seamless

System for Environmental and Agricultural Modelling; Linking European Science and Society

The Conceptual design of SeamFrame

van der Wal, T., Rizzoli, A.E., Svensson, M., Villa, F., Knapen, R., Athanasiadis, I.

Partners involved: Alterra, IDSIA, LU, UVM

Report no.: 21
November 2006
Ref: PD1.3.4
ISBN no.: 90-8585-048-7

ALTERRA WAGENINGEN UR

Logo's main partners involved in this publication

Sixth Framework Programme

SEAMLESS integrated project aims at developing an integrated framework that allows ex-ante assessment of agricultural and environmental policies and technological innovations. The framework will have multi-scale capabilities ranging from field and farm to the EU25 and globe; it will be generic, modular and open and using state-of-the art software. The project is carried out by a consortium of 30 partners, led by Wageningen University (NL).

Email: seamless.office@wur.nl
Internet: www.seamless-ip.org

Authors of this report and contact details

Name: Tamme van der Wal Partner acronym: Alterra
Address: Alterra, PO Box 47, 6700 AA Wageningen, the Netherlands
E-mail: tamme.vanderwal@wur.nl

Name: Andrea Emilio Rizzoli Partner acronym: IDSIA
Address: IDSIA, Galleria 2, 6928 Manno, Switzerland.
E-mail: andrea@idsia.ch

Name: Mats G E Svensson Partner acronym: LU
Address: Lund University, PO Box 170, S-22100, Sweden.
E-mail: mats.svensson@lucsus.lund.se

Name: Ferdinando Villa Partner acronym: UVM
Address: University of Vermont, 590 Main Street, Burlington, 05405, Vermont, USA
E-mail: ferdinando.villa@uvm.edu

Name: Rob Knapen Partner acronym: Alterra
Address: PO Box 47, 6700 AA Wageningen, the Netherlands
E-mail: rob.knapen@wur.nl

Name: Ioannis Athanasiadis Partner acronym: IDSIA
Address: IDSIA, Galleria 2, 6928 Manno, Switzerland.
E-mail: ioannis@idsia.ch

Disclaimer 1:

Disclaimer 2:

When citing this SEAMLESS report, please do so as:

van der Wal, T., Rizzoli, A.E., Svensson, M., Villa, F., Knapen, R., Athanasiadis, I. 2006. The Conceptual design of SeamFrame, SEAMLESS Report No.21, SEAMLESS integrated project, EU 6th Framework Programme, contract no. 010036-2, www.SEAMLESS-IP.org, 46 pp, ISBN no. 90-8585-048-7.

Table of contents

General information

Task(s) and Activity code(s):	Activity 1.3.4
Input from (Task and Activity codes):	T5.2 and T5.3
Output to (Task and Activity codes):	T1.4
Related milestones:	

Executive summary

This project deliverable provides the underlying architecture of a concept for linking models and databases and it provides the design of SeamFrame, delivering its architecture to provide an integration framework for models and simulation algorithms, supported by procedures for data handling and spatial representation, quality control, output visualization and documentation

Knowledge in SeamFrame is created by organizing and structuring the information we gather from a number of *sources*: from databases, from models, from expertise and world-views of the modellers and scientists. We use *ontologies* to store, organize and manipulate knowledge. Ontologies are a knowledge representation tool, which is becoming more and more popular and standardized, given its support by the World Wide Web Consortium. In ontologies, the knowledge is represented by Concepts and Properties. Ontologies will be used in SEAMLESS to represent data, models and workflows. Thanks to ontologies, all the knowledge stored in SEAMLESS-IF will be open, accessible and re-usable and it is therefore independent as much as possible from SeamFrame, the software implementation of SEAMLESS-IF.

SeamFrame is therefore organized according to a layered architecture: we have Sources, the data and models which are then represented in the Knowledge Base by means of ontologies. On top of this, we have the Modelling and the Processing Environments, which are in turn used to deliver Applications. The Modelling Environment is used to access the Knowledge Base and deliver Model Components (discrete software units), which can be processed thanks to the Processing Environment within workflows. A workflow is a sequence of operations which involve data pre-processing, model runs, and data post-processing to perform policy analyses within the context of SEAMLESS-IF. SeamFrame offers a number of Modelling Environments, to target the different needs of different types of modellers, from bio-physical to economic modellers. The Processing Environment will conform to a revised version of the OpenMI standard for model linking and interoperability in order to promote reuse of SeamFrame software components beyond the life of the SEAMLESS project itself.

This document is a snapshot of the current development of design and prototypes in WP5, which is evolving in a series of design/test/evaluate loops, which will converge towards the 18 month prototype.

The document is structured as follows: first an introduction to ontologies and their role to structure the conceptual model of SEAMLESS is provided. Then, the overall design of SeamFrame is presented. Finally, the first version of the implementation design of SeamFrame is discussed.

Specific part

1 Introduction

This project deliverable provides the underlying architecture of a concept for linking models and databases and it provides the design of SeamFrame, delivering its architecture to provide an integration framework for models and simulation algorithms, supported by procedures for data handling and spatial representation, quality control, output visualization and documentation

The document is structured as follows: first an introduction to ontologies and their role to structure the conceptual model of SEAMLESS is provided. Then, the design of SeamFrame is presented and discussed.

This document is a snapshot of the current development of design and prototypes in WP5, which is evolving in a series of design/test/evaluate loops, which will converge towards the 18 month prototype.

2 Ontologies in SEAMLESS

The most popular definition of ontology is ‘a specification of a conceptualisation’ (Gruber,1993). However, this does not help much if one wants to discover the benefits and challenges ontologies have to offer. Ontology is also a container concept and there are many interpretations and uses.

One of the simplest notions of a possible ontology may be a controlled vocabulary: a finite list of terms, like a catalogue. One step further we have a glossary: a list of terms and meanings. A thesaurus provides extra semantics through the relationship between terms. One step further is a taxonomy where a hierarchy in relations is introduced. A taxonomy makes useful subdivisions in (large) groups. One step further is the definition of formal inheritance through “IS A” relationships. Here, we define types of relations that are used i.e. in thesauri.

As stated, an ontology specifies the conceptualisation of a specific domain. So it gives a list of terms with meanings and relations, for that specific domain. Now the kind of ontology (catalogue, glossary etc.) we need depends on our ambition. What do we want to do with it? In SEAMLESS we have introduced the use of ontology for our Knowledge Base: to provide a structure where we can re-use knowledge and combine components like databases, models, rules and methods. Therefore we need to develop standards (vocabulary, formats etc.) for storing and retrieving, we need a common language to define and mine our knowledge. This implies that we do not have enough on selecting (limiting) the terms we use and describe and organise them. We also need intelligent applications to help us exploit our Knowledge Base.

A useful ontology usually contains more than just a list of terms and their definitions. Here is a set of increasingly complex uses of ontologies:

- Controlled vocabulary: A set of concepts with no properties, whose IDs define a vocabulary. For instance, you would not be allowed to use ‘w_temp’ as the name of a variable which measures water temperature, if the controlled vocabulary contains the entry ‘water_temp’.
- Taxonomy: Concepts are arranged in a generalization (is-a) hierarchy. For instance, Wheat ‘is a’ Crop.
- Database schema (relational and object-oriented): Properties other than is-a are defined for classes. Generalization hierarchy may or may not be present. No instance information are given. Limited constraints (such as cardinality) are present. For instance, the Wheat crop has a set of attributes or properties, such as its nutrient and water demand. These properties can be stored in a table, where they are defined for each crop type.
- Frame-oriented Knowledge Base: Concepts are arranged in a taxonomy, with arbitrary properties, constraints, objects, values. This is an extension of the above two concept. We can define ‘frames’ which define a concept, such as: Wheat is a crop. It has a nutrient demand of xx kg/ha of P and xx kg/ha of N.
- Description Logics: specify necessary and sufficient conditions for concepts and instances to belong to a class, allowing full reasoning and implicitly defining an inferred hierarchy as well as defining one explicitly through is-a properties. For instance, Description Logic can be used to infer properties of crop types.

The main rationale for the existence of ontologies is to enable reasoning on them, either by a human actor or by an automated program (reasoner). The main operations in reasoning are subsumption (inferring that class A is more general than class B) and classification (inferring that instance X is a child of class B). Both operations can entail significantly complexities if restrictions are present (see Data example).

Now what is important for SEAMLESS is that we define a common ontology. For our internal communication (both formal and informal) we need to have a shared understanding of terms we use. We need to comply to already existing terminologies, first of all because we do not want to reinvent wheels, but secondly because we need to communicate with others (humans and applications). If we are successful, our ontology can even get a broader impact in our science community. What is also important is that an ontology is not a dead thing. It must be maintained, updated, extended etc. This requires an active process and software tools to support it.

2.1 The composing elements of an ontology

An ontology is a formal description of a conceptualization of a domain of interest. This conceptualisation is structured as a list of *concepts*, identified by their names, connected by their *properties*, again identified by their names. For instance, *crop* could be a concept, *grows* can be a property and *soil-type* can be another concept. In most frameworks, an ontology is a declaration of one or more resources of the following types:

- Concept (also known as: Class, Set, Type, Predicate): the statement of a concept, usually including at least textual description and a short label. . Concept names (IDs) and descriptions do not define meanings formally: concepts are always defined by their properties (see below), not by their names. Most ontology frameworks admit equivalence statements to account for synonyms and alternative definitions of the same concept.

- Property (aka Attribute): the statement of a property, always associated to a concept. One must always specify the type of the property value and its cardinality, or the admitted number of possible values. For example, concept Person has a birth-date property whose value must be a Date and whose cardinality is one. The value of a property can be a Concept, an Instance (see below) or a Literal (see below), although some frameworks place restrictions. Special properties allow building the bones of the knowledge structure: notably the is-a property allows building generalization-specialization hierarchies (e.g. Employee is-a Person) and instance-of allows to define actual instances (e.g JohnSmith instance-of Employee). Properties can be generalized or specialized just like concepts: e.g. “depends-on-economically isa: depends-on”.

- Instance (aka Individual, Object) is a real-world entity that must be the incarnation of a stated Concept. For example, if the concept crop has the property grows on concept soil-type, the individual crop named Wheat grows on soil-type #12 (where 12 is an arbitrary soil classification, which can also be described in the ontology). All properties that have cardinality greater than zero in the concept must have at least a correspondent Relationship in the instance (see below). Concepts can be categorized as abstract, preventing the definition of their instances (e.g. FoodWebConcept), or concrete (e.g. Predator). An ontology comprising a set of Instances is often called a Knowledge Base, although the term is not rigorous. Any database with a formally specified schema can always be considered a set of instances.

- Relationship (aka Slot) links an Instance to the value of a specified Property. E.g. the statement “JohnSmith birth-date 10–10–1972″ can be considered the statement of a relationship in simplified RDF (Resource Description Framework, http://www.w3c.org/RDF, a lightweight ontology system to support the exchange of knowledge) . Relationships must adhere to the model specified by the Concept and its Properties.

- Literal (aka Value): the value of a property expressed in textual form, e.g. “10.4″, “August 10, 1978″ or “John Smith”. Ontology frameworks usually accept and validate of literals according to a limited set of base types (e.g. Floating Point Number, Date), often modelled on the XML Schema types.
- Restriction (aka Constraint) Define allowable values and patterns of values for concepts within an ontology, with varying levels of sophistication. The aforementioned cardinality of properties can be seen as a restriction. Some frameworks (e.g. OWL) allow considerable sophistication in specifying restrictions, as in the example below (Data section).

2.2 Development of an ontology

There are three main paths to the development of ontologies, and some of them can be travelled at the same time:

1. reuse of existing ontologies: applicable to all generic things;
2. knowledge acquisition: design our own conceptualisation for our own purpose. In fact the SEAMLESS domain model is a good example of knowledge acquisition and engineering;
3. text mining: automated extraction from large quantities of texts: not applicable in the context of the SEAMLESS project, because of lack of time and resources.

2.3 Ontology paradigms

Two main paradigms are used in ontology development: Open world vs. Closed world. The closed world assumption corresponds to the normative use: ontologies lists all the possible Concepts and specifies a compulsory schema for an instance of each concept, which can be used e.g. to define the structure of a data base. The open world assumption relies mostly on Restrictions to specify necessary and sufficient conditions for an Object to belong to a Class. These are used by a reasoner to classify pre-existing instances rather than provide a schema to define or validate them. Both paradigms have their uses in knowledge-based systems and are supported by different frameworks. A common closed-world framework is RDF Schema (RDF). A common open-world framework is the Ontology Web Language (OWL).

Normally, each individual ontology refers to a well-defined, narrow domain, and all ontology frameworks have ways to connect concepts across different ontologies. A complete knowledge-based framework usually comprehends many ontologies with different roles, that can be broadly classified as follows:

Generic (a.k.a. upper, core or reference): provide common high level concepts such as “Physical”, “Abstract”, “Structure”, “Substance”, “Unit of measurement” and define the conceptual core of the system.

Domain-oriented: provide domain-specific concepts (e.g. SolarRadiation) and domain generalizations (e.g. FarmTypology).

Task-oriented: provide task-specific concepts (e.g. RiskAnalysis) and task generalizations (e.g. DynamicModel)

2.4 Ontologies and data representation

Data can be considered “static models” of systems: they always conform to conceptual models, and depend on (usually implicit) assumptions and world views just as much as dynamic models. Conventional wisdom divides “raw data” – actual numeric measurements – from “metadata”, the information that allows a user to “make sense” out of the numbers, providing needed spatiotemporal, measurement, and other informative contexts for the numbers. In an ontology-informed framework, we start from accurately formalizing the concept that is represented in the data, and we define all its characteristics as properties. Each dataset or piece of data will be represented as one instance of that concept. One of its properties will be numeric-value and will link to the raw data represented as literal values. The other properties are the “metadata” – only connected to an explicit knowledge model (the ontology) whose arbitrary richness now allows reasoning and mediation with other data using different, compatible knowledge models. An example can clarify some of these concepts:

```
<Measurement rdf:ID="MyLayeredSoilTempereture">
   <hasSpatialContext>
      <Axis_elevation rdf:ID="Vector_Elevation_10">
      <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
          A vector consisted of 10 points on the elevation axis
      </rdfs:comment>
      <height rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
          10
      </height>
      </Axis_elevation>
   </hasSpatialContext>
   <observes rdf:resource="#SoilTemperature"/>
</Measurement>
```

In the ontology snippet above, we find the definition of an instance of the ‘Measurement’ concept. The ID of this concept is the string `MyLayeredSoilTemperature`. The metadata specifies that is has an axis, which measures elevation, which is represented as a vector of ten elements.

The ontology paradigm in its simplest incarnation offers a field-by-field substitution for the relational schemata conventionally used to represent data. Schema information contained in normative ontologies can be construed as an add-on to an existing Data Base Management System, to be mapped over a relational database engine or another (such as an XML database) in a modular fashion. In addition to that, ontologies offer a powerful synthetic way to specify both the data schema and the structure of the knowledge behind it. While a relational schema can be considered the “structural” component of the knowledge, ontologies allow specifying the how and the why at the same time. The database paradigm that includes a rich knowledge model such as that specified by open world ontologies is often called deductive database. Klein et al. (2000) describe the relation between ontologies and XML schemas, comparing it to the relationship between Entity-Relationship (ER) diagrams and database schemas. An ontology iss a formal description which can be mapped on an XML schema, as an ER diagram is a formal description of a database schema. In the same way, an ontology can be used to describe a database schema.

Most common metadata standards can be and have been formalized as ontologies: e.g. the ISO 19115 standard or the Ecological Metadata Language (EML). Yet, providing a set of upper ontologies to define the actual meaning of the metadata properties is another matter, and lends itself to different, competing interpretations that ultimately depend on the application. All mediator systems that can integrate data and models need such a set of ontologies; the ones related to the process of measurement and the conceptualization of time and space are usually crucial. Efforts are underway in projects such as SEEK, IMA, SWEET and SEAMLESS to develop the set of ontologies that best fits the application domain.

2.5 Ontologies and model representation

Dynamic models, like data, always conform to a conceptualization, and there is in fact no philosophical difference between specifying data or models when this is done using ontologies. Any sort of model can be successfully specified as a set of instances of the appropriate ontologies. The main practical difference between data and models is the increased conceptual richness necessary to describe how things change in time and space. This requires at least notions of linkage between concepts with causative or dependency relationships that are not necessary when specifying data. It also requires developing ways to interpret this causality. The set of abstractions (concepts) that allows conceptualizing and expressing those cause-effect relationships and their results is the adopted modelling paradigm, of which examples abound (e.g. ordinary differential equations, stock-and-flow, or individual-based systems). A modelling paradigm, like any consistent conceptualization, can be easily captured into an ontology. Most existing modelling software systems conform to one implicit ontology, which defines their notion of entities familiar to the user such as state variables, flux variables etc. Advanced integrative systems can load different ontologies, which, supplemented by the necessary software, enable them to manipulate models adopting heterogeneous modelling paradigms. Such systems are in the best position to enable integration of independently developed models adopting different paradigms into a higher-level, multiple-paradigm model.

There are several definitions of a 'model' found in the literature, each one of which accommodates specific needs. The one that seems more suitable for SEAMLESS practical needs, where models are used for describing and simulating the agro-environmental and agro-economical domains, could be the following one: Model is a mental construction that, based on the reality, reproduces the main components and relationships of the analyzed segment of the reality (add references). In this respect a model in SEAMLESS can be considered as a construction capable to represent biophysical, economical or farm-economic processes, as:

- a set of variables, that characterize the state of the analyzed segment of reality (often called system), and
- a set of quantitative relationships, that describe the relations between the variables, i.e. describe the behaviour of the system.

Sanchez, Cavero and Marcos (2005) have concluded that in computer science, ontologies can be considered as subsets of models. This is particularly true also for the environmental modelling exercise we try within SEAMLESS project. Our goal is to use ontologies as part of our (biophysical, economical, or farm-economic) models, in order to:

- Build a common consensus on terms used among a wide and diverse research community.
- Ensure interoperability and "openness" of models,
- Add semantics in the description of model interfaces.

In SEAMLESS, ontologies support declarative modelling by providing, at the same time, schemata for model declaration and meaning for these schemata. We can use the ontology concepts to describe the mathematical relationships among model variables, the variables and the parameters themselves, and so on. Thus, declaratively expressed models refer to concepts laid out in the ontologies. Such declarations contain enough information to enable the SEAMLESS infrastructure to simulate the behaviour of the systems represented over a user-defined temporal and spatial extent. Thanks to the rich meaning made possible by ontologies, the SEAMLESS workflow environment can properly connect models to data, and feed quantities calculated by simulation to other models in SeamFrame.

2.6 Ontologies and representation of workflows

A scientific workflow is a pathway between two or more processing steps, along which a flow of data is transformed until a desired result is reached. Workflows are usually assembled by users to connect sources of input data (such as a database query) to models, pre- and post-processing algorithms (such as statistical data reduction or calculation of indicators from model results) and visualization software.

A workflow environment is thus a "black board" for users to assemble the flow of computation needed to address a particular problem. In functional terms, a model can be seen as a workflow, because in the end all models process input information and produce a set of output results. The similarity, however, does not hold when models and workflows are considered in semantic terms: the meaning of a model is not to produce outputs, but to describe a natural process. In fact, a more correct generalization sees workflows as special cases of models, whose "paradigm" entails data transfer and transformation along the connections of an artificial system. In this sense, a workflow is amenable to the same ontology-based description as any other model: concepts of "input", "output", "processing step" can be specialized as needed using ontologies that can describe all steps of any workflow environment.

The most important use of ontologies in workflow environments, however, is another: to allow the system to enforce meaningful, correct connections between inputs and outputs, and – if necessary and possible – insert transformation steps in the workflow that guarantee a proper match. The operation of enforcing and supporting semantic consistency along data paths in workflows is usually called semantic mediation, and it is essential to guaranteeing correct results particularly when processing steps are heterogeneous and users are not domain experts. To allow semantic mediation, all inputs and outputs must come tagged with concepts from ontologies that are known to the workflow environment, and the latter needs to use a reasoner program that ensures the consistency of concepts along each connection made by the user. The operation of associating concepts from ontologies to input and output "ports" of workflow components is usually called semantic annotation, and it is done by the same experts that have developed the models or processing steps.

Semantic mediation is a young discipline and there are frequent misunderstanding about the way it's done. In particular, it's common to think that the ontology-supported system will enforce the conceptual identity along connections, and other operations in the workflow environment will ensure the matching of numeric boundaries, machine types, units of measurement and so on. In fact, a well-designed set of upper ontologies, supported by software, can take care of all those checks in one operation, usually a subsumption check. Another common mistake is the simplistic notion that identity means a literal matching of type IDs: temperature = temperature. In reality, the conceptual compatibility is most often tested with a reasoning operation that can check if different names actually mean the same concept. The example below illustrates how this may be done.

A model X is "packaged" as a workflow component and all its inputs are semantically annotated by its developer according to a set of commonly understood ontologies. The semantic annotation operation requires that all the conceptual details of each "port" is understood and appropriately defined. As an example, an input I representing temperature at surface may require that the temperature is expressed as monthly data over the simulated time span, and the model has only been calibrated for temperatures in the 19–30 C range so it should not accept data outside of these boundaries. Semantic annotation is a way to express such conditions, which normally are only expressed verbally in the model's documentation, in a formal and machine-readable way. In order to do so, an ontology is created to define model X, with concept defined for each exposed “port”. Using restrictions and concepts from appropriate ontologies, the concept definition associated with input I may look similar to this:

```
I ::=
is-a: Temperature,
vertically-distributed-in: PlanetarySurface,
has_unit: Fahrenheit,
max-value:
(Temperature, has-value: 30.0, has-unit: Celsius)
min-value:
(Temperature, has-value: 19.0, has-unit: Celsius)
distributed-in:
(TimeSpan, step: 1, has-unit: Month)
```

When a semantically annotated model is used in a workflow, inputs and outputs are connected by the user. For example, a temporal series of temperature data retrieved from a database may be connected to input I. Upon connection, a semantically aware workflow environment can ensure the appropriate match between the input and the output by feeding the respective semantic annotations to a reasoner and ask if they describe the same concept (a subsumption operation). A reasoner can make the necessary inferences to assess the equivalence of types that have different names, based on their properties. In some cases, the compatibility may not exist directly, but the reasoner can establish that a transformation can be inserted in the dataflow to make the input and output compatible. For example, the data source could be weekly data rather than monthly. A sophisticated workflow environment can understand that the data need to be aggregated into a monthly timeseries, and direct the workflow environment to create a transformation step to perform the aggregation and insert it between the data source and the model.

3 The SEAMLESS Conceptual Model

The SEAMLESS conceptual model defines the scope of our view of the agri-environmental-economic domain we want to model and analyse.

We propose a conceptualisation based on three domains:

1. The SEAMLESS agro-environmental domain (SeamAg);
2. The Application domain;
3. The Modelling domain;

We will refer to this subdivision as the **SeAM** concept. Each sub-domain is in fact a separate dimension, orthogonally situated to the others. It is important that we have a shared understanding about all three 'worlds' in order to deliver a comprehensive product of use to all user roles.

The three domains will build on the core ontology, which provides fundamental concepts to reason about space and time.

3.1 The core ontology

The core ontology contains the following basic concepts:

- *Units*: the units used to measure quantity. For instance, a unit is 'kilograms'.
- *Dimensions*: the dimensional information on a measure. For instance, a dimension is Length, which can have different units, meters, kilometres, miles, and so on. You can convert among units, but not among dimensions.
- *Quantities*: the quantities which are observed, measured, computed in the SEAMLESS conceptual model. A quantity has the following properties:
  - data type (e.g. float);
  - default unit (e.g. m^3);
  - dimension (e.g. volume);
  - domain, a property that describes to which domain this variable is attached. For instance, a crop yield is attached to the crop domain in the SeamAg domain.
- *Measurement*: quantities that are measured and can be processed by models. The processing establishes a role for a quantity (input, output, state variable or parameter) and it also defines an observation context, that is, the spatial and temporal resolution adopted in measuring the variable. Thus, a model variable has:
  - Temporal context: it is a point or a time series;
  - Spatial context: it is a-dimensional or 1D,2D,3D;
  - Observation context: the quantity that is observed by this variable.
- *Observation context*: as said, it is a concept used to specify the characteristics of a model variable, that is, the spatial and temporal context, the sampling frequency (on a spatial grid or over the temporal axis) and the native type (integer, Boolean, float).

An important issue in SEAMLESS is the scale, and the integration of scales in policy analysis. In this conceptual business model, scales are represented by the composition of

Environments. In modelling, scale is not so much the same as in cartography / GIS (like 1:1 000 000, or 1-km^2 grid) but it rather is the level of aggregation of the state variable in the model. For instance, a farm can be seen as an input-output unit, where you get an annual yield of a number of products, given the available resources, but you can also describe the biophysicial processes taking place day by day.

3.2 SEAMLESS agro-environmental domain

The SEAMLESS agri-environment domain (SeamAg) includes ecological, social, economical and institutional dimensions.

Central theme is the systems (-theory) paradigm: A **system** is an assemblage of inter-related elements comprising a unified whole (source: Wikipedia). To understand a system one must study the individual parts of the system as well as the relationships between them.

The SeamAg is organised according to the concepts of Environment, Actors, Actions, Conditions.

3.2.1 Environment

In the study domain of SEAMLESS we study the behaviour of systems in relation to changes in their environment or within themselves.

For our purposes, the words system and environment are compatible: since we both accept them as composites. For connecting as much as possible with real-world words, the term environment prevails.

An environment is an abstract term, leaving to each specific use the task of further specifying what it means in practical terms. One very important aspect of Environment is the fact that it can/must/should have a temporal and spatial extent.

We can distinguish between different types of environment. For instance, we have a biophysical environment (e.g. watershed or mountain region), a socio-economic environment (e.g. country, city) and a market (e.g. the European wheat market).

In another example, the AgroDomain is an Environment which contains subdomains such as Soil, Climate, Water. Soil, in turn, is specialised in SoilWater, SurfaceSoil and so on.

3.2.2 Actors

Another central element in SeamFrame is the Actor. Actor again is a very abstract term. An Actor explicits its behaviour into (meaningful) action, based on preferences, possibilities and conditions. Actor is abstract, meaning that we leave the task of further specifying what it means to each specific use. Actor is a composite.

We can obviously distinguish different types of actors, such as The Farmer, The Government, The Consumer, The Sugar Factory etc. Important aspects of Actors are there preferences, possibilities and conditions imposed by the environment(s) they operate in.

The relation between environment and actors is manifold, but the most direct relation is that (certain) environments are managed by actors. The Farm is managed by The Farmer, and The Country is managed by The Government. When The Farm is located within The Country, the environment The Country imposes conditions onto The Farm and therefore influences the behaviour of The Farmer. The relationship between government and farmer is therefore made explicit through the (nesting of) environments. Finally, also Climate can be seen as an actor which affects the Environment.

3.2.3 Actions and Conditions

As mentioned above, Actors take (meaningful) Actions that operate on an Environment. And Environment imposes Conditions onto Actors.

For instance, nutrient and water management are actions. On the other hand, also soil erosion is an action, which affects soil surface.

This all will look like as in Figure 3.1.

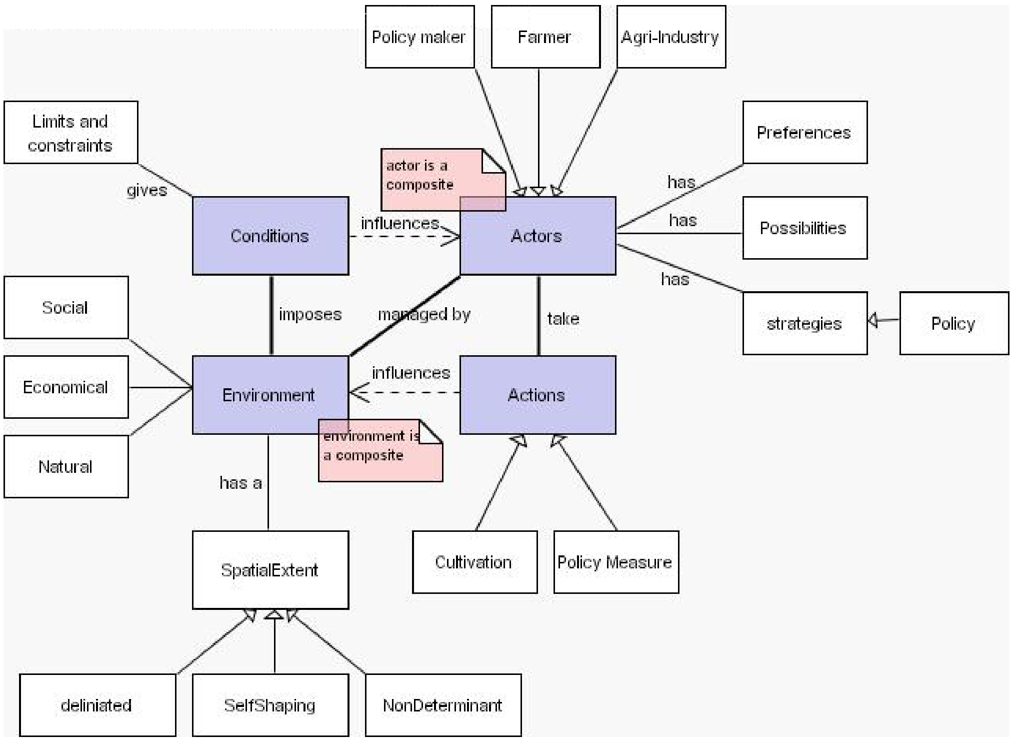

Figure 3.1. The SEAMLESS agri-environmental domain.

3.3 Modelling domain

While the Application domain is listed after the SeamAg domain, that is for cosmetic reasons, related to making up a nice acronym. The flow of logic requires the Modelling domain to be introduced now.

The modelling domain is composed of *Components*.

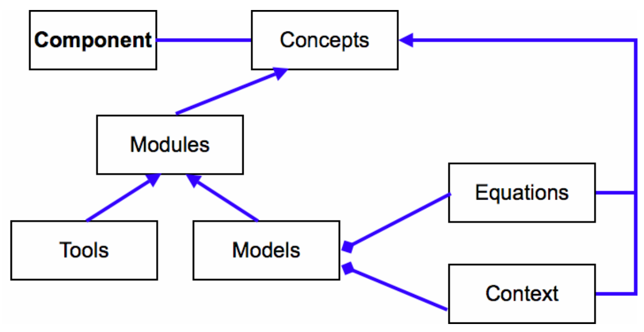

Figure 3.2. The Modelling domain.

Components consist of Concepts: they are ‘packages’ of models and tools, which have been aggregated because of their logical proximity and their functions. In the same component we find all models that are related to the same domain in SeamAg. For instance, we can have a Clima component, which contains all models related to Climate actions, such as rainfall, evapotranspiration, solar radiation, etc.

In Figure 3.2 a UML diagram explains how Models are made of Equations and their Context in time and space (e.g. their parameters). Besides Models we have Tools, such as numerical integration routines. We call *Modules* the class which includes models and tools. Modules are expressed by Concepts, which provide a gateway to store all information in the Knowledge Base.

Models are basic modelling units. The variables of a model are Measurements defined in the core ontology. Models are packaged in Components and they refer to Domains. We can therefore query the ontology to discover all models related to a given domain which are packaged in a specific component.

We have the following model classes:

- Biophysical models: They can compute one or more outputs, based on their inputs and states. Inputs, outputs, states and rates are variables. Models also have parameters. Static models do not have rates.
- Farm economic models: these are mathematical programming models, but we still can distinguish among input variables (exogenous data and decision variables), output variables (endogenous data) and parameters. They also need an objective function to be minimized or maximised.
- Management models: these models process information and return management actions such as irrigate, harvest, and so on. Management models can be open-loop (decisions are taken once and then applied according to a set schema, such as irrigation according to the day of the year) or closed-loop (decisions are revised according to the system state, such as irrigation time as a function of soil water content.
- Constraint models: these are rule-based models, which can be expressed as if-then-else clauses. Constraint models can be used within management models.

3.4 Application domain

The Application domain reflects the way we want to compose / assemble applications from individual / composite software components. We have adopted the application domain from the EU5FP project HarmonIT. In this project, modellers from across Europe have determined how components can work together in a meaningful way in order to look like one cohesive application for supporting a specific task.

In HarmonIT this domain is captured in the Open Modelling Interface: OpenMI. This framework consists of two prime components: Linkables and Links. In this way, OpenMI provides a conceptual model of coupling components, which are linkables, exchanging data between components and directing the order of calculations.

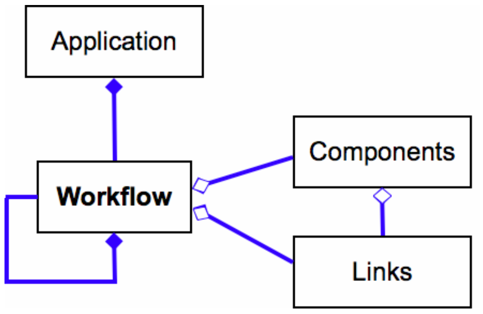

Figure 3.3. The Application domain.

As shown in Figure 3.3, an *application* implements a number of generic *workflows*, which is a chain of linkables and links, where each link connects two linkables.

We can determine, based on the different user roles, several workflows within SEAMLESS-IF:

- Analyze (a current policy);
- Simulate (business as usual, a new policy etc.);
- Communicate (what happens if …);
- Evaluate (scenario's);
- Visualize (results, outcomes etc.).

4 The SeamFrame Conceptual Design

In any development process we can identify a number of 'analyse-design-implement-test' cycles, which can repeat themselves in tighter loops during product prototyping, and become less frequent while the product becomes more and more mature.

The same process can be discovered in the analysis and development of the SeamFrame conceptual model. In the analysis phase, we identified the structure of the conceptual model.

In this section we describe the design of the architecture of SeamFrame, that is the structure and organization of the software architecture that enables us to deliver end-user applications that will support the conceptual model we previously described in Section 3.

The reading of this section can be facilitated by reading the Appendix on Object-Oriented programming, which introduces a lot of the terminology we use here.

4.1 The SeamFrame software system model

During the preparation of the SEAMLESS proposal many meetings were held, during which requirements were collected and screened and modelling frameworks were reviewed. The result of this stage of the project is available in PD 5.2.2.

From the requirements, a first sketch of SeamFrame emerged. In this chapter we briefly describe the SeamFrame system model, which is aimed at supporting a component-oriented approach to software development (Szyperski, 2002).

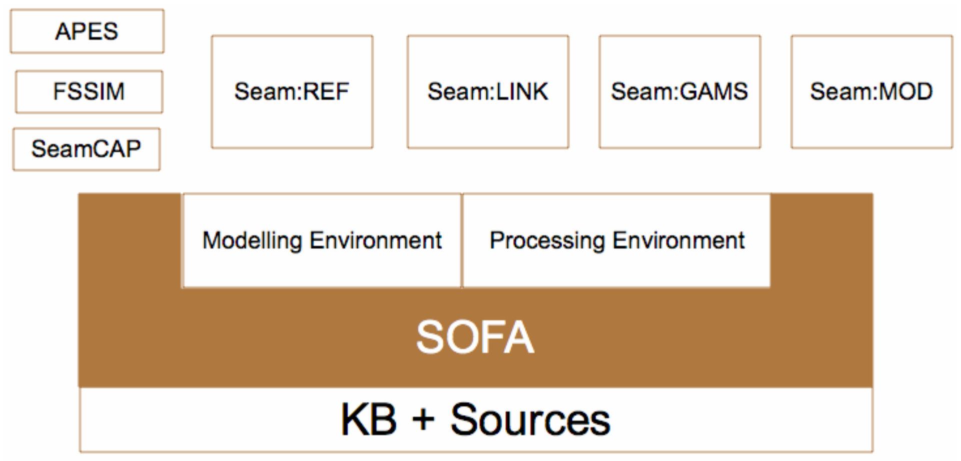

Figure 4.1. The SeamFrame architecture, with its development environment, composed by the modelling and processing environment, the Knowledge Base, and the end-user applications.

The presentation of the system is instrumental to the matching of the requirements to the main architectural elements (see Figure 4.1): the Knowledge Base, the SOFA, that is the SeamFrame core (Knowledge Manager, Model Manager, Tool Manager), the SeamFrame Development Environment (Modelling Environment, Processing Environment), a set of support applications: Seam:REF, Seam:LINK, Seam:GAMS, Seam:MOD. Using this framework it will be possible to deliver a number of SEAMLESS applications (SeamApps: e.g. APES, FSSIM, SEAMCAP).

The core classes and components of SeamFrame are structured in a package named SOFA, or SEAMLESS OpenMI+ Framework Architecture. The SOFA contains the basic components needed for the Knowledge Base and to manage the OpenMI+ compliant executable modules constructed from models and tools.

On top of the SOFA sit the packages that contain the modelling and processing environments. The modelling environment package sets the structure that a modelling application (such as MODCOM) must conform to so that it can integrate into SEAMLESS. For example, it must retrieve its data (knowledge objects) through the SOFA and implement an integration interface (*IEmbeddable*).

The processing environment contains the implementation of the workflow environment for SEAMLESS. It enables the creation of workflows of modules, which are "executable" models and tools that implement OpenMI+ (an adapted version of OpenMI 1.0) interfaces.

4.2 The Knowledge Base

The Knowledge Base (KB) is a repository for references to *sources of information*, from data to models to workflows and scenarios. Differently from the database, where SeamFrame-independent data are stored, the KB contains data elaborations which strictly depend on the SeamFrame architecture and functions.

The information is structured according to *Ontologies* which we have used to model the agri-enviro-economic sector. (the SeAM conceptual model) The ontologies will be represented using the standard OWL/RDFS languages. Adopting such language standards will pave the way to future interoperability with other frameworks and it will make possible the distribution of the SEAMLESS ontology on the web, in the spirit of collaboration with similar initiatives.

Ontologies also serve as a precious aid in browsing and querying a Knowledge Base, as they connect concepts in a meaningful way and lend themselves to powerful and intuitive forms of interactive graphical representation.

Building on the most recent advances in ontology-driven modelling and data mining (Villa, 2001; Ludaescher, 2001), we will employ the semantic characterization of models and data to enable substitutability of components and data within models and analytical pipelines, automatic generation of transformations to enforce compatibility of storage type, units of measurement and mode and scale of representation, and definition of proper aggregation strategies.

The SOFA classes and components need to exchange information with the Knowlede Base. The information to be exchanged is organized in data structures we have named *Concepts*. The Concepts provide a software layer to access the ontology. The concepts and the properties defined in the ontology become 'fields' or attributes of the Concept data structures. Among Concepts we list Models, Tools and Workflows. The KB will contain *Models*, both in binary format and declarative format. Models will be characterised by their inputs, states, outputs and parameters. Models will be searchable. This part of the KB is called the *Model Base*.

The KB will contain *Tools*, implemented as software components, that provide services such as numerical integration, data analyses, data visualisation, scenario optimisation and so on. All these software components communicate with models through a standard interface. This part of the KB is called the *Toolbox*.

The KB will contain *Workflows*, which are ordered sequence of operations performed by tools on models and data. Scenarios are instances of workflows. This part of the KB is called the *Experiment Base*.

4.3 SOFA parts

The SeamFrame Core is constituted by a set of co-operating classes and abstract interfaces packaged in components:

- the Knowledge Manager: provides software components that process the ontology and produce classes to be used in modelling (Concepts);
- the Module Manager, which includes:
  - the Model Manager: provides abstract classes open for implementation inheritance and a set of components able to perform transformations from model's declarative code to imperative code;
  - the Tool Manager: is a set of abstract interfaces to be implemented to deliver *processing tool components* operating on models and data.

4.4 The Knowledge Manager

The *KnowledgeManager* manages the concepts stored in an ontology, that is, in a Knowledge Base. Concepts relate to ontology. Everything used by SeamFrame can be considered to be a Concept. These Concepts are created by a *ConceptManager*, which manipulates information in the Knowledge Base (the ontology) and returns object instances.

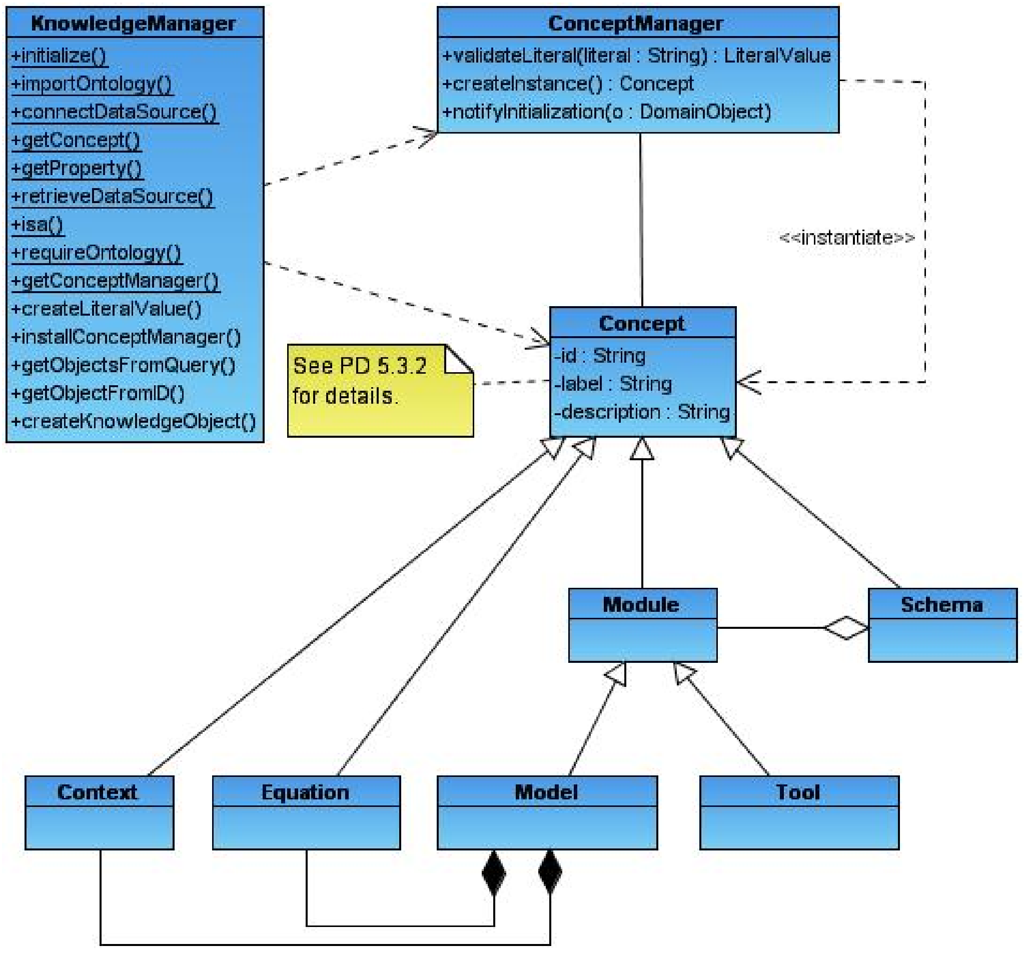

Figure 4.2. The Concept and the Knowledge manager.

As said, a *Concept* instance contains information extracted from the Knowledge Base. This information can range from numerical values, to pointers to a binary object (i.e. a DLL) that is the execution core for a module, and even to a reference to a database. These instances of Concepts thus can be viewed as configuration settings.

For SeamFrame a number of key concepts are introduced (Figure 4.2):

Module

A module is a part in a schema. *Module* itself is an abstract term.

Schema

The schema is a description of interlinked modules.

Model

A model is a module and can be part of a schema. A model can calculate and output data based on calculations or by retrieving it from a data source.

Tool

A tool is a module and can be part of a schema. A tool can control data flow in the schema, visualize data, log data, etc.

Equation

Equations are the parts a model is constructed from.

Context

The context makes a model domain and scale specific.

The KnowledgeManager implements the IKnowledgeManager interface. It contains methods to search (query) the Knowledge Base for concepts, and to store and retrieve them.

4.5 The Model Manager

The *ModelManager* is a module manager that knows how to handle models, based on information it retrieves from the *KnowledgeManager*. It can create (OpenMI+ compliant) components from a model and knows which modelling environment to start for viewing and editing of the model

The Model Manager also contains a Model processor which delivers the (more complex) functionality to transform a model in declarative specification (retrieved from the Knowledge Base) into model source code. It can generate some type of documentation of the model (e.g. in HTML), and transform it to address scaling problems or to create aggregated models from base models. It must interface with compilers for generating binary code from source code and/or declarative code.

The ModelManager implements two interfaces:

IModelManagement

Provides the methods for model management, giving access to the functionality defined for the *ModelProcessor*. Methods can result in the creation of new models.

IModuleManagement

Functionality for searching (querying) of Modules and creation of Components from them by the *ModelManager* and the *ToolManager*.

4.6 The Tool Manager

Processing tools are software components that operate on models and data. An example is a simulator, which takes as input a model, with its parameters and input data, and the simulation parameters (horizon, integration routine, step, etc.) and produces the simulation results, which, in turn, can be passed on to another component, a grapher, that displays them in a window.

There is a wide range of processing tools able to perform various activities. Tools can be added by a third-party developer since SeamFrame must be extensible. The key to extensibility is provided by the fact that tools, models and data communicate through interfaces. As long as a new processing tool implements an interface which is recognised by SeamFrame, the new tool will be usable within the framework. The interfaces are specified by the part of SeamFrame core named the *Tool Manager*.

This component defines the interfaces for SEAMLESS compatible tools (basically OpenMI+), can retrieve tool meta-information from the Knowledge Base and instantiate a tool as a component. Tool components provide additional functionality for workflows, besides the calculation and data mining functionality of the model components. Examples of possible tools are a grapher, that displays data, a simulator that runs a model according to specifications (e.g. a time horizon), a calibrator, an optimizer, or data processing tools (to convert, aggregate, dis-aggregate, interpolate, extrapolate data).

The ToolManager therefore implements the same two interfaces of the ModelManager: IModelManagement and IModuleManagement.

4.7 Module Manager

Module manager is the mother class of the ModelManager (which deals with models) and the ToolManager (which deals with tools).

Figure 4.3 describes how a *ModuleManager* will take the *Concepts* from the Knowledge Base and use them to create *Components*. These components are OpenMI+ compliant (they implement the (extended) OpenMI interfaces) and can be linked to each other for data exchange.

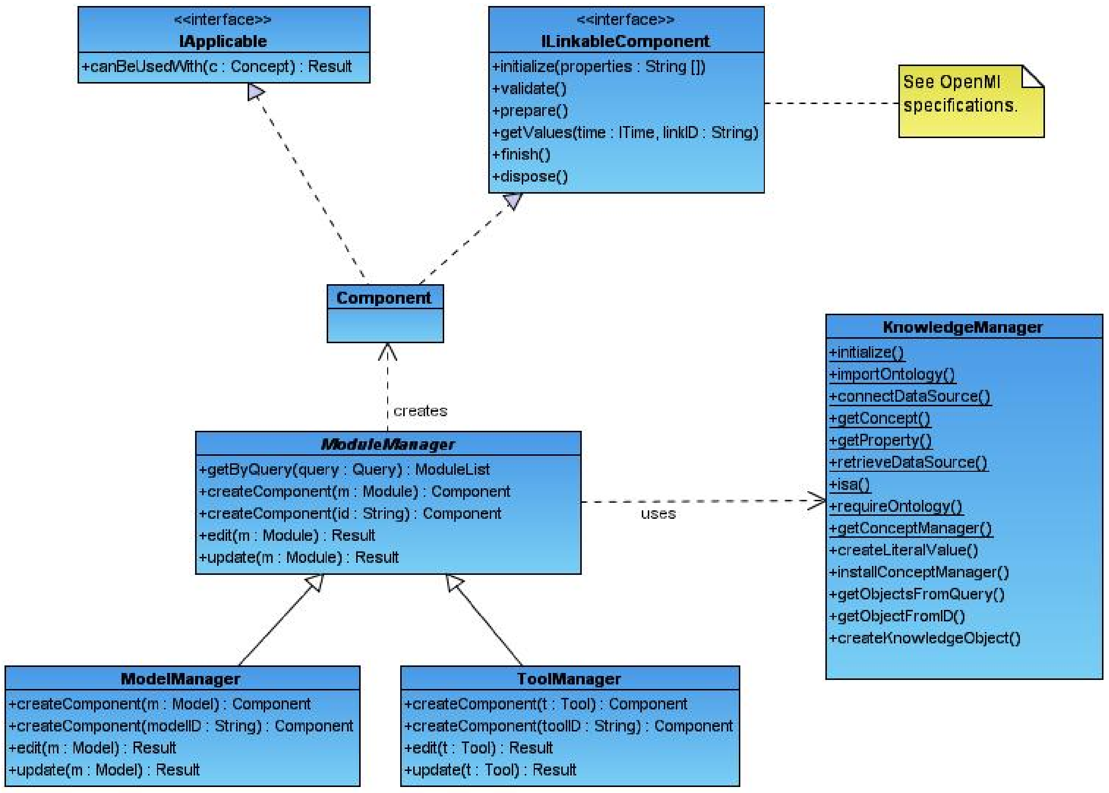
<<interface>>
IApplicable
+canBeUsedWith(c : Concept) : Result
<<interface>>
ILinkableComponent
+initialize(properties : String [])
+validate()
+prepare()
+getValues(time : ITime, linkID : String)
+finish()
+dispose()
See OpenMI specifications.
Component
creates
ModuleManager
+getByQuery(query : Query) : ModuleList
+createComponent(m : Module) : Component
+createComponent(id : String) : Component
+edit(m : Module) : Result
+update(m : Module) : Result
uses
KnowledgeManager
+initialize()
+importOntology()
+connectDataSource()
+getConcept()
+getProperty()
+retrieveDataSource()
+isa()
+requireOntology()
+getConceptManager()
+createLiteralValue()
+installConceptManager()
+getObjectsFromQuery()
+getObjectFromID()
+createKnowledgeObject()
ModelManager
+createComponent(m : Model) : Component
+createComponent(modelID : String) : Component
+edit(m : Model) : Result
+update(m : Model) : Result
ToolManager
+createComponent(t : Tool) : Component
+createComponent(toolID : String) : Component
+edit(t : Tool) : Result
+update(t : Tool) : Result

Figure 4.3. The Module Manager.

A *Component* refers to the *Concept* it was derived from. This allow to the possibility to process the knowledge contained in the Concept and associated with the Component so that matches and relations can be verified with other components. This functionality is contained in the *IApplicable* interface.

ModelManager and *ToolManager* both are types of *ModuleManagers*, allowing for different handling of models and tools, and to have method signatures for the specific types.

4.8 The Development Environment

The SeamFrame Development Environment is composed of two main applications: the Modelling Environment and the Processing Environment. In the first application, the modeller can define and edit the data structures to be used within the model and s/he can then write a model, which will be then saved in the model base. In the Processing Environment models can be run, tested, optimised, the result visualised and all these operations can be structured for later re-play in work-flows.

4.9 The Modelling Environment

The modelling environment is a part of SeamFrame’s development environment. It aims to provide the users with all the appropriate infrastructure for building, retrieving and editing agro-environmental models of various types and scales, by exploiting the functionalities of the Knowledge Manager, the Model Manager and the Tool Manager we find in the SOFA.

In order to be able to place multiple modelling environments on the SOFA they must conform to the provided abstract modelling environment component, i.e. conform to the interfaces *IModelManagement* and *IEmbeddable* and operate in a similar fashion.

The *IEmbeddable* interface provides methods for object linking and embedding to integrate modelling environments into the processing environment (and applications). Preferably so that a modelling application can be "seamlessly" integrated into one of the applications.

Implementing IEmbeddable ensures that applications like MODCOM, SIMILE, GAMS, etc. can all be used with the SeamFrame. Seam:MOD will be a OpenMI+ compliant version of MODCOM, while Seam:GAMS will be a modelling environment which delivers GAMS models which have been wrapped up as OpenMI+ compliant components. All realizations of modelling environments use the *ModelProcessor* to retrieve from and store information in the Knowledge Base. It might prove to be necessary to have a dependency on the *IKnowledgeManagement* interface of the *KnowledgeManager* as well. The modelling environments provide the infrastructure for developing both opaque (accessible only through its interface) and clear (exposing its equations) models. An opaque model is accessible only through its interface, while a clear model exposes its equations. The terms opaque and clear refer to the software interpretation and they do not have anything in common with the more usual terms of black and white box modelling. For instance, a white box model, which is a conceptual or process-based model where the causal relationships among variables are explicit, can be implemented as an opaque model component.

Development of opaque models: using the classes and components provided by Knowledge Manager and the Model Manager, a programmer will be able to use pre-formatted templates of a model. Based on these templates, written in source code, s/he will be "guided" to write the model's code, compatible with the SeamFrame standard interface, in the same way we can fill a form given its template.

Development of clear models: through the Modelling Environment's GUI a modeller will be enabled to define variables and parameters of a model using the ontology, specify the model algorithm in a declarative way (XML-based model-representation language) and finally, to create the complete source of the model, by the use of Model Manager facilities.

Thus, the Modelling Environment enables:

- Introduction of new concepts into the Knowledge Base;
- Creation of collections of concepts;
- Definition of model variables;
- Build simple models in a guided way, declarative or imperative;
- Retrieve, edit and store models in the Knowledge Base;
- Create composite models by assembling models;
- Semantic based navigate through data and model worlds.

Figure 4.4 illustrates the relation between modelling environment, concepts, components, and the *IEmbeddable* and OpenMI+ compliant interfaces. The modelling environment creates models based on equations, context and (sub)models. The *ModelManager* can turn a model concept into an OpenMI compliant component for use in workflows.

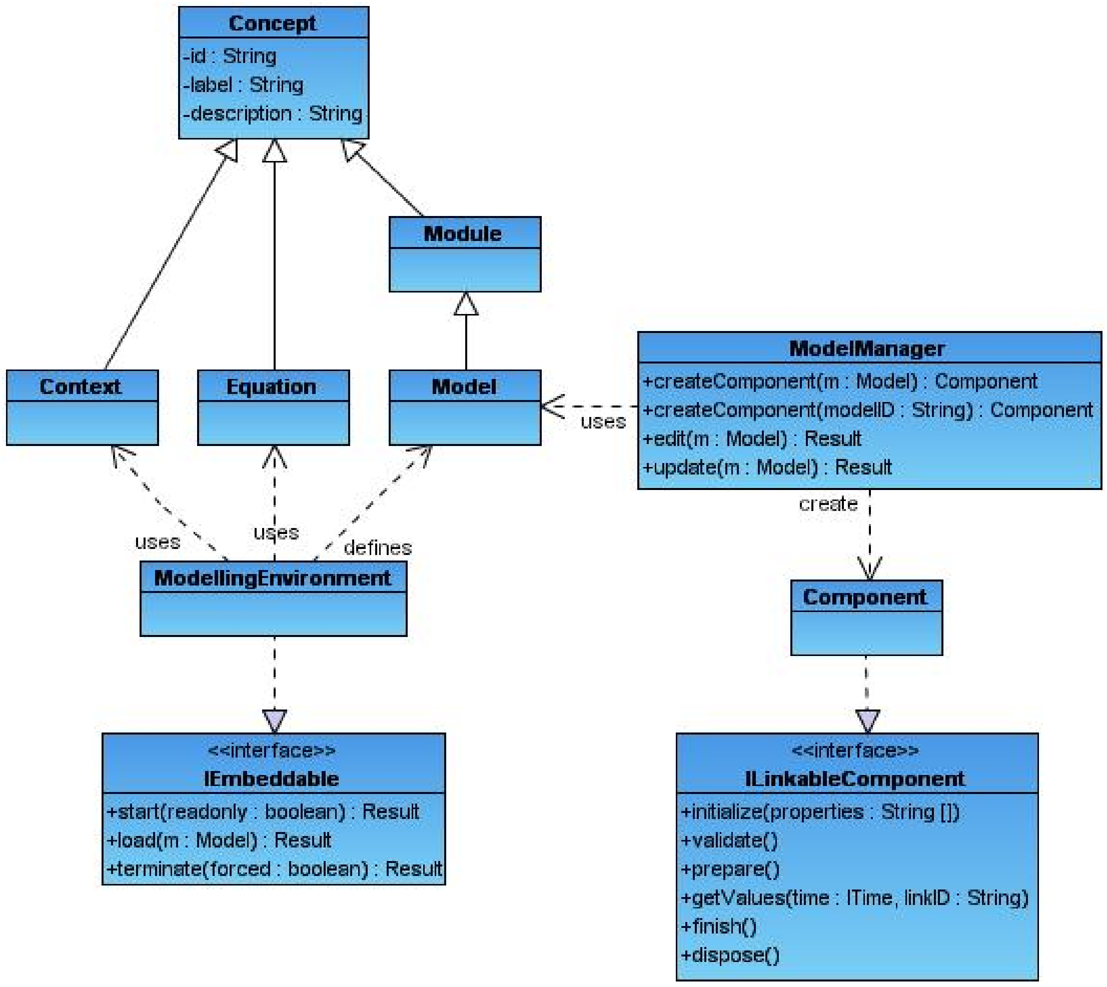
Concept
-id : String
-label : String
-description : String
Module
Context
Equation
Model
ModelManager
+createComponent(m : Model) : Component
+createComponent(modelID : String) : Component
+edit(m : Model) : Result
+update(m : Model) : Result
uses
create
uses
uses
defines
ModellingEnvironment
Component
<<interface>>
IEmbeddable
+start(readonly : boolean) : Result
+load(m : Model) : Result
+terminate(forced : boolean) : Result
<<interface>>
ILinkableComponent
+initialize(properties : String [])
+validate()
+prepare()
+getValues(time : ITime, linkID : String)
+finish()
+dispose()

Figure 4.4. The Modelling Environment.

A modelling environment might be made OpenMI+ compliant as well. The properties argument of the *initialize()* method could be used to load a model configuration, which than should be run to be able to answer to *getValue()* calls. Since this solution might raise some performance issues mitigation measures will be considered for the software implementation.

4.10 The Processing Environment

The Processing Environment is a software application built on top of SeamFrame. Its purpose is to let the user apply processing tools to models and data in order to perform the operations implemented by the tools.

The Processing Environment is designed adopting some key ideas of the OpenMI architecture (Brakkee et al. 2004).

Models and tools (as Components) can be included in workflows, that can be executed for defined experiments so that results can be obtained, viewed and analyzed. A workflow is composed of *Links* and *Components*, following the concepts of the OpenMI standard. Functions provided are:

- Prepare and run a calibration experiment:
    - Select a model, define the free parameters;
    - Select a calibration data set;

    - Select a calibration tool and objective function;
    - Create a data sink to store the calibration outputs;
    - Link the calibration tool to the model and to the data sink;
    - Store the set of calibration parameters.
- Prepare and run a simulation experiment:
    - Select a model;
    - Instantiate the model for a domain context;
    - Select data feeds for exogenous inputs;
    - Select a simulation tool, set the simulation parameters;
    - Run the simulation;
    - Store the simulation results.
- Prepare and run an optimization experiment;
- Prepare and run a scenario comparison experiment.

The *ProcessingEnvironment* will be implemented based on the current OpenMI specifications, but with the necessary fixes and adaptations (hence OpenMI+). This core is then extended with packages containing the functionality for ontology's and support for working with transparent models (in contrast to the opaque – legacy – model access from OpenMI). The processing environment will have a minimal user interface, most of that will be in the *seam:LINK* end-user application.

Everything that is to be included in a workflow must be a generalization of the abstract *Component*. These are the tools and executable models that process the raw data in an experiment. Components should conform to the OpenMI+ interfaces. The next sections examine in greater detail Components and Workflows.

4.10.1 Components

From one point of view there are two types of components, the models and the tools. However, from the workflow perspective there are just components with different capabilities and it is more important to know how they are technically constructed. This leads to a number of subtypes, *LibComponent, SourceComponent, ScriptComponent,* etc.

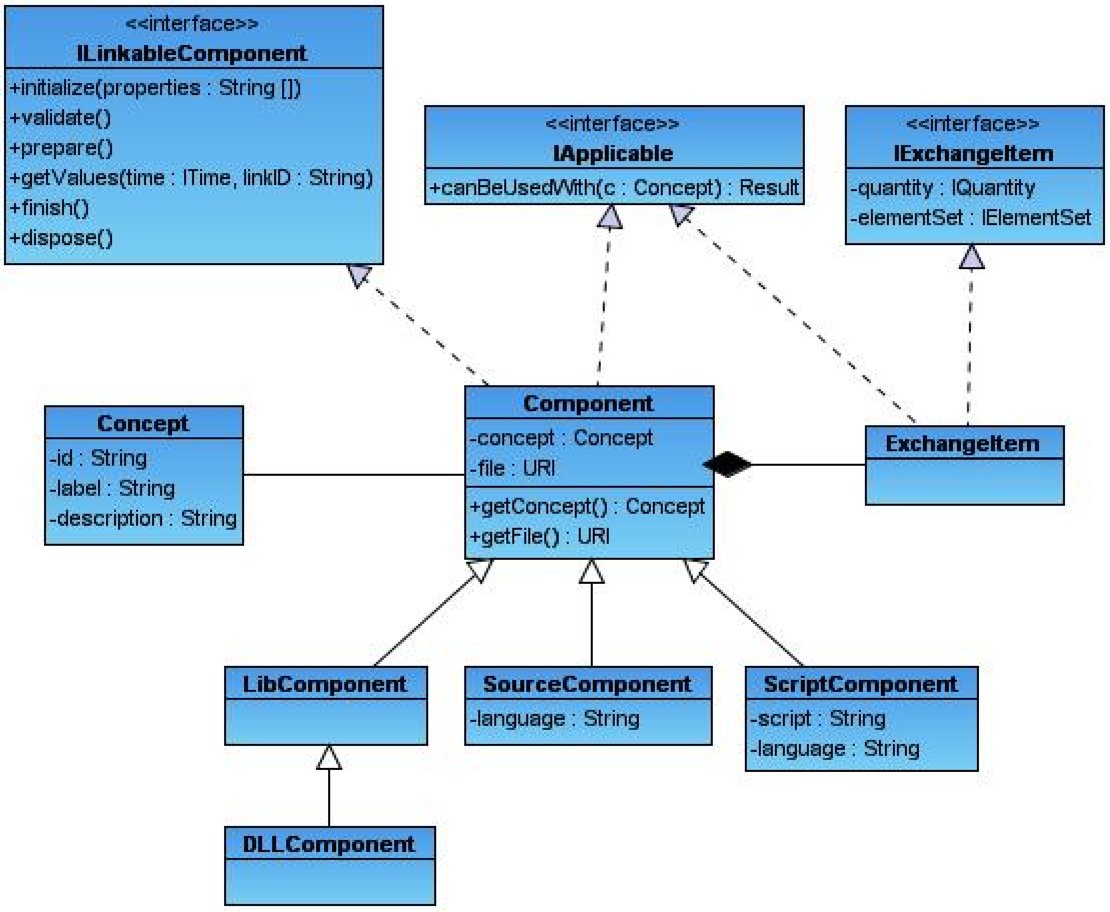

Figure 4.5. The different kinds of components.

Whether a component is based on a library (e.g. a DLL), or on source code that needs to be compiled (e.g. Java or C#), it must be contained within the *Concept*. The *ModuleManager* can then use it when instantiating a *Component*. This is graphically represented in Figure 4.5.

4.10.2 Workflows

A *Workflow* consists of a number of *Components* connected through *Links*. This will be based on OpenMI+ specifications, a *Component* must implement the OpenMI *ILinkableComponent* interface and a *Link* the *ILink* interface. For OpenMI a linkable component must specify its inputs and outputs as *ExchangeItems* (implementing the *IExchangeItem* interface).

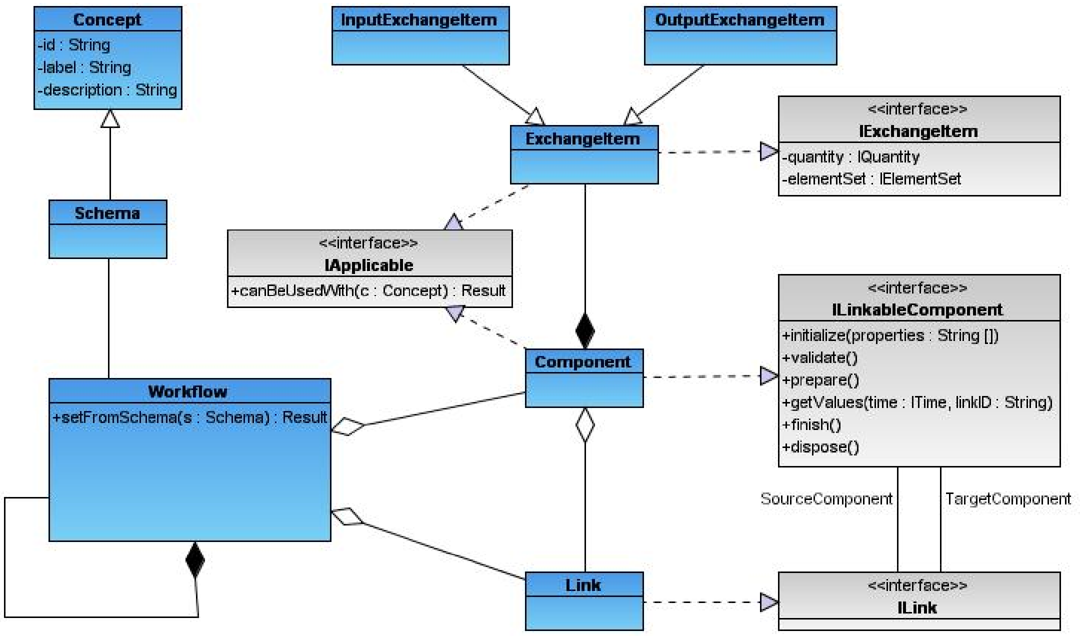

Figure 4.6. Workflows .

In Figure 4.6, which extends Figure 3.3, it is shown how one of the updates to OpenMI will be that an *ExchangeItem* must also be connected to a *Concept* (or an attribute of a concept) and implement the *IApplicable* interface, so that links can be validated based on the ontology.

A *Workflow* uses the *Schema* concept for its persistency in the Knowledge Base (using the *KnowledgeManager*).

4.11 Model Handling: where the Modelling and the Processing environment meet

Figure 4.7 illustrates the relation for modules, models and components, based on interactions of a user (a modeller) with the processing environment.

Three interactions are shown; searching for a model; adding a model to a workflow; editing a model in a workflow. The user always interacts with the processing environment and the resulting actions are shown for the *ModelManager*, the *KnowledgeManager*, a *Model* (concept) instance, and a modelling environment. The *Component* created by the *ModelManager* (step 10) is not shown to contain the diagram within the page.

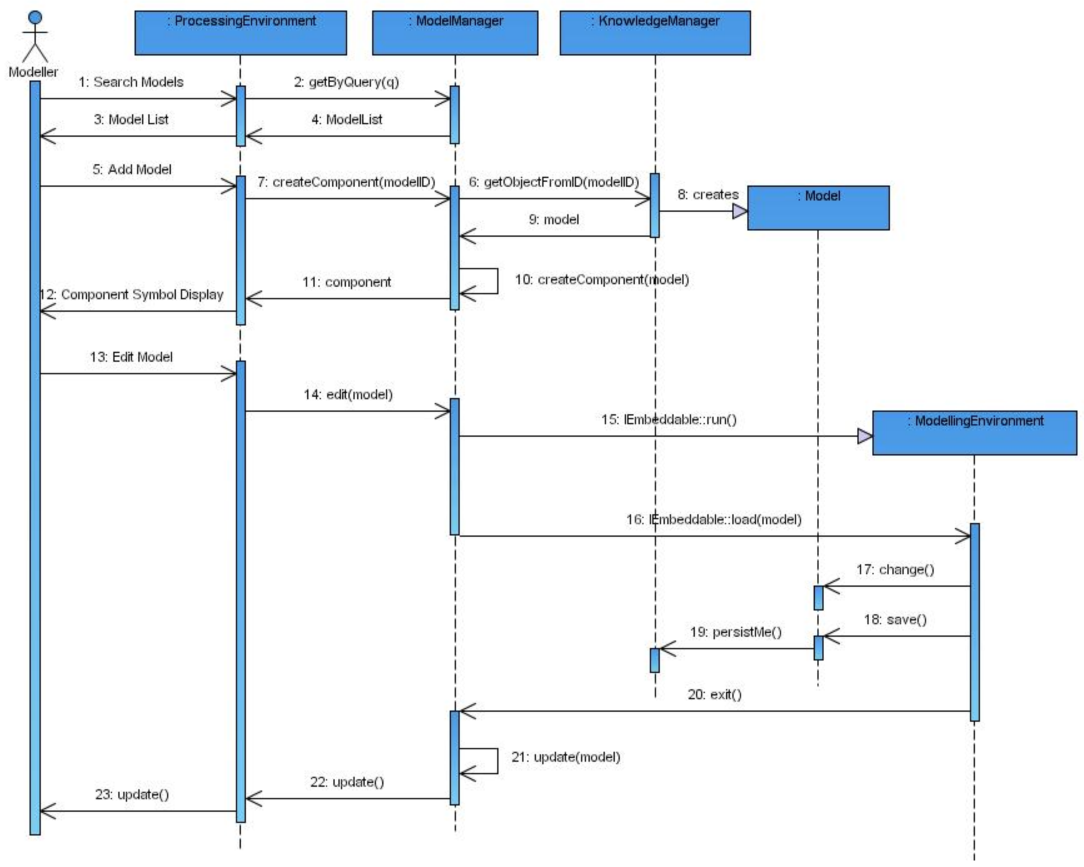

Figure 4.7. A diagram showing the interaction between the processing environment and the modelling environment, via the model and the knowledge managers.

4.12 Applications

Thanks to the Processing Environment of SeamFrame, it will be possible to use it for building end-user applications like *seam:LINK* (workflow construction and execution) and *seam:REF* (reference book application for viewing workflows and experiments). Moreover it will be possible to deliver applications targeting specific problems (SeamApps). In particular: APES, FSSIM, SEAMCAP, Landscape models and the integration of SEAMLESS with GTAP. This is not a limited set, in the future it will be possible to develop and deliver other applications exploiting the features of SeamFrame.

APES is a modular system of biophysical components that will allow quantification of biophysical impacts of management (management actions will be treated as inputs by APES) and weather variability (also an input) on agricultural systems, by estimating input and output coefficients, including production and externalities of production activities. APES will provide components for the main agricultural activities related to plant production (arable crops, orchards, agroforestry, grasses, vineyards) .

FSSIM models the technical aspects and the decision making at the farm level given specific biophysical conditions, using different sets of constraints to derive a set of feasible technological alternatives for each farm model. The objective function will represent farmers' behaviour in particular concerning risk. FSSIM connects the bio-physical processes modelled by APES, given the bio-physical and technological parameters, and the resource constraints of farm types.

SEAMCAP is a re-factoring of parts of the CAPRI model, on the basis of the SeamFrame architecture and design principles.

Landscape models will allow to quantitatively or visually assess changes in land use due to the consequences of the application of new policies and conditions. The interplay of SEAMCAP and FSSIM will produce a series of results which will provide the inputs to specialized landscape models that, in turn, help to study landscape spatial patterns and landscape dynamics, and to test hypotheses related to environmental or ecological objectives.

GTAP will also be integrated in SEAMLESS, even if not as tightly as CAPRI. The integration will happen at the data level, in order to allow agricultural policies, designed with SEAMCAP, to be inserted in the global economic model described by means of GTAP.

As anticipated, this list is not closed. The extendibility of SeamFrame is one of its key features. SeamFrame will deliver tools to build new applications using the core features of the framework.

References

Antoniou, G., Van Harmelen, F. 2004. A Semantic Web Primer. The MIT Press, Boston.

Brakkee, Brinkman, Gergersen, Hummel, Westen et al. 2004. HamronIT Document Series Part C – the org.OpenMI.Standard interface specification. The HarmonIT Document Series, IT Frameworks (HarmonIT) Contract EVK1-CT-2001-00090.

Gruber, T.R. 1993. A translation approach to portable ontologies. Knowledge Acquisition, 5(2), 199-220.

Klein, M., Fensel, D., van Harmelen, F., Horrocks, I. 2000. The relation between ontologies and XML schemas. Proceedings of the ECAI'00 workshop on applications of ontologiesa and problem-solving methods. August 2000. Berlin, Germany.

Ludaescher, B., Gupta, A., and Martone, M.E. 2001. Model-based mediation with Domain Maps. In: Proceedings of 17th Intl. Conference on Data Engineering (ICDE), Heidelberg, Germany, IEEE Computer Society.

Pasetti, A. 2002. Software Frameworks and Embedded Control Systems. Lecture Notes in Computer Science, 2231. Springer Verlag, Berlin.

Sanchez, D.M., J.M. Cavero, E. Marcos. 2005. On models and ontologies. 1st International Workshop on Philosophical Foundations of Information Systems Engineering (PHISE'05), 13 June 2005, Porto, Portugal.

Szyperski, C., Gruntz, D., Murer, S. 2002. Component Software: Beyond Object-Oriented Programming, 2nd Edition. ACM Press, New York.

Villa, F. 2001. Integrating modelling architecture: a declarative framework for multi-paradigm, multi-scale ecological modelling. Ecological Modelling 137, 23-42.

Glossary

Application: it is a software package obtained from the modelling environment. An application includes a custom GUI and it includes several components.

Architecture: Blue-print and styles to define structure;

Attribute: a property of a instance of a real world concept;

Class: in object oriented programming a class is a data type. It provides an implementation of the concept of an Abstract Data Type, that is a data type with a set of attributes (often called data members) and a set of operations (named methods) associated with the class. For instance, a *Crop* class might have a *biomass* attribute and an *irrigate* method.

Component: it is a model of a physical piece of the system being built. For example, source code files, DLL's, Java beans and other discrete pieces of the system may be represented as components. By building the system in discrete components, localisation of data and behaviour allows for decreased dependency between classes and objects, providing a more robust and maintainable design. Components may be either models or tools/utilities.

Conceptual model: describing (relevant) aspects of the real world in a functional context.

Conceptualization: Making (usually) a graphical representation of a model, a scenario, processes etc, for improved and joint understanding.

Declarative modelling: as opposed to the procedural/imperative approach to modelling, where the model equations are translated into code as a set of imperative statements. In declarative modelling we represent a model not as a series of assignment and control statements, but as a set of facts that are true about the model. Declarative modelling is strictly affine to the Prolog programming language.

Experiment: in the context of SeamFrame an Experiment defines all the information related to a simulation or an optimization run. This includes the data used as inputs, the model parameters, the simulation parameters (horizon, time step, etc). If the Experiment has been performed, it also contains links to the results.

Framework: productivity tool(box) to assemble components using architecture.

Indicators: outputs of either static or dynamic models which allow evaluating system performance.

Inheritance: in object-oriented programming inheritance is the property that allows a data type which inherits from another data type to include all its attributes without the need for their redefinition.

Integrated framework: an application which allows the evaluation of agricultural systems accounting for technical, environmental, economic and social indicators. One or more integrated frameworks will be the main deliverables of the integrated project.

Model: focused simplification of (a phenomena or process in) the real world;

Modelling environment: a software which allows developing components and applications. The modelling environments contains components which include a GUI for either component or application development

Modelling framework: the kernel component for static and dynamic model components use.

Object-oriented programming: The idea behind object-oriented programming is that a computer program is composed of a collection of individual units, or objects, as opposed to a traditional view in which a program is a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects.

Ontology: it is a specification of a conceptualisation. Once a modelling exercise has defined all the variables and relationships existing in a given model, this information can be stored in an ontology. There are many languages which can be used to represent an ontology, but RDFS/OWL is one of the most common since, being based on XML, it can be easily processed by computers.

Project: when referred in the use cases of the user Application/component developer (see below) a project is either a component or a system model which includes several components. When the user is a farmer or a policy maker, a project is the use of an application in specific conditions (a farm, a region etc.)

Programming interface: a set of component interfaces, packaged in a binary object together with their implementations, with the necessary documentation to reuse them in a software application.

Simulation: monitoring dynamics of attributes (in time and/or in space);

Software framework: see Framework.

Source: a source produces the value of an attribute;

System: a portion of the real world, with clearly defined borders, described both in a static and dynamic fashion by models

Appendix – Object-Oriented Programming Fundamentals

An object is a software entity that has *attributes*, *behaviour* and *identity*. Objects are members of a *class*. An object is often called an *instance* of a class. Attributes and behaviour are defined in the class. A class is a description of a set of objects. It is analogous to the concept of data type in non object-oriented programming languages. What's the difference then? It is called inheritance. A class may inherit attributes and behaviour from its predecessors.

Properties of an object oriented system

A software program can be defined as 'object-oriented' if it implements the following properties:

1. Abstraction
2. Encapsulated
3. Communication via messages
4. Object lifetime
5. Class hierarchies
6. Polymorphism

Abstraction

The complexity of the real-world is represented by a simpler model, which is described by the object. In the context of abstraction, an object is a thing or a concept. To adopt a good object-oriented programming style, one must start to 'think in objects': a crop is an object, a farm is an object, a rotation is an object, soil is an object.

Encapsulation

The meaning of encapsulation is to hide the internal details from the outside world. Thanks to encapsulation different data structures and algorithms can be adopted and implemented without changing the class interface. In other words, we hide the implementation from the rest of the world. Encapsulation is enforced thanks to the structure of the class itself: Each class has attributes (their values define the *state* of the object instance), which are analogous to fields in a record data type, and methods (which define the class' behaviour), which are analogous to functions, procedures. Encapsulation is achieved by making the class' state inaccessible from the outside. This means that the class attributes must be made *private*. For example, let's assume we have a class Soil which has an attribute named waterContent. If this was a standard record, a user of the class could write in her program a statement such as 'mySoil.waterContent' to access the value. In an encapsulated class this is not possible and it is forbidden. Access to attributes shall only happen through accessor methods (also known as *setters* and *getters*).

Messages

The interaction between objects is handled by sending messages. Messages are methods, operations we can perform on an object. The program 'tells' an object to do something. For instance: in an object-oriented programming language we write myStack.push('a'), that is Object.Message(Params), while in a procedural language we would have written something in the form Procedure(Variable, Params), where Procedure can be mapped to Message, Variable to Object.

Object lifetime

If Classes are similar to data types, objects can be assimilated variables, so they need to be initialised. While in existence they maintain their identity and state. Identity, each object has a unique name, State, each object is characterised by the values assumed by its attributes. Objects are allocated in memory (typically in the part of memory which grows dynamically and it's called 'heap'), so they can be eventually destroyed, freeing up memory again.

The creation of an object is called *instantiation*. An object variable is called an *instance*. The method that returns an initialised instance is called *constructor*.

Hierarchies

Object-oriented programming languages offer a number of powerful constructs which are useful to organise and represent our knowledge about the modelled system. Among these we find hierarchies, which allow to express a number of relationships which exists among classes.

The main relationships are:

- Association: a class has an attribute which puts it in relation with another class (a company has an employee)
- Aggregation/Composition: a class has attributes which define other classes as its parts (a car has 4 wheels)
- Inheritance: a class descends from the definition of another class (a bear is a mammal)

UML (Unified Modelling Language) is a powerful tool to represent such relationships. In the following Figures we present some examples of UML, in particular Class diagrams. Other important diagrams are State and Sequence diagrams, used to represent the dynamics of object interactions.

In Figure A.1 a UML class diagram is used to represent an inheritance relationship between SimpleClass and SuperClasss.

The box which describes SuperClass also contains the names of the attributes and the operations. A minus sign in front of an operation or attribute means it is *private* (they can be accessed only within the class), while a plus sign means it is *public* (they can be accessed from the outside). A hash sign means *protected*, that is, they can be accessed only from classes which are derived (inherit from) that class.

The inheritance relationship is also called an 'is-a' relationship.

Also in Figure A.1 we find an instantiation relationship. The variable named myObject is an instance of SimpleClass, and this is represented by means of the dashed arrow pointing to SimpleClass. The attributes in the object instance (e.g. 'id') assume specific values, that define the object's state.

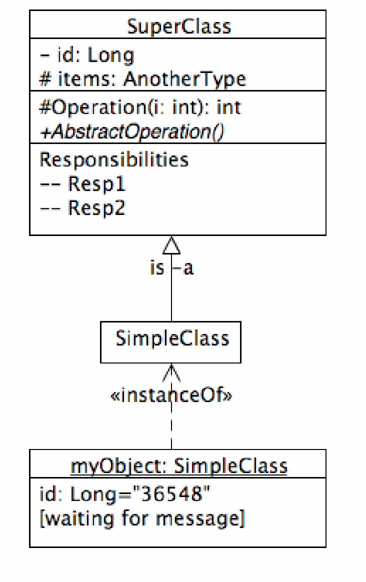

Figure A.1. Inheritance and instantiation. SimpleClass inherits from ClassName, while object is an Instance of SimpleClass.

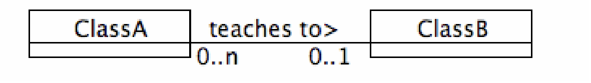

Figure A.2. An association relationship between ClassA and ClassB. The association can be named 'teaches to' and we can also quantify the relationship (from o to n instances of ClassA can teach to 0 to 1 instances of ClassB).

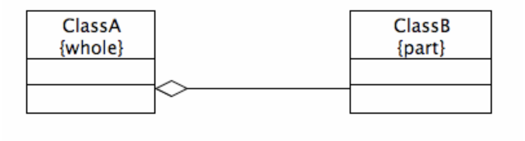

Figure A.3. An aggregation relationship. Class B is part of Class A, but Classs A can still exist even without Class B.

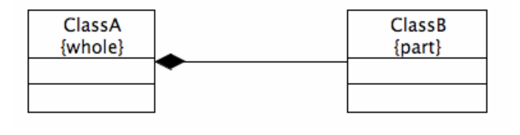

Figure A.4. A composition relationship. It's like the aggregation relationship, but in this case the whole does not exist without the parts.

In an association (Figure A.2) two classes are related by a named relationship. Multiplicity can quantify the relationship. The two classes are independent of each other. They simply collaborate and for this reason they must be (mutually) aware of the other class.

In an aggregation (Figure A.3) the two classes are not independent, and a whole-part relationship is expressed. For instance, a Library has a number of Books, and a Book can be part of the Library.

A composition (Figure A.4) is very similar to an aggregation, but the whole cannot exist without the parts, as in the example of a Book and its Pages. If you take away a Page from a

Book, it's not the same book anymore, while if you borrow a Book from a Library it's still a valid Library!

The relationships are expressed by means of attributes of a class. For instance, the class Library may have an attribute which is a List of Books.

Finally, a very important relationship is generalization/specialization, which represents inheritance relationships among classes (see Figure A.1 and Figure A.5). In Figure A.5 we represent the fact that Felines and Bears are Animals and that Panda Bears and Polar Bears are Bears, but they are also Animals.

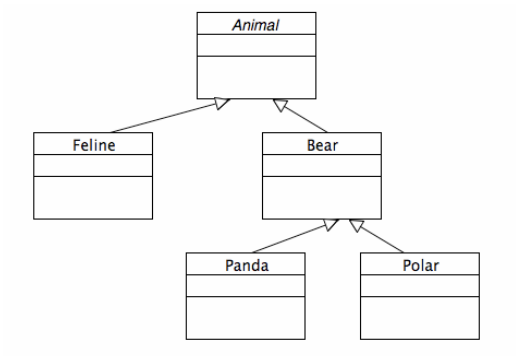

Figure A.5. An inheritance tree.

Polymorphism

Polymorphism (from Greek, it means 'many shapes') it is a fundamental property of object-oriented programming languages. Thanks to polymorphism the behaviour of specialised classes (classes lower in the inheritance tree) can be different from the one of the generalised classes. This means that the same method can have different implementations in the different classes (this is called *overriding*). Polymorphism thus allows to send the same message to different objects and have them behave differently. For example, think of a composed model of a Farm, which is composed of submodels for the Crop, the Animals, the Soil and so on. Each submodel is a subclass of the generalised class Model, that has the polymorphic method run() (also called a *virtual* method). The main simulation algorithm will simply dispatch the message run() to each model in the farm. This has a considerable impact on extensibility of the code: let's imagine we now want to add a new model component in the Farm, such as Orchards. We just need to implement the model class Orchard and to provide an appropriate implementation of the run() method and finally to add it to the list of Farm components. The simulation algorithm will not change at all, since it simply sends the message run() to each element in the collection.