4.1.3. Scalar Arguments by Reference.

To call a subroutine with arguments by reference, you shall **not** use the `VALUE` keyword on FORTRAN argument declaration. Now you can assign new values to the arguments, that will be later visible to Java.

```
SUBROUTINE ffunc(a, b) BIND(C,"reffunc")
```

```
   INTEGER :: a,b
   a = 3
   b = 5
END SUBROUTINE
```

The Java interface method needs to be modified to support call by reference via the JNA API `ByReference` classes.

```
...
void reffunc(ByReference a, ByReference b);
...
```

The `reffunc` subroutine will be called as follows:

```
...
IntByReference a = new IntByReference(0);
IntByReference b = new IntByReference(0);
F95Test.lib.reffunc(a, b);
assertEquals(3, a.getValue());
assertEquals(5, b.getValue());
...
```

Now you create the int reference objects, pass them into reffunc and retrieve the values with `.getValue()`.

4.1.4. Array Arguments

Single and Multidimensional arrays can be handled in JNA/Java and FORTRAN. Like with Strings, the length of the array has to be passed in with additional arguments.

```
SUBROUTINE inc(arr, len) BIND(C, name='fooinc')
   INTEGER,DIMENSION(len) :: arr
   INTEGER,VALUE :: len
   INTEGER :: i

   DO i = 1, len
       arr(i) = arr(i) + 30
   END DO
END SUBROUTINE

SUBROUTINE arr2d(arr, m, n) BIND(C, name='arr2d')
   INTEGER,DIMENSION(m,n) :: arr
   INTEGER,VALUE :: m
   INTEGER,VALUE :: n
   INTEGER :: i,j

   DO i = 1, m
       DO j = 1, n
           arr(i,j) = arr(i,j) + 1
       END DO
   END DO
END SUBROUTINE
```

The examples above show the declaration and the use of a one and two dimensional array as subroutine arguments. The array is dimensioned by the extra parameter, they are passed in as value arguments.

The JNA/Java declaration part is shown below. Note that the multidimensional array, has to be one-dimensional in Java. FORTRAN will lay it out correctly by using the dimension lengths that are passed in.

```
interface F95Test extends Library {
  ...
  void fooinc(int[] arr, int len);
  void arr2d(int[] arr, int m, int n);
  ...
}
```

The use if the one dimensional array is pretty simple. The other example required a bit management on the Java side, that is not shown here.

```
//1D
int[] a = {1, 2, 3, 4, 5};
lib.fooinc(a, a.length);
assertArrayEquals(new int[]{31, 32, 33, 34, 35}, a);

//2D
int[] a = {1, 2, 3, 4, 5, 6};
lib.arr2d(a, 3, 2);
assertArrayEquals(new int[]{2, 3, 4, 5, 6, 7}, a);
```

If a real Java multidimensional array needs to used in FORTRAN, it needs to be flattened into 1D, or you use an access method in Java to use a 1D Array in a 2D way.

4.1.5. String Arguments

String arguments are always special, since Strings are represented differently in almost all languages. In FORTRAN, you declare a string argument as follows, note that the size of the string has to be passed in as an additional argument.

The following function takes a string argument and verifies the content and length. The argument line is defined as a `CHARACTER` array, its length is passed as a second argument by value, and it is being used to dimension the length of the string.

```
FUNCTION strpass(line, b) BIND(C, name='foostr')
   CHARACTER(len=b) :: line
   INTEGER, VALUE :: b
   LOGICAL :: strpass

   strpass = (line == 'str_test') .AND. (b == 8)
END FUNCTION
```

The Java/JNA prototype looks like this:

```
...
boolean foostr(String s, int len);
...
```

The application will need to pass in the string and obtain the actual string length.

```
...
String test = "str_test";
boolean result = lib.foostr(test, test.length());
assertTrue(result);
...
```

4.1.6. Modules

Modules can be used to place all subroutines/functions that should be used via JNA, its good practice. A module allows for global data, an module level `IMPLICIT NONE`. Again, it is recommended to use the `BIND` keyword since the compiler might alter the subroutine name in the DLL otherwise, since it is a different scope.

```
MODULE test

 IMPLICIT NONE

 CONTAINS

 SUBROUTINE ffunc(a, b) BIND(C,"reffunc")
    INTEGER :: a,b
    a = 3
    b = 5
 END SUBROUTINE

END MODULE test
```

The example above the subroutine `ffunc` can still be called as `reffunc` from JNA/Java.

4.1.7. TYPE Arguments

Type arguments for functions can be handled too. This allows the passing of complex objects directly from Java to FORTRAN. Lets suppose you have the following FORTRAN code, that defines a `TYPE` for a City and a subroutine `typepass` that takes such an argument.

```
MODULE test

 IMPLICIT NONE

 TYPE :: City
   INTEGER  :: Population
   REAL(8)  :: Latitude, Longitude
   INTEGER  :: Elevation
 END TYPE

 CONTAINS

 SUBROUTINE typepass(c) BIND(C, name='footype')
   TYPE(CITY) :: c

   c%Population = c%Population + 1000
   c%Latitude = c%Latitude + 5
   c%Longitude = c%Longitude + 5
   c%Elevation = c%Elevation + 9
 END SUBROUTINE

END MODULE test
```

Both the `TYPE` and the `subroutine` are placed in a `module`.

Now lets look at the JNA/Java counterpart that defines the interface for `typepass`:

```
import com.sun.jna.Library;
import com.sun.jna.Structure;

public static class City extends Structure {
   public int Population;
   public double Latitude, Longitude;
   public int Elevation;
}

@DLL(F95Test)
interface F95Test extends Library {
   void footype(City c);
}
```

There is an Java class called City that must have the identical internal layout to its FORTRAN TYPE. The names, however, do not matter. It also has to be subclass of Structure which is defined in the JNA API.

Note that all fields of `City` have to be public to allow JNA to compute its size. The F95Test method again used the `BIND` name and the City argument.

An application will instantiate theCity object and pass it in as usual.

```
...
City city = new City(3000, 0.222, 0.333, 1001);
F95Test.lib.footype(city);

assertEquals(4000, city.Population);
assertEquals(5.222, city.Latitude, 0.0001);
assertEquals(5.333, city.Longitude, 0.0001);
assertEquals(1010, city.Elevation);
...
```

4.1.8. Pitfalls and Obstacles

- Always be aware that FORTRAN subroutine/function arguments are passed by reference, unless the `VAL-UE` modifier is used. You might end up accessing memory that might cause a segfault. Therefor use always `Native.setProtected(true)` to provide for more memory protection in the JNA site, if supported for your architecture.
- If JNA cannot find your function in a DLL and both names match in source, do not panic. You should explore the DLL to find out the real name in your DLL, since this is what JNA is looking at not the source. Do something like `nm libF90Test.dll | grep reffunc` if `reffunc` is the function you'd like to call. You'll see maybe a different (more underscores in the name, or a module name prefix) name depending on the compiler and compiler flags. This is the name you should use in your Java interface. To make this more transparent use the `BIND` keyword in your source to ensure the proper name in the DLL.
- If you pass Java objects to FORTRAN as TYPE, all Java fields have to be public. JNA will complain at runtime not being able to determine the size of the Java object.
- Be aware of the array ordering in FORTRAN that sees a two dimensional array always in COLUMN/ROW order. Also, you cannot pass a real multidimensional Java array to FORTRAN, since those do not have a continuous memory layout. On the Java side you always have to manage a one dimensional array that you reshape for FORTRAN by passing its dimensions into the function/subroutine.
- If a DLL cannot be found at runtime, you need to set the search path. You can set the system property `jna.library.path` to point to paths on your file system. You also use the `NativeLibrary.addSearchPath` method to add a map a directory to a specific DLL name.

4.1.9. Data Type Mapping

The following table shows equivalent data types between FORTRAN and Java, when passed by value

Table 6.1. JNA FORTRAN-Java Data Type Mapping

FORTRAN	JAVA
INTEGER(Kind=8)	int
INTEGER(Kind=4)	short
REAL(Kind=4)	float
REAL(Kind=8)	double
LOGICAL	boolean
CHARACTER	byte
CHARACTER(len=)	String

4.1.10. DLL Generation

he following sections will provide some help for managing the build process using different compilers. GNU's compiler collection and the G95 spin-off, as well as the Intel Compiler suite seem to be the most important tools for the general developer.

4.1.10.1. G95

G95 allows compiling and linking into a DLL. Note that G95 is not a part of the GNU compiler collection. To compile and link a FORTRAN source into a DLL use the following flags for GCC tools:

Compile a FORTRAN source into an object file:

```
$ g95 -fno-underscoring  -c -g -o build/ftest.o ftest.f90
```

Link the DLL:

```
$ g95 -Wl,--add-stdcall-alias -shared -o dist/libF90Dyn.dll build/ftest.o
```

Note that you have to use G95 for linking too. This ensures for linking the right FORTRAN runtime libraries into your DLL

4.1.10.2. GFortran

[TBD]

4.1.10.3. Intel FORTRAN

[TBD]

4.2. C/C++

[TBD]

4.2.1. Dynamic Link Library Generation

[TBD]

4.3. References

- JNA
- GCC
- Intel Compiler

5. Embedding OMS

[tbd]

Use cases:

- hand over a simulation to a user who just wants to apply the model
- a model will be used in a deployment environment such as a web server, etc.
- a simulation should be certified for production by an authorized person or institution, the simulation can be explored since it is self-documenting with respect to its components, model, and parameter files.

A simulation is deployed as a Jar file. This is called a Simulation Jar. This simulation jar has the following characteristics

- It contains all the resources that are required by that simulation such as the simulation file, the model, the components, default parameter sets, libraries.
- It also contains all the OMS runtime classes to execute the simulation.
- It contains description about the origin and version of those resources.
-

The simulation jar is self contained, no other external classes are required to run the simulation, everything needed is packaged together. The simulation jar is also 'sealed'. Only classes from within the simulation file are being used for execution, no external code cannot be injected into the simulation. This is an important security feature.

Appendix A. Glossary

Annotation	A Java annotation is a special form of syntactic meta data that can be added to Java source code. This feature is available in Java 5+.
CBSE	Component Based Software Engineering
Component	A component is a software unit (class, module) which provides an implementation for exact modeling concept. It is context-independent both in the conceptual and technical domain.
Compound	A Compound is a complex component containing other simple and compound components.
Model	A compound assembly of components to that has a application use case. In general is being used to express relevant system aspects in a mathematical/algorithmic form.
Model Base	A family of related model components.
JAR (Java Archive)	Java archive. A jar file (*.jar) contains a directory structure of Java files and other resources. It can be compressed.
JVM	Java Virtual Machine. The execution environment for Java Byte code.
Interface	An interface declares a certain behavior of a class. It is a type that specifies but never implement methods.
Meta Data	Context information about data such as physical unit of a variable, its valid range constraints, etc.
Unit	Predefined unit to express or measure a quality.
CLASSPATH	A classpath is an environment variable that list a set of directories containing Java .jar or .class files that are being used in an application.
Java Virtual Machine (JVM)	The JVM executes the bytecode (*.class) produces by a java compiler
POJO	Plain Old Java Object
IEF	Initialize/Execute/Finalize, a shortcut describing the structural concept of components.
SHA	Secure Hash Algorithm

Appendix B. Annotation Reference

1. Annotation Types

Annotations are used to specify resources within a class that relate to its use as a modeling component for OMS3. Such annotations may have different relevance and importance to different aspects of the use of component use. The same annotations can also play different roles depending on their context. There are three main annotation categories:

Mandatory Execution Annotations	Such meta data is essential information for component execution (in addition to the documentation purpose). Thee describe method invocation points and data flow between components. This is required meta data.
Supporting Execution Annotations	Such meta data supports the execution by providing additional information about the kind of data flow, physical units, and range constraints that might be used during execution. This is optional meta data.
Documentation Annotations	Those annotations are being used for documentations, presentation layers, databases, and other content management system. This is required meta data for component publication, but optional for execution.

What are Annotations? Annotations are a Java feature since version 1.5. They are an add-on to the Java language to allow for custom and domain specific markups of language elements. They do not affect directly the class semantics, but they do affect the way classes are treated by tools, such as a modeling framework. Annotations can be seen as extension of the Java Classes with meta information that can be obtained up from sources files, compiled classes, or loaded classes at runtime. They respect also languages scopes and are supported by Java IDEs with code completion and syntax highlighting.

The table below shows all modeling annotations categorized by language elements they are describing.

Table B.1. Component Annotations Overview

Class	Field	Method
`@Description`	`@Description`	`@Execute` 1)
`@Author`	`@In` 1)	`@Initialize`
`@Bibliography`	`@Out` 1)	`@Finalize`
`@Status`	`@Unit`	
`@VersionInfo`	`@Range`	
`@Keywords`	`@Role`	
`@Label`	`@Bound`	
`@SourceInfo`	`@Label`	
`@License`		
`@Documentation`		
`@DLL` 2)		

Note

1) Required annotation for a component.

2) Interface annotation for Native Language integration

Annotation names always start with the '`@`' character, indicating the difference to a regular class. The `@` (at) sign was chosen because 'AT' can be seen as abbreviation for 'Annotation Type'. Annotations can only appear once for a given language element. For example it is illegal to use the `@Author` annotation twice for a component. Instead, the name field of an `@Author` should list the two names, separated by some delimiter.

The following sections introduce all modeling annotations in detail, examples are given.

1.1. @Description

The `@Description` annotation provides for component summary information, such as a brief paragraph about its purpose, scientific background, etc. It is being used for automatic capturing the purpose of a component by archiving tools, online presentation or documentation tools, or to supplement database integration. The component selection during the process of model building and repository management can be supported by this annotation which should not exceed a few sentences. If more context information need to be provided, the `@Documentation` annotation should be used in addition.

Synopsis `@Description(<String>)`

`arg` - the description paragraph

The description can be localized for different languages. Add the ISO language code (http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt) to provide description in a different language. Description for multiple languages is supported.

Type Documentation Annotation

Scope Class, Field

Example

```
 @Description
   ("Circle Area Calculation.")

 public class CircleArea {
    ...
    @Description("Radius")
    @In public double r;
    ...
 }
..
```

```
 @Description
   (en="Circle Area Calculation.",
    de="Berechnung der Kreisflaeche")

 public class CircleArea {
    ...
    @Description("Radius")
    @In public double r;
    ...
 }
..
```

1.2. @Documentation

The `@Documentation` annotation serves as a connector or link to more detailed background documentation about the component. It allows to reference other documents via a URL. Usually those documents reside on a public server or local hard drive as PDF, HTML, Docbook, or an other text document. The reference is provided as using different URL protocols such as `http`://..., `https`://..., `file`://...

Synopsis `@Documentation(<URL>)`

`arg` - URL reference to more detailed documentation.

Type Documentation Annotation

Scope Class

Example

```
@Documentation
  ("http://myserver.com/docs/CricleArea.pdf")
public class CircleArea {
   ...
}
```

1.3. @Author

The optional `@Author` annotation provides information about the authorship of the component. The annotation has sub the fields `name`, `org`, and `contact` provide more details about the name, the affiliated organization, and some contact information such as an email address, or a link to a home page.

Synopsis `@Author(name=<String>, org=<String>, contact=<String>)`

`name` - the name of the authors(s)

`org` - organization name (optional)

`contact` - contract information such as email or address (optional)

Type Documentation Annotation

Scope Class

Example

```
@Author
  (name="Joe Scientist",
   org="Research Org",
   contact="joe.scientist@research-org.edu")

public class HamonET
    ...
}
```

1.4. @Status

This annotation enriches a component with some development and deployment status information. A status is a component quality indicator. A developer can specify the level of completeness or maturity of a component with this tag.

Synopsis `@Status(<Enum>)`

arg - `Status.DRAFT (Initial development status, private prototype)`

`Status.SHARED, (component worth sharing, still in development)`

`Status.TESTED, (component is tested in a model, test datasets and unit tests available)`

`Status.VERIFIED, (component is implemented properly, complete)`

`Status.VALIDATED, (Component fulfills requirements, validation tests available`

`Status.CERTIFIED (Component accepted and certified by authorty)`

Type Documentation Annotation

Scope Class

Example

```
@Description
   ("Circle Area Calculation.")
@Status
   (Status.TESTED)
public class CircleArea {
    ...
}
```

This annotation might be consumed by tools that publish the component to a component repository, it should control the publication process. Another use case would be the pre-run check of a deployed model that all of its components are certified by a authority.

1.5. @VersionInfo

The `@VersionInfo` annotation takes a argument that represents the version of this component. A developer might use version control system supported keyword substitution for this. The example below shows the use of the Subversion keywords `$Id` to provide revision number, modification time, and committer name as version information record. Major version control systems (CVS, Subversion,...) either have a built-in support for this feature or it can be used in conjunction with external tools (Mercurial, GIT). Therefore this annotation should not only contain an arbitrary version number, but a full version record instead is good common practice.

Synopsis	`@VersionInfo(<String>)`
	
	`arg - Version information record`

Type	Documentation Annotation
Scope	Class
Example	see below

```
@VersionInfo
   ("$Id: ET.java 20 2008-07-25 22:31:07Z od $")
public class ET {
  ...
}
```

Component repositories can use and present this information, archiving tools or documentation generators might pick this up too.

1.6. @SourceInfo

The `@SourceInfo` annotation captures information about the source. This should be some hint about source availability, maybe the source location or some contact information. The example below shows the use of Subversion's keyword substitution for the head URL of a source file. It can also point to a specific tagged version with a repository.

Synopsis	`@SourceInfo(<String>)`
	
	`arg - source URL reference`

Type	Documentation Annotation
Scope	Class
Example	see below

```
@SourceInfo
   ("$HeadURL: http://www.test.org/repo/ET.java $")
public class ET {
   ...
}
```

`@SourceInfo` is optional. Component repositories or documentation generators can use and present this information

1.7. @Keywords

A component an be tagged with the `@Keywords` annotation to characterize/categorize it. It does have the same purpose like a keyword list in a scientific paper. This is optional meta data and can be used to index, search, and retrieve archived and stored components. It is optional meta data.

Synopsis `@Keywords(<String>)`

`arg - list of kerwords separated by comma`

Type Documentation Annotation

Scope Class

Example

```
@Description
   ("Circle Area Calculation.")
@Keywords
   ("Geometry, 2D")
public class CircleArea {
   ...
}
```

1.8. @License

The `@License` annotation to specify the license for a component. It is optional meta data. If not present it is assumed the component is in the public domain and there are no restrictions for its reuse. The license can be in lined text, however it is recommended to use a URL to point to the license text.

Synopsis `@License(<String>)`

`arg - the license text or a URL to its location`

Type Documentation Annotation

Scope Class

Example

```
@Description("Circle Area Calculation.")
@License("http://www.gnu.org/licenses/gpl-2.0.html")
public class CircleArea {
   ...
}
```

1.9. @Label

Labels relate to ontologies (label is an OWL annotation). Labeling a field or component maybe provides for alternative names. They can be used to relate components or fields to another naming convention, terminology, or ontologies.

Synopsis `@Label(<String>)`

`arg - an alternative name`

Type Documentation Annotation

Scope Class, Field

Example

```
public class Calc {
    @Label("latitude")
    @In public double lat;
     ...
}
```

Labels are optional.

1.10. @In

The `@In` annotation on a field specifies it as input to the component. The field **must** be public. It indicates a read (or input access) from within the `Execute` method to the field. There are no arguments for this annotation.

Synopsis	`@In`
Type	execution, documentation annotation
Scope	Field
Example	

```
...
@In public double latitude;
...
```

This annotation is a *required* annotation for execution to enable data flow between components. `@Out` fields of one component might be connected to an `@In` field of a second component.

1.11. @Out

The @Out annotation on a field specifies it as output of the component. The field **must** be public and the `Execute` method will write to it. It is used to connect to an `@In` field of another component. There are no arguments for this annotation.

Synopsis	`@Out`
Type	execution, documentation annotation
Scope	Field
Example	

```
 ...
 @Out public double daylen;
 ...
```

This annotation is a *required* annotation for execution to enable data flow between components. `@Out` fields of one component might be connected to an `@In` field of a second component.

1.12. @Range

The `@Range` annotation is supporting an `@In` or an `@Out` field. If present, it defines a min/max range in which the value of the field is considered valid. It is up to the execution runtime to handle the range information. Violating a ranges might lead to execution abortion if it is a serious problem or just a warning message. Another use of the range information would be in component testing, see Section ???.

Synopsis `@Range(min=<double>, max=<double>)`

`min` - the minimum value, (default=`Double.MIN`)

`max` - the maximum value, (default=`Double.MAX`)

Type Execution Annotation

Scope Field

Example

```
...
@Range (min=-90, max=90)
@In public double latitude;
...
```

In the example above the latitude value can only be in the range of -90 to +90 degree. A value out of this range would probably break any equation that is using latitude. The range use above is similar to a pre-execution check.

1.13. @Role

The `@Role` annotation gives an `@In` or `@Out` tagged field a certain meaning within the modeling domain. It allows someone to understand the meaning of a data field within the modeling context. A @Role annotation categorizes a field. Such categories might be "Parameter", "Variable", "Output", "Input", "Simulated" and others. The Role annotation takes the category as a String parameter. There are predefined categories defined in @Role, however categories can be defined by the component developer.

If the @Role annotation is not provided, the default `Role.VARIABLE` it is assumed.

Synopsis

```
@Role(<String>)

arg - the role that this field is playing in context of the component.

predefined:

Role.PARAMETER, Role.VARIABLE, Role.SIMULATED, Role.OBSERVED, Role.STATE,
Role.OUTPUT
```

Type
Documentation Annotation, Testing

Scope
Field

Example

```
@Role(Role.PARAMETER)
@In public double latitude;
```

This example tags 'latitude' as Parameter.

```
@Role(Role.OUTPUT_FILE + Role.PARAMETER)
@In public File input;
```

Roles can be combined too. Now the 'input' field is a parameter and an output file.

1.14. @Unit

A `@Unit` annotation binds a physical unit to a component field that is tagged as `@In` or `@Out`. Units are usually attached to scalars and arrays fields. This information allows the frameworks to perform unit checking/validation and unit conversion. There are several open source unit conversion libraries available that could be used to perform unit conversion. An example unit conversion implementation is given in Section ???.

Synopsis

```
@Unit(<String>)

arg - the physical unit of the field
```

Type
Documentation Annotation, execution support

Scope
Field

Example

```
public class Calc {

   @Unit("degree")
   @In public double latitude;
   ...
}
```

.

1.15. @Bound

A `@Bound` defines a binding to another field. It allows to express dependencies between fields. An array field could be bound to another field that holds the size for that particular array.

Synopsis

```
@Bound(<String>)

arg - the name of the field that this field is bould to.
```

Type
Documentation Annotation, execution support.

Scope	Field

Example

```
public class ET {

   @Bound("nsim")         // 'jh_coeff' is bound to 'nsim'
   @In public double[] jh_coeff;
   ...
   @In public int nsim;
   ...
}
```

1.16. @Execute

The method that is tagged with the `@Execute` annotation provides the implementation logic of the component. In this method the component Input is being transformed to output. The execution method can have any name, it has to be non-static, `public`, `void` return type, no arguments.

This is *required* meta data for a component.

Synopsis	`@Execute`
Type	Execution Annotation
Scope	Method

Example

```
public class Component {

   @Execute
   public void executemethod() {
       // execute code here
   }
}
```

1.17. @Initialize

Within the `@Initialize` method the internal state of a component is initialized. For example opening a file for reading, or a creating a data base connection would be something that should be done within `@Initialize`

Synopsis	`@Initialize`
Type	Execution Annotation
Scope	Method

Example

```
public class Component {

   @Initialize
   public void start() {
      // initialization code
   }
}
```

Name the initialize method any name you want, but annotate it with `@Initialize` The initialize methods has to be non-static, `public`, `void`, and has no arguments. This method gets called once after component instantiation and before the first execution. This is optional meta data.

1.18. @Finalize

This method provides the notion of a final cleanup after model execution (e.g. Closing a DB connection). Usually the `@Finalize` method and the `@Initialize` method are both present.

Synopsis	`@Finalize`

Type Execution Annotation

Scope Method

Example

```
public class Component {

    @Finalize
    public void cleanup() {
        // execute code here
    }
}
```

The `@Finalize` method gets called after the final `@Execute` and the termination of the model.

1.19. @DLL

The `@DLL` Annotation simplifies the integration of native Libraries written in C++, C, and FORTRAN. It takes an argument that corresponds to the name of the DLL (Windows), Shared Object (Linux/Unix), or Library (OSX).

If for example the argument is

Synopsis `@DLL(<String>)`

`arg` - the core name of the DLL (without lib prefix in Linux, no file extension).

Type Interface Annotation

Scope Interface<T extends Library>

Example

```
import oms3.annotation.*;

 @DLL("F90Dyn")
 interface F95Test extends com.sun.jna.Library {
    // java interface method to FORTRAN
    int foomult(int a, int b);
 }

 // Bind 'F90Dyn.dll' to the interface 'F95Test'
 F95Test lib = Libraries.bindLibrary(F95Test.class);
```

Note: This annotation is supported by the

2. Meta Data Representation

There are various strategies for attaching meta data annotations to components.

2.1. Embedded Annotations

Embedded Annotations are the preferred method for annotating modeling components. They are placed directly into the source code. Therefore it is easy to keep code and meta data in sync during development.

```
import oms3.annotations.*;

 public class Daylen {

    static final int[] DAYS = {
        15, 45, 74, 105, 135, 166, 196, 227, 258, 288, 319, 349
    };

    @Range(min=6, max=18)
    @Out public double daylen;

    @In public Calendar currentTime;

    @Role("Parameter")
```

```
   @Range(min=-90, max=90)
   @In public double latitude;

   @Execute
   public void execute() {
       int month  = currentTime.get(Calendar.MONTH);
       double dayl = DAYS[month] - 80.;
       if (dayl < 0.0)
           dayl = 285. + DAYS[month];

       double decr = 23.45 * Math.sin(dayl/365.*6.2832)*0.017453;
       double alat = latitude*0.017453;
       double csh = (-0.02908 - Math.sin(decr) * Math.sin(alat))
                        /(Math.cos(decr) * Math.cos(alat));
       daylen = 24.0 * (1.570796 - Math.atan(csh /
                        Math.sqrt(1. - csh * csh))) / Math.PI;
   }
}
```

2.2. Attached Annotations

The following Listing show a alternative implementation of the `Daylen` component. It was split into two parts, (i) a pure computational component class `Daylen.java` and (ii) the component meta data class `DaylenCompInfo.java` . Only the latter has meta data dependencies to `OMS3`.

DaylenCompInfo.java

```
 public abstract class DaylenCompInfo {

   @Range(min=6, max=18)
   @Out public double daylen;

   @In public Calendar currentTime;

   @Role(“Parameter”)
   @Range(min=-90, max=90)
   @In public double latitude;

   @Execute
   public abstract void execute();

}
```

As a rule, an attached component meta data class has the same name like the component but ends with `CompInfo`. This class has to be public and abstract. It duplicates all the relevant fields and methods that should be annotated for OMS3. The methods should all be abstract. It is important to use the same spelling for fields and methods.

Daylen.java

```
public class Daylen {

   static final int[] DAYS = {
       15, 45, 74, 105, 135, 166, 196, 227, 258, 288, 319, 349
   };

   public double daylen;
   public Calendar currentTime;
   public double latitude;

   public void execute() {
       int month  = currentTime.get(Calendar.MONTH);
       double dayl = DAYS[month] - 80.;
       if (dayl < 0.0)
           dayl = 285. + DAYS[month];
```

```
        double decr = 23.45 * Math.sin(dayl/365.*6.2832)*0.017453;
        double alat = latitude*0.017453;
        double csh = (-0.02908 - Math.sin(decr) * Math.sin(alat))
                         /(Math.cos(decr) * Math.cos(alat));
        daylen = 24.0 * (1.570796 - Math.atan(csh /
                         Math.sqrt(1. - csh * csh))) / Math.PI;
    }
}
```

There are pro and cons for using embedded and attached component meta data. External meta data enables clean and neutral computational components parts with no framework dependency. However, two separate files have to be managed and have to kept in sync while doing component development.

2.3. Attached XML

[tbd]

Appendix C. Recommended Practices

This Chapter discusses general best practices with respect to model development. Those practices may or may with respect to easy language interoperability, ...

1. FORTRAN Coding Conventions

This document addresses coding conventions for OMS components and scientific code written in Java and the FORTRAN programming language.

The purpose of this document is to ensure that new FORTRAN code will be as portable and robust as possible, as well as consistent throughout the system. It builds upon commonly shared experience to avoid error-prone practices and gathers guidelines that are known to make codes more robust.

This document covers items in order of decreasing importance (see below), deemed to be important for any code. It is recognized in the spirit of this standard that certain suggestions which make code easier to read for some people (e.g. lining up attributes, or using all lower case or mixed case) are subjective and therefore should not have the same weight as techniques and practices that are known to improve code quality. For this reason, the standards within this document are divided into three components; Standards, Guidelines and Recommendations:

Required	Aimed at ensuring portability, readability and robustness. Compliance with this category is mandatory.
Recommended	Good practices. Compliance with this category is strongly encouraged. The case for deviations will need to be argued by the programmer.
Encouraged	Compliance with this category is optional, but is encouraged for consistency purposes.

Depending on the projects, programmer may opt to adhere to all three levels or just the two first. All projects must adhere at least to the mandatory standards.

1.1. General Good Practices

These usually help in the robustness of the code (by checking interface compatibility for example) and in the readability, maintainability and portability. They are reminded here:

- Encapsulation: Use of modules for procedures, functions, data.
- Use Dynamic Memory allocation for optimal memory usage.
- Derived types or structures which generally lead to stable interfaces, optimal memory usage, compactness, etc.
- Optional and keyword arguments in using routines.
- Functions/subroutines/operators overloading capability.
- Intrinsic functions: bits, arrays manipulations, kinds definitions, etc.

1.2. Interoperability and Portability

Required

- Source code must conform to the ISO FORTRAN 95 standard.
- No use shall be made of compiler-dependent error specifier values (e.g. IOSTAT or STAT values).
- No compiler- or platform-dependent extensions shall be used.

- Source code must compiled and run under gfortran that is part of the GNU Compiler Collection.

Recommended

- Note that STOP is a F90/95 standard. EXIT(N) is an extension and should be avoided. It is recognized that STOP does not necessarily return an error code. If an error code must be passed to a script for instance, then the extension EXIT could be used but within a central place, so that to limit its occurrences within the code to a single place.
- Precision: Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The F90/95 KIND feature should be used instead.
- Do not use tab characters in the code to ensure it will look as intended when ported. They are not part of the FORTRAN characters set.

Encouraged

- For applications requiring interaction with independently-developed frameworks, the use of KIND type for all variables declaration is encouraged to facilitate the integration.

1.3. Readability

Required

- Use free format syntax
- Use consistent indentation across the code. Each level of indentation should use at least two spaces.
- Use modules to organize source code.
- FORTRAN keywords (e.g., DATA) shall not be used as variable names.
- Use meaningful, understandable names for variables and parameters. Recognized abbreviations are acceptable as a means of preventing variable names getting too long.
- Each externally-called function, subroutine, should contain a header. The content and style of the header should be consistent across the system, and should include the functionality of the function, as well as the description of the arguments, the author(s) names. A header could be replaced by a limited number of descriptive comments for small subroutines.
- Magic numbers should be avoided; physical constants (e.g., pi, gas constants) should never be hardwired into the executable portion of a code; use PARAMETER statements instead.
- Hard-coded numbers should be avoided when passed through argument lists since a compiler flag, which defines a default precision for constants, cannot be guaranteed.

Recommended

- Use construct names to name loops, to increase readability, especially in nested loops.
- Similarly, use construct names in subroutines, functions, main programs, modules, operator, interface, etc.
- Include comments to describe the input, output and local variables of all procedures. Grouping comments for similar variables is acceptable when their names are explicit enough.
- Use comments as required to delineate significant functional sections of code.
- Do not use FORTRAN statements and intrinsic function names as symbolic names.
- Use named parameters instead of “magic numbers”; REAL, PARAMETER :: PI=3.14159, ONE=1.0

- Do not use GOTO statements. These are hard to maintain and complicate understanding the code. If absolutely necessary to use GOTO (if using other constructs complicates the code structure), thoroughly document the use of the GOTO.

Encouraged

- When writing new code, adhere to the style standards within your own coding style. When modifying an old code, adhere to the style of the existing code to keep consistency.
- Use the same indentation for comments as for the rest of the code.
- Functions, procedures, data that are naturally linked should be grouped in modules.
- Limit to 80 the number of characters per line (maximum allowed under ISO is 132)
- Use of operators <, >, <=, >=, ==, /= is encouraged (for readability) instead of .lt., .gt., .le., .ge., .eq., .ne.
- Modules should be named the same name as the files they reside in: To simplify the makefiles that compile them. Consequently, multiple modules in a single file are to be avoided where possible.
- Use blanks to improve the appearance of the code, to separate syntactic elements (on either side of equal signs, etc) in type declaration statements
- Always use the :: notation, even if there are no attributes.
- Line up vertically: attributes, variables, comments within the variables declaration section.
- Remove unused variables
- Remove code that was used for debugging once this is complete.

1.4. Robustness

Required

- Use Implicit NONE in all codes: main programs, modules, etc. To ensure correct size and type declarations of variables/arrays.
- Use PRIVATE in modules before explicitly listing data, functions, procedures to be PUBLIC. This ensures encapsulation of modules and avoids potential naming conflicts. Exception to previous statement is when a module is entirely dedicated to PUBLIC data/functions (e.g. a module dedicated to constants).
- Initialize all variables. Do not assume machine default value assignments.
- Do not initialize variables of one type with values of another.

Recommended

- Do not use the operators == and /= with floating-point expressions as operands. Check instead the departure of the difference from a pre-defined numerical accuracy threshold (e.g. epsilon comparison).
- In mixed mode expressions and assignments (where variables of different types are mixed), the type conversions should be written explicitly (not assumed). Do not compare expressions of different types for instance. Explicitly perform the type conversion first.
- No include files should be used. Use modules instead, with USE statements in calling programs.
- Structures (derived types) should be defined within their own module. Procedures, Functions to manipulate these structures should also be defined within this module, to form an object-like entity.
- Procedures should be logically flat (should focus on a particular functionality, not several ones)

- Module PUBLIC variables (global variables) should be used with care and mostly for static or infrequently varying data.

Encouraged

- Use parentheses at all times to control evaluation order in expressions.
- Use of structures is encouraged for a more stable interface and a more compact design. Refer to structure contents with the % sign (e.g. `Absorbents%WaterVapor`).

1.5. Arrays

Required

- Subscript expressions should be of type integer only.
- When arrays are passed as arguments, code should not assume any particular passing mechanism.

Recommended

- Use of arrays is encouraged as well as intrinsic functions to manipulate them.
- Use of assumed shapes is fine in passing vectors/arrays to functions/arrays.

Encouraged

- Declare DIMENSION for all non-scalars

1.6. Dynamic Memory Allocation / Pointers

Required

- Use of allocatable arrays is preferred to using pointers, when possible. To minimize risks of memory leaks and heap fragmentation.
- Use of pointers is allowed when declaring an array in a subroutine and making it available to a calling program.
- Always initialize pointer variables in their declaration statement using the NULL() intrinsic. INTEGER, POINTER :: x=> NULL()
- The preferable mechanism for dynamic memory allocation is automatic arrays, as opposed to ALLOCATABLE or POINTER arrays for which memory must be explicitly allocated and deallocated; space allocated using ALLOCATABLE or POINTER must be explicitly freed using the DEALLOCATE statement.

Recommended

- Always deallocate allocated pointers and arrays. This is especially important inside subroutines and inside loops.
- Always test the success of a dynamic memory allocation and deallocation - the ALLOCATE and DEALLOCATE statements have an optional argument to allow this.
- In a given program unit do not repeatedly ALLOCATE space, DEALLOCATE it and then ALLOCATE a larger block of space - this will almost certainly generate large amounts of unusable memory.

Encouraged

- Use of dynamic memory allocation is encouraged. It makes code generic and avoids declaring with maximum dimensions.
- For simplicity, use Automatic arrays in subroutines whenever possible, instead of allocatable arrays.

1.7. Looping

Required

- Do not use GOTO to exit/cycle loops, use instead EXIT or CYCLE statements.

Recommended

- No numbered DO loops such as (DO 10 ...10 CONTINUE).

1.8. Functions/Procedures

Required

- The SAVE statement is discouraged; use module variables for state saving.
- Do not use an entry in a function subprogram.
- Functions must not have pointer results.
- The names of intrinsic functions (e.g., SUM) shall not be used for user-defined functions.
- Procedures that return a single value should be functions; note that single values could also be user-defined types.
- All communication with the module should be through the argument list or it should access its module variables.

Recommended

- All dummy arguments, except pointers, should include the INTENT clause in their declaration
- Limit use of type specific intrinsic functions (e.g., AMAX, DMAX - use MAX in all cases).
- Avoid statically dimensioned array arguments in a function/subroutine.
- Check for invalid argument values.

Encouraged

- Error conditions. When an error condition occurs inside a function/procedure, a message describing what went wrong should be printed. The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list.
- Functions/procedures that perform the same function but for different types/sizes of arguments, should be overloaded, to minimize duplication and ease the maintainability.
- When explicit interfaces are needed, use modules, or contain the subroutines in the calling programs (through CONTAINS statement), for simplicity.
- Do not use external routines as these need interface blocks that would need to be updated each time the interface of the external routine is changed.

1.9. I/O

Required

- I/O statements on external files should contain the status specifier parameters err=, end=, iostat=, as appropriate.
- All global variables, if present, should be set at the initialization stage.

Recommended

- Avoid using NAMELIST I/O if possible.
- Use write rather than print statements for non-terminal I/O.
- Use Character parameters or explicit format specifiers inside the Read or Write statement. DO not use labeled format statements (outdated).

1.10. FORTRAN Features that are obsolescent and/or discouraged

Required

- No Common blocks. Modules are a better way to declare/store static data, with the added ability to mix data of various types, and to limit access to contained variables through use of the ONLY and PRIVATE clauses.
- No assigned and computed GO TOs - use the CASE construct instead
- No arithmetic IF statements - use the block IF construct instead
- Avoid DATA, ASSIGN Labeled DO BACKSPACE Blank COMMON, BLOCK DATA
- Use REAL instead of DOUBLE PRECISION
- Branch to END IF outside the block IF
- DO non-integer Control
- Hollerith Constants
- PAUSE
- multiple RETURN
- Alternate RETURN

Recommended

- Do not make use of the equivalence statement, especially for variables of different types. Use pointers or derived types instead.

Encouraged

- No implicitly changing the shape of an array when passing it into a subroutine. Although actually forbidden in the standard it was very common practice in FORTRAN 77 to pass 'n' dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in FORTRAN 90, is still possible possible with external routines for which no Interface block has been supplied. This only works because of assumptions made about how the data is stored.

1.11. Source Files

Required

- Document the function interface: argument name, type, unit, description, constraint,defaults.
- The INCLUDE statement shall not be used; use the USE statement instead.

Recommended

- Try to limit source column length, including comments, to 80 columns (or follow language specific limits).
- A component should not exceed 300-500 effective lines of code, be efficient with your coding.

- Use blank lines (or lines with a standard character in column 1) to separate statement blocks to improve code readability.
- Apply consistent indentation method for code.
- Module/subprogram names shall be lower case; the name of a file containing a module/subprogram shall be the module/subprogram name with the suffix *.f90."

Encouraged

- Clearly separate declaration of argument variables from declaration of local variables.
- Use descriptive and unique names for variables and subprograms (so as to improve the code readability and facilitate global string search);
- try to limit name lengths to 12-15 characters.
- Indent continuation lines to ensure that, for example, parts of a multi-line equation line up in a readable manner.
- Start comment text with a standard character (e.g. !, C, etc.); if a stand-alone line then start comment character in the first column.

1.12. General Coding Guidelines

- Reduce or eliminate global variable usage.
- Attempt to limit the number of arguments in argument list - long lists make it hard to reuse.
- Limit of only one return point per component.
- Use exceptions as error indicators if supported.
- Components should be specific to one and only one purpose.
- Components with side effects are not allowed (e.g. Don't mix I/O code with computational code).
- Program against a standard (e.g., ANSI C, C++, Java, FORTRAN 77/90/95) -
- Make sure your code compiles under different compilers and platforms.
- Use preprocessor directives for adaptation to different architectures/compilers/OS.
- Make I/O specific components separate from computational components.
- Avoid static allocation of data (compile time allocation).
- Be most specific with your data types.
- Avoid using custom data types for argument types.

Appendix D. License

OMS3 is licensed under the Open Software License ("OSL") version 3.0:

1) **Grant of Copyright License**. Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, for the duration of the copyright, to do the following:

> a) to reproduce the Original Work in copies, either alone or as part of a collective work;
>
> b) to translate, adapt, alter, transform, modify, or arrange the Original Work, thereby creating derivative works ("Derivative Works") based upon the Original Work;
>
> c) to distribute or communicate copies of the Original Work and Derivative Works to the public, with the proviso that copies of Original Work or Derivative Works that You distribute or communicate shall be licensed under this Open Software License;
>
> d) to perform the Original Work publicly; and e) to display the Original Work publicly.

2) **Grant of Patent License**. Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, for the duration of the patents, to make, use, sell, offer for sale, have made, and import the Original Work and Derivative Works.

3) **Grant of Source Code License**. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work.

4) **Exclusions From License Grant**. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior permission of the Licensor. Except as expressly stated herein, nothing in this License grants any license to Licensor's trademarks, copyrights, patents, trade secrets or any other intellectual property. No patent license is granted to make, use, sell, offer for sale, have made, or import embodiments of any patent claims other than the licensed claims defined in Section 2. No license is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under terms different from this License any Original Work that Licensor otherwise would have a right to license.

5) **External Deployment.** The term "External Deployment" means the use, distribution, or communication of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether those works are distributed or communicated to those persons or made available as an application intended for use over a network. As an express condition for the grants of license hereunder, You must treat any External Deployment by You of the Original Work or a Derivative Work as a distribution under section 1(c).

6) **Attribution Rights.** You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent, or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7) **Warranty of Provenance and Disclaimer of Warranty.** Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately preceding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of non-infringement, merchantability or fitness for a particular purpose. THE ENTIRE RISK AS TO

THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to the Original Work is granted by this License except under this disclaimer.

8) **Limitation of Liability.** Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to anyone for any indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to the extent applicable law prohibits such limitation.

9) **Acceptance and Termination.** If, at any time, You expressly assented to this License, that assent indicates your clear and irrevocable acceptance of this License and all of its terms and conditions. If You distribute or communicate copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. This License conditions your rights to undertake the activities listed in Section 1, including your right to create Derivative Works based upon the Original Work, and doing so without honoring these terms and conditions is prohibited by copyright law and international treaty. Nothing in this License is intended to affect copyright exceptions and limitations (including "fair use" or "fair dealing"). This License shall terminate immediately and You may no longer exercise any of the rights granted to You by this License upon your failure to honor the conditions in Section 1(c).

10) **Termination for Patent Action.** This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11) **Jurisdiction, Venue and Governing Law.** Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of copyright or patent law in the appropriate jurisdiction. This section shall survive the termination of this License.

12) **Attorneys' Fees**. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13) **Miscellaneous.** If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14) **Definition of "You" in This License.** "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15) **Right to Use.** You may use the Original Work in all ways not otherwise restricted or conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

16) **Modification of This License.** This License is Copyright © 2005 Lawrence Rosen. Permission is granted to copy, distribute, or communicate this License without modification. Nothing in this License permits You to modify this License as applied to the Original Work or to Derivative Works. However, You may modify the text of this License and copy, distribute or communicate your modified version (the "Modified License") and apply it to other original works of authorship subject to the following conditions: (i) You may not indicate in any way that your Modified License is the "Open Software License" or "OSL" and you may not use those names in the name of your Modified License; (ii) You must replace the notice specified in the first paragraph above with the notice "Licensed under <insert your license name here>" or with a notice of your own that is not confusingly

similar to the notice in this License; and (iii) You may not claim that your original works are open source software unless your Modified License has been approved by Open Source Initiative (OSI) and You comply with its license review and certification process.

Index