Design of a Model-Driven Web Decision Support System in Agriculture: From Scientific Models to the Final Software

Ludovic Tambour, Vianney Houlès, Laurence Cohen-Jonathan, Valérie Auffray, Pierre Escande, and Eric Jallas

Abstract This chapter aims at introducing a new type of design of decision support systems (DSSs). The DSS presented here is a software based on client–server technology that enables great accessibility by the Web. Its conception flow has been established to be generic and not explicitly problem-oriented. In this way, once the first DSS is built, the creation of other DSSs will be easy and time-saving. The creation of the DSS requires the collaboration of different experts such as agronomists, computer specialists, and interface experts. Their communication is improved by the use of the formal language Unified Modeling Language (UML) throughout the process of software design. The relevance of the DSS comes from its use of scientific mechanistic models adapted to the users' needs and from a flexible architecture that allows easy software maintenance. The chapter is structured as follows: after the introduction, the second section will explain in detail the methods used to build the scientific models that describe the biological system. The third section describes the methods for the validation and implementation of those models, and the fourth section deals with the transcription of the models into software components processable in the DSS. Finally, the last section of this chapter describes the architecture of the client–server application.

1 Introduction

1.1 General Points

Modern agriculture has multiples stakes in economics, environment, society, health, ethics, and even geopolitics. Farmers must integrate more and more information with increasing production constraints. Decision support systems (DSSs) have been developed with the intention of providing farmers with relevant information for diagnosis assistance or more generally to facilitate

L. Tambour (✉)
SAS ITK, Montpellier, France
e-mail: ludovic.tambour@itkweb.com

strategic or operational decision-making in an inaccurate and/or uncertain environment.

These systems are particularly useful for pest control; indeed, these tools can provide risk indicators through the use of models running with meteorological data, or more basically, with a description of land history (preceding crop, soil management, etc). This enables the farmer to make a more accurate diagnosis and anticipate treatments or change his strategy. For instance, when the DSS foresees a low pest pressure, the farmer may favor environmental characteristics instead of efficiency when planning his phytosanitary program.

In France, the Plant Protection Service (Service de la Protection des Végétaux) has been working on disease prediction models since the 1980s and has included them in DSS since the beginning of the 1990s. More than 30 pests (insects or diseases) have been studied, and 21 models are currently used in France [1, 2]. Since then, many other DSSs have appeared. They have benefited from technological advances in data acquisition and treatment and from the Internet for the diffusion and update of information. Many agriculture-oriented institutes are involved in the development of DSSs such as research centers (INRA,[1] CIRAD[2]), technical institutes, cooperatives, and agropharmaceutical companies. The aims of DSSs are varied and may concern, for instance, variety choice (Culti-LIS), weed control (Decid'herb [3]), disease control (Sépale +), or nitrogen fertilization (Ramsès).

Despite this variety of offerings, DSSs are not currently used by farmers. The cooperative In Vivo, one of the major DSS providers, covers only 1.45 million hectares with its DSS for fertilization and plant protection [4] among 29.5 million hectares of agricultural lands in France. To be profitable, a DSS must be simultaneously reliable (scientifically validated) and user-friendly for the farmers.

With regard to plant phytosanitary protection, major effort is still required to adapt DSSs to practical needs of farmers. Special attention should be paid to the following three points:

1. *Integration of plant sensitivity*. Currently, most DSSs aimed at plant protection provide a pest risk indicator that depends on weather data but does not take into account the host plant. It would be more interesting to provide a risk indicator that considers plant sensitivity, for instance, by modeling the phenological stage as well as age and surface of the different organs.
2. *Integration of the major diseases for a specific crop*. Fungicides are indeed seldom specific to only one disease; each treatment often controls two diseases or more. Moreover, farmers strive, whenever it is possible, to group together the treatments to limit workload and fee. An efficient DSS should

[1] Institut National de Recherche Agronomique (French National Institute for Agricultural Research).

[2] Centre de coopération Internationale en Recherche Agronomique pour le Développement (French Agricultural Research Center for International Development).

therefore take all the major diseases of a crop into account, yet most current DSSs have been built for only one disease. The aggregation of different DSS outputs to define a global protection strategy is therefore difficult.
3. *Modeling the effects of the applied treatments*. Most models simulate disease evolution without any control. They thus provide useful information for the first treatment but they cannot indicate if other treatments are necessary afterwards.

The next point depicts the different steps and know-how involved in the design of a generic DSS that tries to avoid the drawbacks described above.

1.2 Generic Design of Decision Support Systems

The LOUISA project (Layers of UML for Integrated Systems in Agriculture) launched by the ITK[3] company is a general design for DSSs in agriculture with the following characteristics:

1. It integrates scientific models simulating the entire biological system. This requires the description of not only each element composing the system but also the interactions between those elements. The relevance of the information simulated by the DSS depends directly on the quality of the models.
2. The computing environment is flexible in order to integrate new formalisms, for example when agricultural knowledge improves.
3. The outputs of the models must be useful and easily accessible for the users, including farmers and farming advisers.

Therefore, the proposed DSS is a Web-based system. The Web-service is preferred over the stand-alone model because it enables an easy management of the diffusion of the tool, its update, and communication with linked databases such as those providing meteorological data.

The design of a DSS involves the following different specialists:

1. Agronomists who build the scientific model by conceptualizing the system and then creating a prototype.
2. Computer specialists who translate the scientific model into a processable model. This model needs to be easy to implement in the final DSS and must be simultaneously composed of independent components reusable in other DSSs.
3. Web interface specialists who define the functionalities of the DSS and design an appropriate and user-friendly interface considering the needs of users.

[3] See http://www.itkweb.com.

Those specialists have their own language, needs, aims, and tools. In such a multidisciplinary team, communication is the keystone of success. In our case, UML [5] is used by all specialists to facilitate this communication.

The developed DSS greatly depends on the agronomic knowledge and technologies. As these elements can evolve quickly, attention has been paid to flexibility throughout the design of the DSS.

The different stages of a DSS construction are depicted in the flowchart of Fig. 1.

The first phase is to establish the document of requirements specification, which defines the technical characteristics and functionalities of the complete DSS. Then two main objects are performed: the model and the interface. The model is first studied from a scientific point of view. A prototype is created to verify that the outputs are consistent. Once the model is validated, it is transcribed by computer specialists who are interested in the practical integration of the model into the DSS. The design of the interface can be done directly from the requirements specification. Thus, model and interface can be designed simultaneously and evenly matched into the DSS.

The next section describes in detail these phases on the basis of a practical case of DSS. For each phase, we will present our experience and progress, the encountered difficulties, and will justify our choices.

1.3 Development of DSS Software for Phytosanitary Plant Protection

This document illustrates a practical use of the generic DSS design LOUISA. The example taken here is a DSS designed for phytosanitary plant protection against diseases, although other aspects of the crop management could have been chosen as well.

The aim of this DSS is to enable farmers to choose their phytosanitary program with the support of risk indicators illustrating parasite pressure and short-term predictions of contamination risks.

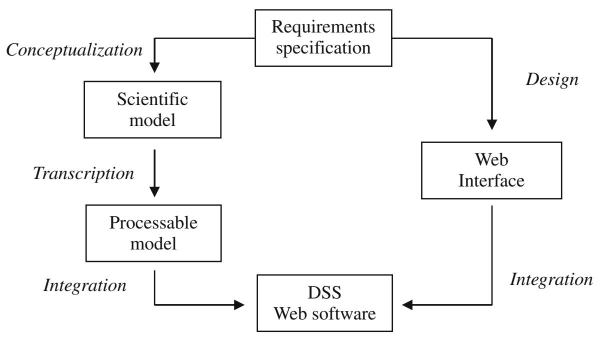

Fig. 1 Flowchart of the construction of the DSS

These indicators are obtained by means of scientific model simulating the behavior of the entire biological system. This system is composed of three submodels:

- Host plant
- Disease
- Phytosanitary products (pesticides).

The three submodels interact with each other and are driven by meteorological data including daily measures and short-term predictions. The accuracy of the model could be improved using an hour time-step for weather data, however we preferred the daily time-step as these data are far more easily obtained by farmers nowadays.

The main functionalities of the application are synthesized in the UML use-case diagram in Fig. 2.

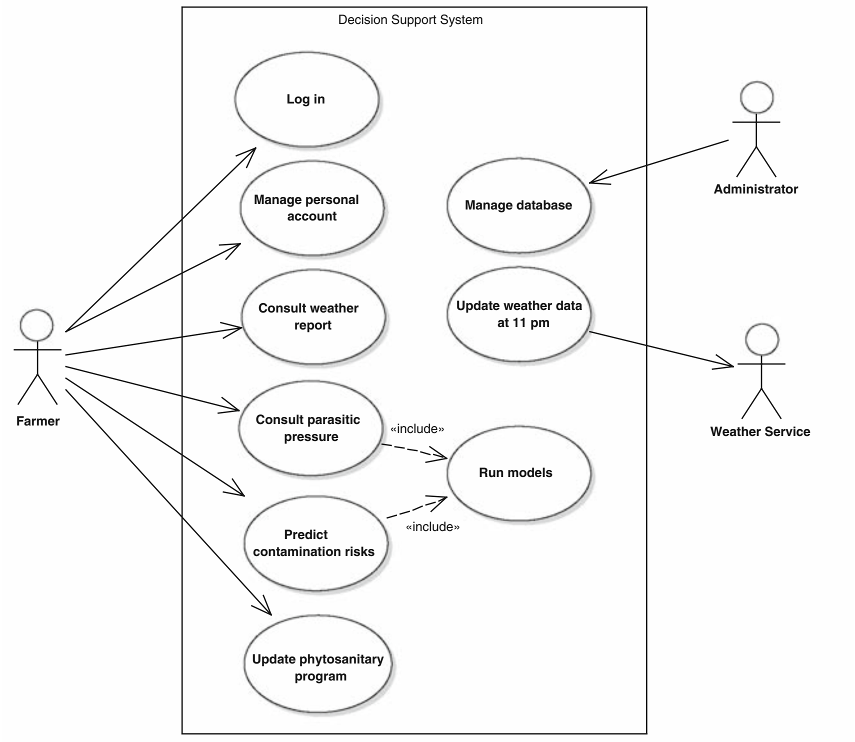

Fig. 2 UML Use-case diagram of the functionalities of a DSS intended for phytosanitary plant protection

This diagram depicts the different actors of the systems, human and non-human, and their interactions with the system. The main actor within the software is the farmer. The functionalities related to this user are

- Connection and disconnection to a server
- Personal account management including registration of land characteristics and cropping practices
- Access to weather data (daily records and predictions)
- Access to current parasitic pressure thanks to simulated indicators
- Access to contamination risks predictions thanks to simulations using predictive weather data
- Update of the phytosanitary program according to the DSS suggestions, taking into account simulated risk indicators and phytosanitary products regulations.

The second major actor is the administrator, who must manage the database.

The weather service is the last actor and, more precisely, a non-human secondary actor. The DSS must be connected to the weather service to get measured and predictive weather data. "Update weather data at 11 pm" is an inner use-case executed by the system, because the update is made at this fixed hour without the intervention of an actor or another use-case.

The next sections describe the different phases of the DSS design. First, the scientific model is described globally and decomposed into three submodels (e.g., host plant, parasite, and phytosanitary protection) interacting with each other and their environment. Another section deals with the scientific models set-up and validation. The next section describes the transcription of the scientific model into a processable model using UML, and finally, the last section concerns the software architecture.

2 Design of the Scientific Model

2.1 Description of the "Plant–Parasite–Phytosanitary Protection" System

Developing the scientific model means formalizing the system and describing the relationships between its components with equations. In the following example, the system has three components (the plant, the parasite, and the phytosanitary protection) that interact together and with the environment, which includes both climate and soil.

To develop the scientific model, different information sources are used including scientific publications, farming magazines, field data, and expert knowledge; each source has advantages as well as drawbacks.

In general, scientific publications describe a particular phenomenon accurately thanks to models, in the best case, or at least with a qualitative description.

Concerning our example, it is possible to find models describing the influence of temperature and humidity on disease development in the scientific literature but it is not sufficient for describing the entire system. For instance, no model is available that treats the influence of plant growth on phytosanitary protection. Moreover, the final aim of published models is more often the understanding of biological phenomena rather than their use in DSSs. Before integrating a published model in a DSS , it must be verified that the inputs are consistent with the DSS specification requirements. Moreover, published models may not be generic but only valid for specific experimental conditions.

Farming magazines provide general information that cannot directly be used to construct models but may help in formalizing and expressing hypotheses. For instance, advice concerning pesticides spray frequency can be useful information with respect to the phytosanitary protection model and its interaction with the crop and the climate. This information source is also helpful for keeping in touch with the concerns of farmers.

In the case when accumulated information is not sufficient for describing a system, it is recommended to interact with experts and to use field data.

The conception of the scientific model is not a simple bibliographic search and compilation. Published models are not prefabricated bricks that can be assembled to build the DSS. To be integrated in a coherent system, models must first be adapted with specific regard to their accuracy in relation with the DSS needs. Moreover, lack of information has to be filled in with new formalisms coming from hypotheses that will be tested and validated. Even if specific published models are useful, it is important to consider their collaboration for modeling the entire system. The scientific model is just an element that must serve the final aim of the DSS, taking into account technical constraints and specific concerns of farmers.

For a DSS designed for phytosanitary protection, a few judicious risk indicators have been selected:

- Parasitic pressure
- Phytosanitary protection level
- Plant sensitivity that may depend on specific organs and their age.

These risk indicators must be part of the outputs generated by the model. The UML class diagram in Fig. 3 expresses the relationships between concepts/entities and the outside user.

The system has five components, which are the submodels plant, parasite, and phytosanitary protection added to the soil and the climate. In this conceptual view, climate has an influence on the four other components. For instance, rain refills the soil with water and washes off phytosanitary products. The plant is the medium for the parasite's development. The phytosanitary protection destroys the parasite but its efficiency may decrease because of rain and plant growth (the latter phenomenon is called "dilution" in the diagram). The soil provides the plant with water and nutrients and modifies the

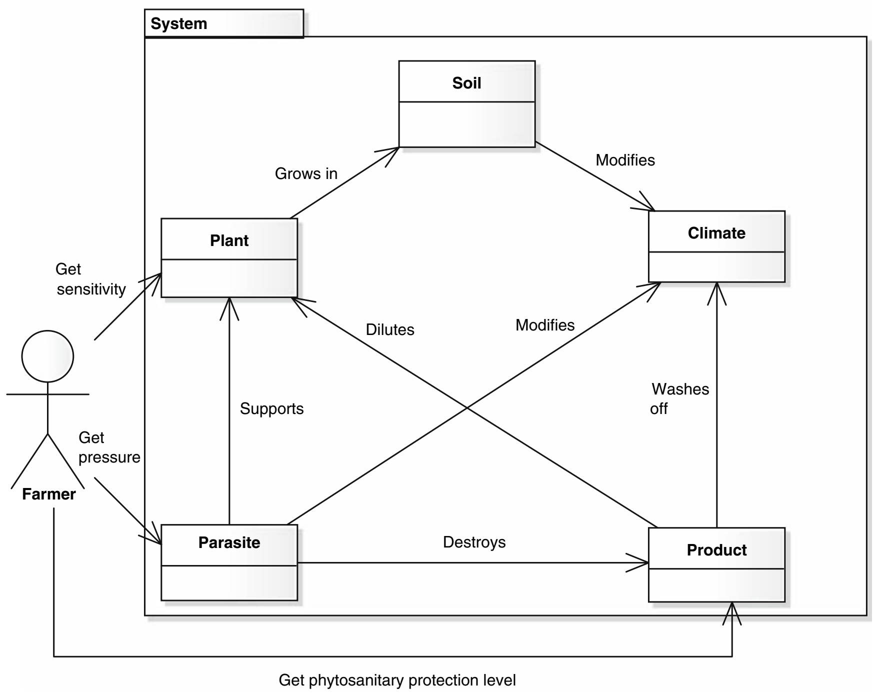

Fig. 3 UML class diagram of the whole system and its components

microclimate. For example, a soil with high water retention can increase humidity and thus favor the development of some parasites.

For the moment, soil behavior is not explicitly modeled. As a first approach, the plant grows without soil-limiting factors (i.e., neither water stress nor nitrogen stress, for example). In the same way, the soil influence on the parasite's development (through action on microlimate) was not modelled, because of the scarcity of knowledge on the issue.

It could, of course, be possible to add more relationships in this diagram. For example, the influence of the parasite's development on plant growth or the effect of the plant on microclimate could be modeled. For an initial approach, we decided to model only the most influential phenomena regarding the three selected risk indicators. Once a prototype is built, the outputs of the model can be compared with observed data. In case of discrepancies due to excessive simplification, the model can be made more complex by adding more relationships.

2.2 *The Plant Model*

Plant models can be divided into three groups according to their abstraction level:

- *Big leaf models*. In these models, the plant is considered as a homogeneous structure. Organs, such as leaves, are not modeled individually but aggregated into one large organ. This is for instance the case for the "CERES-maize" [6] and "STICS" [7] models.
- *Topological models*. These models describe the structure and the relative position of organs with relationships such as organ X bears organ Y. Each organ is described individually regarding its behavior and its attributes. For example, the rice model "EcoMersitem" [8] and grapevine model [9] can be cited.
- *Architectural models*. These models are even more accurate than topological models by adding information about organs geometry and spatial position. An example of this kind of model is "COTONS" [10].

An output of the model concerns plant sensitivity to a particular parasite. This sensitivity depends on the plant's phenological stage as well as on the age and surface of susceptible organs. For this reason, a topological model seems to be the most appropriate for the DSS. Indeed, a big leaf model cannot simulate properly the age and surface of each organ (e.g., leaves), and all the details provided by an architectural model do not seem useful to simulate the selected indicators. Modeling at the organ scale is also convenient for flexibility. For example, if knowledge concerning fruit growth improves and gives birth to a new model, it is possible to integrate it without rewriting the entire plant model.

The topological model can describe the plant's structure. However, it is not sufficient for the DSS needs, as plant evolution modeling involves other submodels (Fig. 4):

- *Phenological model*. This submodel aims to determine the major plant phenological stages such as growth start, flowering, fruit setting, and so forth. The phenological model drives the appearance of organs type (e.g., fruits appear with fruit setting), and the equations describing organs evolution may change according to phenological stage.

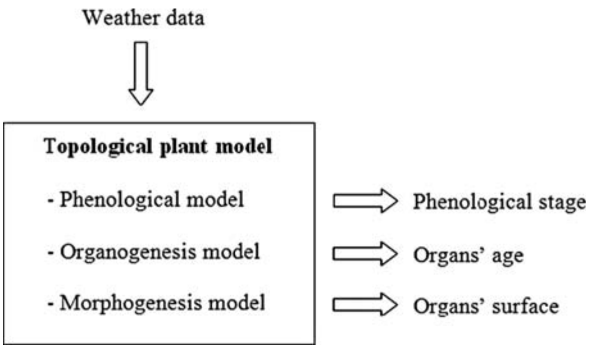

Fig. 4 Inputs and outputs of the plant model

- *Organogenesis model*. This submodel determines the apparition of new organs. It adds dynamics to the plant model by describing the evolution of plant topology with time.
- *Morphogenesis model*. This model simulates the growth of each organ individually.

The chosen topological model considers the above-ground part of a plant as a group of phytomers (Fig. 5). A phytomer is a cell cluster that will evolve into organs having the same age, and it can be considered as the basic plant element. Organs coming from a phytomer can vary with plants or within a plant. For example, a plant can have vegetative phytomers composed of a node, an internode, a leaf, and a bud and fruiting phytomers composed of a node, an internode, a leaf, a flower, and a bud.

A plant's structure can thus be modeled in terms of axes (e.g., branches for trees and tillers for *graminaceae*). Each axis is composed of a series of phytomers, and each phytomer may bear another axis as its bud itself can evolve into a new phytomer. Assuming this organization, describing a plant consists of determining the composition of its phytomers and defining the rules driving their evolution. For instance, the following questions have to be answered: where are the fruiting phytomers? In what conditions does a bud develop into a new phytomer? The rules describing the plant's structure can be determined with support of a statistical analysis using Markov chain models [11, 12, 13].

The evolution of an individual organ is influenced by its surroundings and is narrowly linked with the rest of the plant [14], meaning that the organogenesis and morphogenesis models have to be regulated by the status of the whole plant. Regulations are taken into account through parameters included in the equations describing organogenesis and morphogenesis.

If there is no limiting factor, phytomer emission (organogenesis) and organ growth (morphogenesis) are only driven by thermal time (timescale depending on temperature), although the equations can vary with the type of axis. For instance, equations describing the phytomer emission rhythm (also called

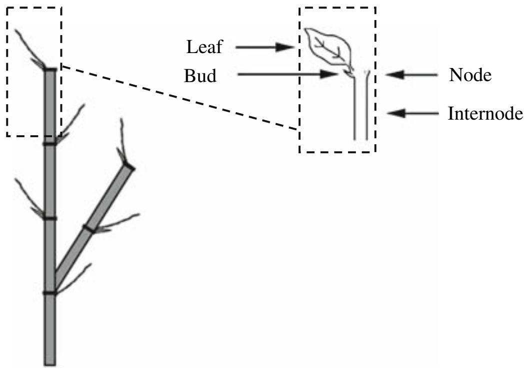

Fig. 5 Topology of a plant modeled as a group of vegetative phytomers

plastochrone) and the final size of leaves can be different between a phytomer constituting the main axis and another phytomer constituting a secondary axis.

Phenology has a major impact on the plant and on its organs. For example, during the vegetative stage, only vegetative phytomers appear, whereas during the reproductive stage, both vegetative and fruiting phytomers can appear. Phytomer emission rhythm and vegetative organ growth can also be slowed down from fruit setting on because of preferential carbohydrates allocation to fruits.

Water stress and nitrogen deficiency can also be factors slowing down phytomer emission and organ growth. Stresses are generally modeled at the plant scale.

2.3 Parasite Model

The risk indicators selected for the DSS concern phytosanitary protection, plant sensitivity, and parasitic pressure. The latter element is the most important factor to decide phytosanitary products application, which is why the parasite submodel interacts with all the other submodels. As the DSS is not designed to deal with yield prediction, the effects of the parasite on the plant are not modeled. Thus, it is also assumed that there is no retrospective effect on the parasite. In reality, as parasites weaken or destroy parts of the plant that represent their support, their own development should also be affected, but this phenomenon is voluntary neglected.

The following description of the parasite model concerns diseases in general resulting from bacteria, viruses, or fungi. Parasites transform into several biological stages during their development. For instance, fungi have two principal forms: sexual and asexual. Other forms can be added, specialized in propagation, development, or survival during critical periods (winter for temperate zones, dry period for tropical zones). The evolution from one form to another is modeled more or less accurately depending on the chosen formalism.

Three main kinds of models are used to simulate disease evolution:

- Epidemiologic models
- Mechanistic models
- Intermediate models based on the parasite biological forms.

Epidemiologic models [15] describe disease progression by means of a single equation with the following type:

$$\frac{dX}{dt} = kX(t)\cdot(1 - X(t)),$$

where X stands for the modeled variable that is often the surface colonized by the disease, t stands for the time, and k is the disease amplification factor. This differential equation is solved by approximating the temporal derivative term by finite differences method resulting in:

$$\Delta X = k \cdot \Delta t \cdot X(t_n) \cdot (1 - X(t_n)),$$

with $\Delta X = X(t_{n+1}) - X(t_n)$standing for the variation of X during $\Delta t = t_{n+1} - t_n$ corresponding with the model time-step.

Factor k is difficult to determine because it varies with climate and plant evolution. In epidemiologic models, disease progression is modeled globally without distinguishing the different parasite forms.

Mechanistic models describe the different physical, chemical, and biological processes involved in disease development and simulate accurately the parasite's biological cycle (e.g., [16, 17]). This kind of model requires a meticulous description of plant structure, microclimate, and the parasite's biological processes. Such a complexity can be useful for research purposes but it requires too many inputs to be integrated in the DSS. Moreover, increasing complexity often leads to less reliability and may result in erroneous outputs [18]. These models are also specific to a particular parasite and are hardly adaptable to other parasites.

An intermediate complexity model consists of representing each parasite's biological form as a stock containing a quantity of items. The evolution of items from one form to another is modeled but not necessarily in a mechanistic way. For instance, to pass from an ungerminated to a germinated fungus spore, an empirical relationship calculating the germination rate according to climate can be used instead of the description of the physical and chemical reactions involved. The number of items in a stock can also increase by intern multiplication without implicating the other stocks. This intermediate complexity is easier to understand and set up compared with mechanistic models and more factors can be explicitly integrated compared with epidemiologic models. Furthermore, this type of model is more generic. Indeed, the number of stocks and equations driving the transitions can be adapted to model another disease. For all the above-mentioned reasons, this kind of model has been chosen.

Most diseases start with an initiation phase followed by an amplification phase corresponding with epidemic cycles. Figure 6 depicts this general representation of diseases with the Stella formalism [19]. Rectangles on the left-hand side (stages 1 and 2) illustrate the stocks during the initiation phase; rectangles forming a cycle (stages 3 to 6) illustrate the amplification phase. The movement of items from one stage to the next is controlled by a flow. This flow can depend on factors related to weather conditions, the plant, or the disease itself. This dependency is depicted by a circle named "Factor" on the figure. Each stock can also be affected by mortality, depicted by a flow named "Mort." for mortality. There is no stock following mortality so that items concerned by mortality are destroyed. Of course, Fig. 6 has only illustrative value, and the number of stocks depicted will depend on the considered disease.

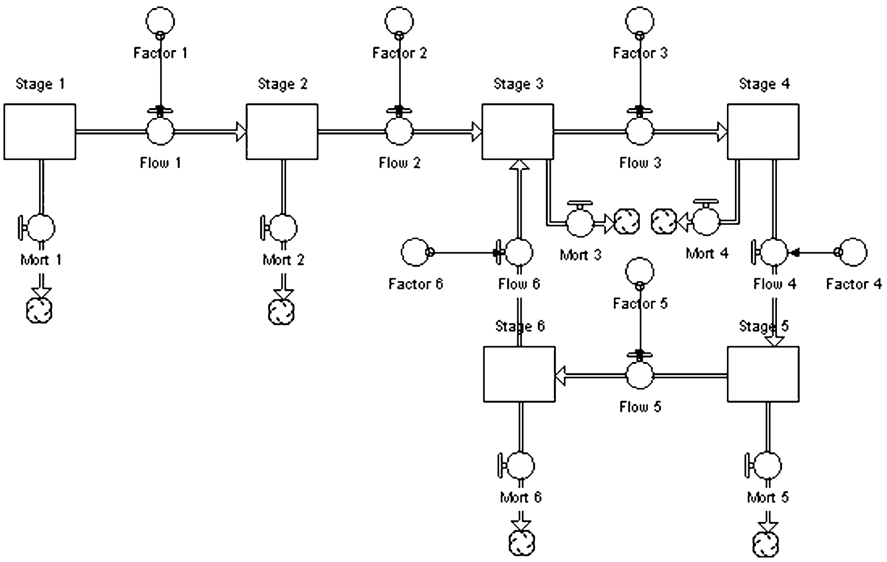

Fig. 6 A virtual parasite's biological (according to the Stella formalism [19])

Disease development can also be considered from the plant point of view. In this case, stocks would represent the different status of plant (e.g., healthy, contaminated in latency, contaminated and infectious, contaminated and no longer infectious), while from the parasite's point of view, stocks symbolize the different biological forms such as larval and adult forms for insects. Items contained in these stocks can be surface units for the plant and individuals for the parasite. The choice of this point of view will depend on knowledge availability.

It is then necessary to determine the different factors influencing disease development.

Of course, phytosanitary protection has a major influence, although *a priori* all crop management techniques can positively or negatively influence disease development. For example, pruning modifies susceptible organs available for contamination. Therefore, the parasite submodel must use outputs from the plant and the phytosanitary protection submodels as inputs.

Weather data must also be used as inputs, because the climate influences the disease development. It is thus necessary to analyze *a priori* the weather data in terms of availability, accuracy, and costs. A simple model using few inputs should be preferred compared with a more complex model using inputs hardly available or inaccurately measured [18].

The influence of crop location characteristics on disease development can be modeled by an impact on microclimate. For instance, the water absorption and retention characteristics of the soil influence the humidity in the plant vicinity. Another example is the land slope that influences the plant's microclimate and the radiations intercepted by the plant.

This point shows that the pest model choice depends on its biological cycle, its action on the plant, and, most of all, on the existing scientific knowledge. Even if different kinds of model can be distinguished, there is no fixed methodology for building a pest model. It is essential to adapt the model to the project's global aim. In particular, special attention must be paid to the accuracy needed because a useless increase of complexity often leads to lower reliability.

2.4 The Phytosanitary Protection Model

Phytosanitary products contain chemical molecules operating on the metabolism of parasites in order to limit their development or to stimulate plant defense processes. As these molecules are often toxic and polluting, they should be applied only when necessary (Integrated Pest Management concept). The aim of the DSS is precisely the adaptation of applications of phytosanitary products to the needs.

Existing DSSs are usually focused on pests without taking into account phytosanitary protection. Yet the necessity of a phytosanitary product application depends not only on the parasitic pressure but also on the current plant protection level. It would be useful to determine an application remanence, taking into account the plant's development, climate (including rain and radiation), and characteristics of phytosanitary products.

The phytosanitary protection model has to simulate a protection level that is one of the three selected risk indicators for the DSS. This protection level takes into account:

- Active period of products corresponding with the duration of effectiveness of molecules.
- Plant growth: A product applied at time t will be less efficient at $t + \Delta t$ because of the plant's volume and surface increase. The effect of the plant growth depends on product properties.
- Rain wash-off.
- Spraying quality depending on equipment and weather conditions, in particular wind and humidity.

There are three types of phytosanitary products:

- *Protective products*. They have a surface action and protect only the sprayed surfaces.
- *Eradicant products*. They can penetrate plant tissues and are thus protected from rain wash-off. Their penetration speed depends on the characteristics of products and on weather conditions.
- *Systemic products*. They are capable of moving throughout a plant using the vascular system and thus they can protect organs created after spraying. They are very useful when the crop has an intense growth phase.

Each product operates on specific stages of a parasite's cycle. For instance, in the case of fungicides, protective products usually block spore germination: using the formalism depicted in Fig. 6, this action can be modeled by decreasing the flow linking the stocks "ungerminated spore" and "germinated spores."

To model plant phytosanitary protection, each product has to be characterized in terms of biological action on parasites, active period, rain wash-off sensitivity, and efficacy decrease due to crop growth. Thanks to this submodel, the DSS can provide a graph simulating the decrease of plant protection with time.

3 The Scientific Model 's Set Up and Validation

3.1 Principle

Once the formalisms are chosen, the scientific model needs calibration. This operation consists of determining the parameters values that minimize deviations between simulations and observations.

The parameters values can be specific to particular situations such as the geographic zone, the cultivated variety, or, of course, the parasite. In reality, these situation-dependent parameters express variations of factors that are not explicitly modeled and their value must be adapted according to the case [20]. For example, rather than modeling the influence of soil on parasite development, a set of parameters can be proposed for each type of soil.

A model can have an important number of parameters but they do not have the same influence. Some parameters will have a low impact on the model; it is thus unnecessary to spend energy to optimize these parameters. On the contrary, other parameters are essential and have a large impact on simulations. A widely used method for determining the influence of parameters is called *sensitivity analysis* [21, 22].

Finally, the model's performance has to be analyzed in terms of prediction quality and accuracy regarding to the final user needs.

The following section briefly presents the methods for sensitivity analysis, parameter estimation, and model assessment. All of these methods are worth implementation during model design. For more information on this subject, an exhaustive description of good agronomic modeling practices has been previously published [23].

The methods described below require high data quantity. Quality and quantity of data directly impact calibration quality, model accuracy assessment, and consequently model relevance. To be exploitable, a data set must be composed of variables corresponding with model inputs, observations corresponding with outputs, and, if possible, intermediate variables corresponding with variables calculated by the model during the simulation process. The latter variables may refine the diagnosis and may detect the origin of discrepancies. In the DSS, for example, leaf area is not an output shown to the farmer, but it is important to control that it is correctly modeled because it influences parasite growth.

3.2 Methods Used for Sensitivity Analysis, Calibration, and Validation

Two types of sensitivity analysis are distinguished: local methods and global methods [24].

Local methods explore the model's behavior in the vicinity of an input parameter set. They are used when an approximate value of input parameters is known.

Global methods explore all the input parameters' possible values constituting the parameter space. They often use stochastic methods based on random number selection inside the parameter space (Monte Carlo methods). They are preferably used if there is little prior information on input parameter values and give an accurate idea of importance of parameters. A major drawback of some global methods is that the effect of a parameter can be hidden if the model is strongly nonlinear. Some methods are model-independent [21] and cope with nonlinearity and interaction between factors (e.g., the FAST method [21]). Global methods generally require a large computing time.

Local and global methods provide sensitivity indicators allowing classification of parameters according to their influence on the model, but the obtained results depend highly on the explored space.

Parameter estimation consists of finding the parameters values minimizing discrepancies between model outputs and observations. The cost function is a metric of that discrepancy and is to be minimized. In the famous least-square method, for instance, the cost function is the sum of squared errors, but many other cost functions exist that are adapted to different purposes [20]. Observed data must cover various situations and be numerous enough, with the ideal number depending on the number of parameters to be estimated. Many algorithms can be used for parameter estimation such as the Gauss–Newton method, simulated annealing method, or genetic algorithm. The choice of both cost function and optimization algorithm is not easy. It can be interesting to test several couples but the most important point is data quality.

After model calibration, the next stage is the model performance assessment. Concerning a DSS, two complementary analyses should be made [25]. The first one classically compares model outputs with observations. It may detect bias or dispersion and assesses calibration quality. The data used here must be different from those used for calibration. The second analysis concerns recommendations performed by the model and requires an adapted observed data set. For instance, if the DSS recommends phytosanitary spraying dates, it would be useful to have experimental data sets assessing phytosanitary protection with different spraying dates. The model's ability to perform recommendations is validated if the recommended dates are close to the observed spraying dates leading to the best phytosanitary protection.

Once the model's reliability is validated, it is interesting to compare the DSS recommendations with a classic phytosanitary spraying program. The aim of this phase is the assessment of the DSS benefits, for example, in terms of cost, plant protection efficiency, or environmental impact.

3.3 The Choice of Modeling and Validation Tools

The design of scientific models requires appropriate modeling tools. In our experience, an appropriate tool should have the following capabilities:

- To enable quick and easy modeling of agronomic concepts
- To be sufficiently flexible to easily modify the model when changing assumptions
- To be executable in order to check the modeling assumptions by simulation
- To provide a flexible environment facilitating result validation (e.g., graphics, analysis functions of results).

Many tools and languages are available for model set up. The following is a historic classification of these tools. For each type, the advantages and drawbacks are discussed.

The first supports used for modeling were classic procedural languages or object-oriented languages (e.g., C [26], C++ [27], JAVA [28], Delphi [29]). These languages are still intensively used, even if they are not well adapted to model conception and development. They are indeed too verbose to be quickly implemented and modified. At least 50% of a model written with these languages is destined to purely computing aspects (e.g., variables' declaration, tables' size management), decreasing the attention that is paid by the agronomist in his modeling task. Moreover, as those languages do not supply a validation environment, validation methods must be computerized.

To avoid those computing problems, new languages appeared, intended for numerical computing (MATLAB [30], R [31], Scilab [32]), and formal calculus such as Mathematica [33], Maple [34]). These languages are similar to programming languages but they suppress pure programming aspects. For instance, variables are not declared, table size is automatically managed, and simple functions allow table manipulation. Interpretive languages also supply a wide range of mathematical functions assisting modeling and validation. These latter two tasks are concise and also easy to set up and modify. Validation is also facilitated by command lines coupled with the execution environment. After a simulation, all the simulated variables are available from the workspace and, by means of command lines, the user can rapidly visualize, manipulate, and compare the results. In return, this conciseness makes the code difficult to understand and to maintain. Moreover, as these languages are interpretive, their execution is slower. For example, a well-written MATLAB program is at least 10 times slower than a C++ program, and this factor can reach 1,000 if the MATLAB program is written without taking care of executive efficiency.

If numerical computing languages are intended for modeling, most modeling agronomists are resistant to the use of textual languages as modeling support. To overcome this problem, graphical modeling environments have been created (Simulink [30], Stella [19], ModelMaker [35]). By this graphical approach, a model can be decomposed into submodels (describing a process) linked to each other by wires symbolizing dependencies. This graphical modeling environment is easier to handle for a neophyte and further favors the understanding of the model's general concepts inside the multidisciplinary modeling team or with external partners.

On the contrary, these graphical environments are not flexible to change, especially when models are complex or are decomposed into fine granularity components. Another drawback is their limited expressive power. Indeed, available tools impose construction rules that may not be adapted to agronomic concepts. That is, for instance, the case when modeling conditional aspects or states (in the UML sense of state) or for dynamic construction/destruction of components (e.g., simulation of appearance of new organs). Moreover, graphical modeling environments generally execute simulations far slower than do compiled or interpretive programs.

Agronomists also use spreadsheet programs (e.g., Excel [36]) for modeling. These programs allow immediate visualization of numerical results, but they are not adapted for complex models due to their restriction to spreadsheets, and their equations edition is not sufficiently clear. However, they are still very convenient for the quick modeling of simple phenomena.

To develop our scientific model, we chose to use MATLAB because of its flexibility. This flexibility involves the first model description, its assessment, and its possible modifications (change flexibility). Prior to the MATLAB model, a preliminary graphical description was generated with Stella. This software can also simulate, but we only use the graphical functionality to study how the model can be decomposed into submodels of approachable complexity and to depict the phenomena that need studying. Stella graphical aspect also favors communication between people involved in the DSS project.

4 Software Architecture of the Scientific Model

The MATLAB model aims at checking the reliability of scientific concepts prior to their implementation into the final software. Even though MATLAB provides the environment needed for modeling and analysis, this language is not suitable for the complex DSS software because of the following problems:

- *Integration*. A MATLAB model cannot be executed out of the MATLAB environment. Thus, it is difficult to link a MATLAB model with other software components such as database, Web services, or functional components written in another language.
- *Performance*. MATLAB executes slowly, yet response time is decisive for software acceptance by users.
- *Robustness*. MATLAB is devoted to numerical computing but not to programming. It is thus not robust enough for software design.

Therefore, the MATLAB scientific model has to be transcribed into a model using a programming language that we call the processable model. For our DSS, we have chosen the JAVA [28] language because it is well adapted to Internet applications programming. The transcription from a scientific MATLAB model to a processable JAVA model cannot be done directly. Whereas the scientific model aims at simulating reliable outputs, the JAVA model must produce a model with the same functionality that is able to be integrated with other software components and allow quick execution and flexibility regarding change. This latter point is essential because the scientific model continues evolving, and the processable model must be able to go along with this evolution. Using design patterns [37] is a good practice for achieving flexibility. In software engineering, a design pattern formally describes (via UML diagrams) a standard solution to a general recurrent computing problem.

This fourth section deals with the processable model design from the MATLAB model. This design phase uses the UML formalism as specification support, independently from the programming language. Consequently, the work presented here can be generalized to any object-oriented programming language.

4.1 Class Diagram of the Plant–Parasite–Phytosanitary Protection System

The system is studied with a modular approach [38], allowing the decomposition of a complex modeling problem into subelements having a more controllable complexity. The subelements, called components or classes, are autonomous and can communicate with each other. A component, for instance, can be a model of the system (e.g., plant, parasite, phytosanitary protection) or a part of these models (leaves or fruits). Components contain the objects' characteristics constituted by attributes and methods. Attributes correspond with the physical elements to model, and methods correspond with the physical processes.

We will now define the relationships between the classes of the studied system. As the soil influence is not considered at the present time (cf. Section 2.1), it is not depicted in the diagrams in order to lighten the figures.

Explanations given here have a generic value and can be applied to any system decomposed into classes.

As recommended previously in [39], each class diagram element implements an interface depicted by circles as shown in Fig. 7. The use of interfaces creates a "plug and play" architecture where each component is interchangeable without changing the whole implementation; components to interchange only have to share the same interface. For instance, if the modeling agronomist prefers using a big leaf plant model instead of the topological model first chosen, the computer scientist only replaces the "PlantTopo" class by a "PlantBigLeaf" class without modifying the other system components and connections. As a result, the system is very flexible. Moreover, interface creation only requires minor additional implementation time, which is not significant compared with the time saved to implement modifications. For this reason, the plug and play architecture is systematically used for our complex systems implementation.

Figure 8 depicts the general class diagram with the introduction of a simulation controller. The simulation controller aims at controlling the model execution flow and the tasks scheduling. The "singleton" design pattern [37] is used to instantiate the controller so as to ensure its uniqueness. Indeed, if several instances of the controller command the same system, a conflict may be induced.

The control job can be decomposed as follows using pseudo code:

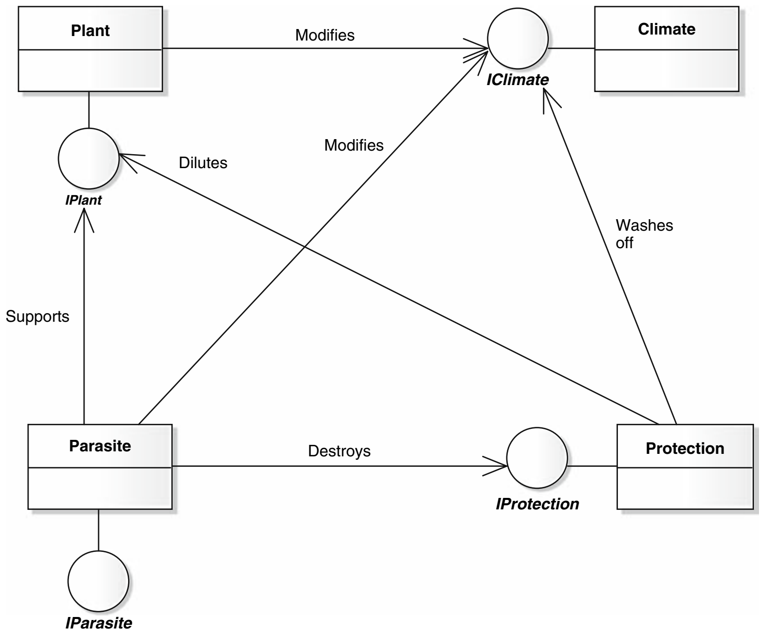

Fig. 7 General class diagram: "plug and play" architecture

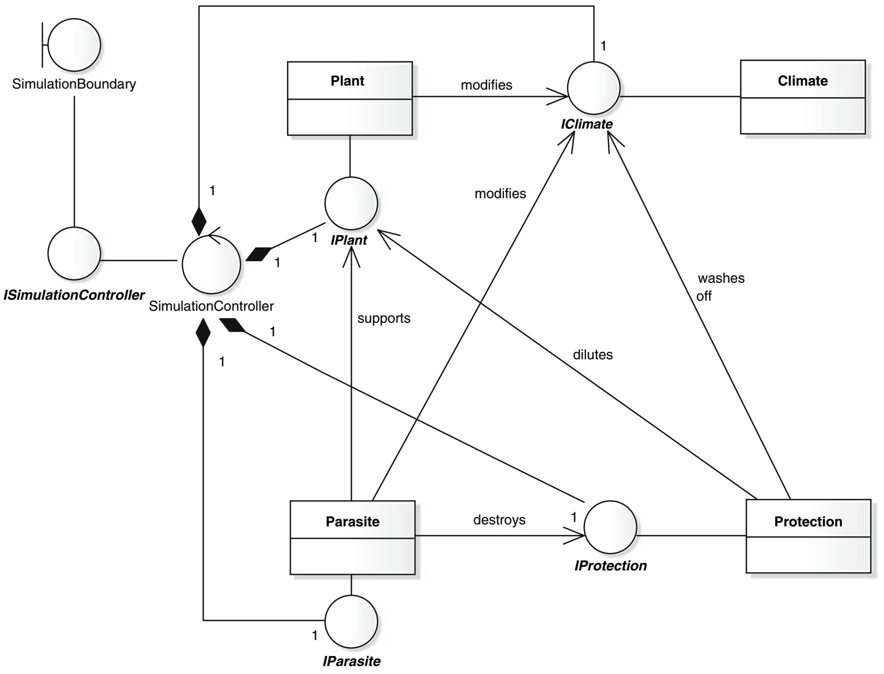

Fig. 8 General class diagram: inclusion of the simulation controller

- Creation of the different components and realization of the components' connections
- Components initialization
- For *date* going from the first simulation date until the last date:
  - Run the execution of the simulation step corresponding with the simulation date *date* for each component
  - Determine the next simulation date *date*
- End of the loop
- Components finalization (e.g., save the simulation results).

The creation of concrete class instances is delegated to a factory and follows the factory design pattern [37]. According to the simulation query, this design pattern creates different simulation scenarios (including the phytosanitary protection or not, instantiating the crop "wheat" or "maize," for example).

A multifrequencies management is added to the modular approach [38] so that components can run with different time-steps. For example, plant growth can be modeled at a daily time-step, whereas the light interception component runs with an hourly time-step in order to take into account the solar angle variations during the day. To take multifrequency into consideration, the execution of a component's simulation step is unlocked in two stages:

- If *date* is equal to the next component simulation date *nextSimulationDate*, execute:
  - Component's attributes update
  - *nextSimulationDate* update
- End of the IF block.
  When the component has a periodic execution, the next simulation date can be calculated as follows:
  - *nextSimulationDate* = *date* + *sampleRate* where *sampleRate* is the component's execution period.

The calculation of the next simulation date can be even more elaborated by implementing numerical differential calculus algorithms. These algorithms allow optimization of a number of simulation steps according to the evolution of attributes. If the attributes do not evolve much, it is not useful to simulate. However, if they evolve significantly, the simulation steps need to be reduced.

4.2 The Plant Model

This section describes the software architecture of the plant model. As explained in Section 2.2, a plant can be seen as a phytomers set having a determined topology. The plant's structure evolves with time (organogenesis) and each organ, through the effect of the environment, evolves individually (morphogenesis) but in close connection with the entire plant's status. The evolution of an individual organ can be reduced due to global constraints operating at the plant scale, such as water stress or carbohydrates limitation. Organogenesis and morphogenesis rules can be modified according to the plant's phenological stage.

For these reasons, the software architecture of the plant model must take into account the following elements:

- A topological structure evolving with time
- The individual evolution of organs
- The modification of individual organ evolution according to global factors (water stress, carbohydrates balance)
- The modification of the evolution rules of organs at phenological key stages.

Concerning the topology, a plant can be implemented as a tree structure in the computer science sense (Fig. 9) with a node of the tree structure representing the abstract class "Phytomer. " The composite relation in Fig. 9 depicts the structure's recursion. This relation means that a phytomer (called the *parent phytomer*) can bear other phytomers (called the *child phytomers*), which subsequently can bear other phytomers on their turn; child phytomers will constitute the axis (branch/tiller) held by the parent phytomer. A phytomer may not hold

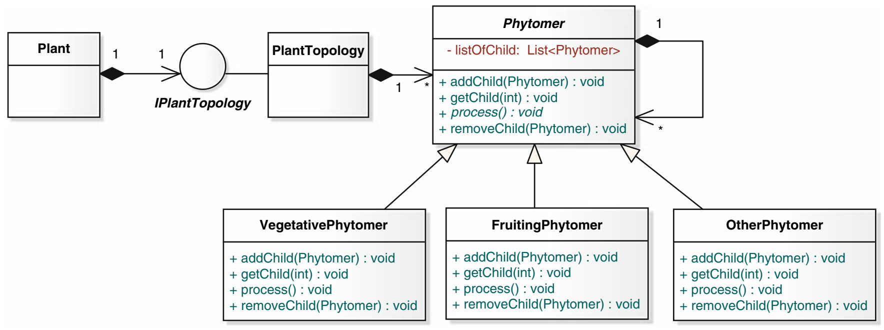
Plant
1
1
IPlantTopology
PlantTopology
1
*
Phytomer
- listOfChild: List<Phytomer>
+ addChild(Phytomer) : void
+ getChild(int) : void
+ process() : void
+ removeChild(Phytomer) : void
1
*
VegetativePhytomer
+ addChild(Phytomer) : void
+ getChild(int) : void
+ process() : void
+ removeChild(Phytomer) : void
FruitingPhytomer
+ addChild(Phytomer) : void
+ getChild(int) : void
+ process() : void
+ removeChild(Phytomer) : void
OtherPhytomer
+ addChild(Phytomer) : void
+ getChild(int) : void
+ process() : void
+ removeChild(Phytomer) : void

Fig. 9 Class diagram of the plant's topology

another phytomer (which is the case when the phytomer's bud is dormant). In this case, the phytomer can be seen as the tree structure's terminal element.

The abstract class "Phytomer" provides the methods for construction and destruction of phytomers and for the topological structure evolution. For a plant, the destruction concerns not only a phytomer and its descent but also the younger siblings. Indeed, when a phytomer is cut, both a part of the main axis constituted by sibling phytomers and the secondary axes developed from buds are destroyed. Each concrete class modeling a phytomer (e.g., Vegetative Phytomer, Fruiting Phytomer, Other Phytomer depicted in Fig. 9) inherits from the abstract class Phytomer. The inheritance link means that these concrete classes inherit the construction/destruction methods although they define their own individual growth and evolution methods.

The exact concrete object type is decided during execution according to the rules defined by the modeling agronomists. It is thus possible to model all kinds of a plant's structure, no matter the complexity. This architecture is therefore very flexible because it can be used to model different plants or varieties of the same plant with different structures.

In return, it is the computer scientist's responsibility to prevent absurd structures from being modeled (e.g., a phytomer cannot bear another phytomer if its bud is dormant). Indeed, with a tree structure, these errors cannot be checked during the compilation and are only detected during the execution.

Formal verification tools can aid in error detection. Plant construction rules can indeed be formalized as grammar in the language theory sense. Postsimulation, a formal verification tool can thus be used to check the plant's structure consistency of the grammar.

Recursion also facilitates the complex structures management. For example, applying a global plant growth method consists of using the phytomer's own growth method for each phytomer and then to invoke the growth methods of the child phytomers. The use of a tree structure can become complex when the execution order is different from the one naturally imposed by recursion. This is, for instance, the case if the execution order follows the phytomers creation order. This problem can be bypassed by using an external iterator.

At the phytomer scale, any concrete phytomer is constituted by a set of organs (node, internode, leaf, bud, flower, etc.). Phytomers' content is modeled with the UML composition concept (Fig. 10) where the phytomer class contains a reference on its constitutive organs. Both the organ type and number can vary from one phytomer to another.

Each organ individually manages its own behavior, but this behavior also depends on global processes operating at the plant's scale. For example, organ growth is a function of thermal time (timescale depending on temperature) but depends also on carbohydrates allowance. This latter factor is determined by a carbohydrate pool that calculates the global offer produced by leaves photosynthesis and then allocates the produced carbohydrates to the organs. Those global processes are naturally modeled as objects composing the "Plant" class. The question is therefore how to allow an information or service exchange between classes managing global and local processes (Fig. 11).

A solution to enable global–local relationships in a tree structure is the use of the "visitor" design pattern [37]. By definition, this design pattern adds new functionalities to a composite objects set without modifying the structure itself.

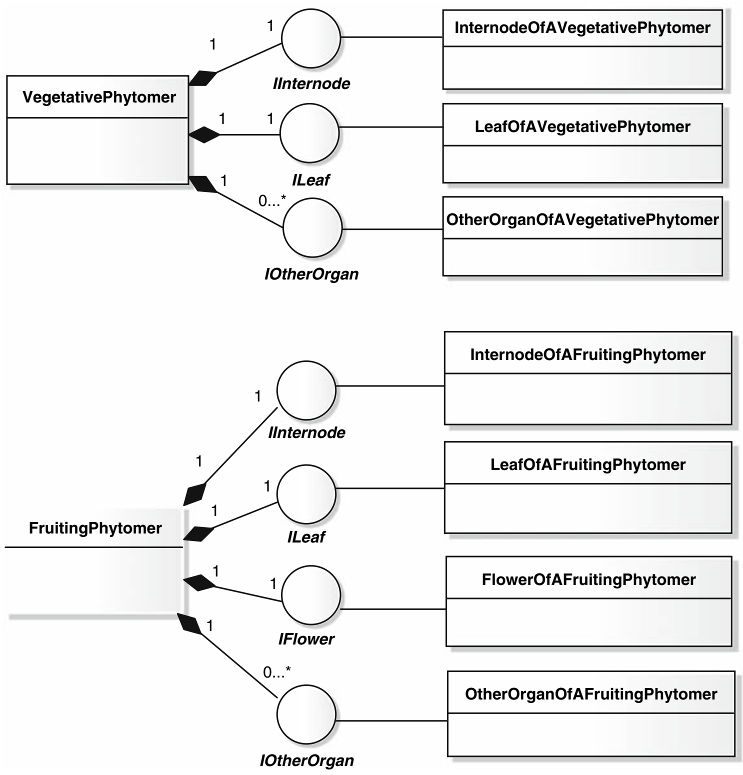
VegetativePhytomer
1
1
IInternode
InternodeOfAVegetativePhytomer
1
1
ILeaf
LeafOfAVegetativePhytomer
1
0...*
IOtherOrgan
OtherOrganOfAVegetativePhytomer
FruitingPhytomer
1
1
IInternode
InternodeOfAFruitingPhytomer
1
1
ILeaf
LeafOfAFruitingPhytomer
1
1
IFlower
FlowerOfAFruitingPhytomer
1
0...*
IOtherOrgan
OtherOrganOfAFruitingPhytomer

Fig. 10 Two examples of a phytomer's composition

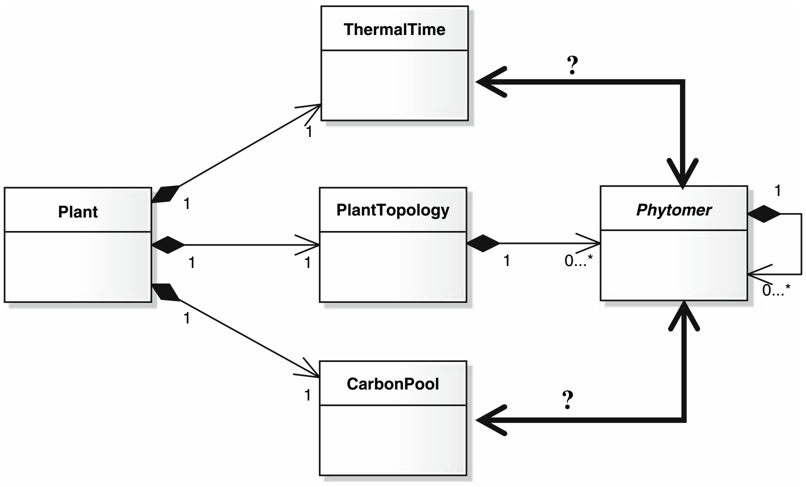

Fig. 11 Each phytomer individually manages its own behavior, but this behavior also depends on global processes operating at the plant's scale. The problem is here: how to establish relationships between local and global processes?

Nevertheless, this solution has not been chosen because it breaks the encapsulation principle. The chosen solution consists of passing a reference to the global object into an argument of organs processes' methods. Therefore, each organ manages its processes taking into account the global object (it may also modify this object's attributes state) and transmits it to its child.

As explained in Section 2.2, the plant passes through different phenological stages and its behavior as well as that of its organs can change radically from one stage to another. The evolution from a stage to the following is generally modeled in an abrupt way as a response to a discrete event (for instance, if a defined thermal time sum reaches a threshold). Consequently, a state diagram is well adapted to model the plant's phenology (Fig. 12). A state diagram enables the modeling of the different states of a system and the events ruling the passing from one state to another. For the plant, the state diagram describes:

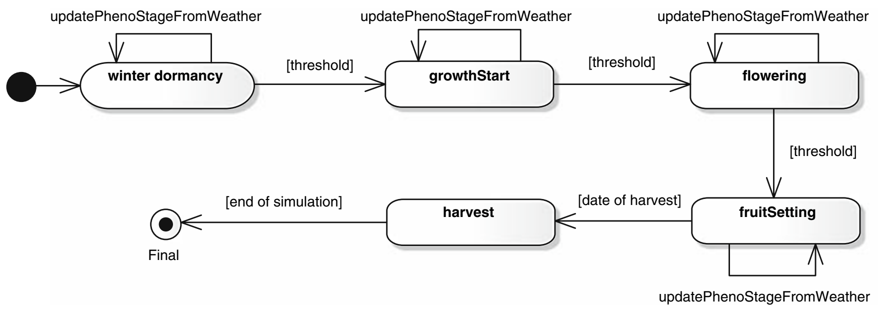

Fig. 12 Example of a state diagram concerning the plant's phenological stages

- The different phenological stages including winter dormancy, growth start, flowering, fruit setting, and harvest. These stages correspond with the plant states (in the UML sense of state).
- The performed actions during a particular stage; for instance, the update of the thermal time sum.
- The transitions between the states: A transition specifies the event imposing the state's change and defines what will be the next state. In the case of phenological stages, these events are primarily a threshold overshoot or dates chosen by the farmer (e.g., harvest date).

In the plant model, management of phenological stages is delegated to a class called "PhenoStageManager" implementing a "state" design pattern [37] (Fig. 13). This design pattern has been conceived to systematize the implementation of a state diagram from its schematic representation. In a "state" design pattern, each phenological stage implements a common interface "IPheno-Stage." At any simulation time, the "PhenoStageManager" knows the current phenological stage by the "currentStage" attribute and can ask for the update of this phenological stage by executing the object referenced by "currentStage." If this execution causes a state change, the manager is informed by the "current-Stage" attribute modification. As the phenological stages implement a common interface, the "PhenoStageManager" does not need to know what stage is concretely executed. This "state" design pattern is thus flexible as regards

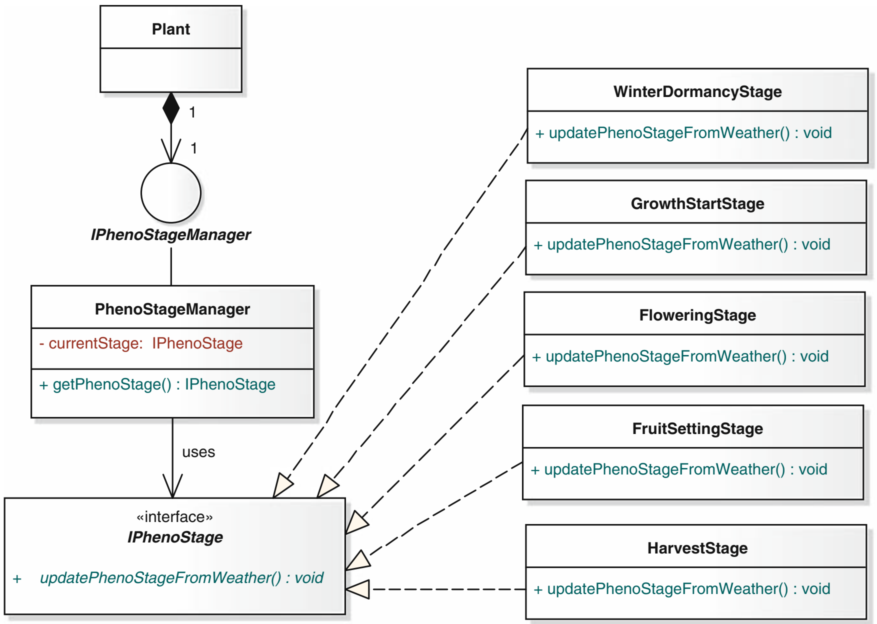
Plant
1
1
IPhenoStageManager
PhenoStageManager
- currentStage: IPhenoStage
+ getPhenoStage() : IPhenoStage
uses
«interface»
IPhenoStage
+ updatePhenoStageFromWeather() : void
WinterDormancyStage
+ updatePhenoStageFromWeather() : void
GrowthStartStage
+ updatePhenoStageFromWeather() : void
FloweringStage
+ updatePhenoStageFromWeather() : void
FruitSettingStage
+ updatePhenoStageFromWeather() : void
HarvestStage
+ updatePhenoStageFromWeather() : void

Fig. 13 Class diagram of the state design pattern concerning phenology

changes because it is independent from the phenological stages type and it allows integration of new stages and removal of others.

4.3 The Parasite Model

As explained in Section 2.3, each parasitic item follows different developmental stages. The developmental stages of parasites could be modeled like the plant's phenological stages by using a "state" design pattern. Yet this solution is not adapted to simulation of parasites because of the great number of "item" objects whose management has an important impact on simulation time and memory space.

The chosen solution models the disease as a stocks set where each stock simulates a quantity of items having the same developmental stage. This conceptual view is similar to the "flyweight" design pattern [37]. In this approach (Fig. 14), each object corresponds with a particular developmental stage, and the relationships between objects model the migration of a given number of items from one stage to another. Items migration can be seen as an assembly line where, once operated by a stage, a certain number of items migrate to the next stage.

Considering the development, stage behavior as a stock can be generalized to any other developmental stages or type of disease. Noting a developmental stage N, N − 1 the previous stage, and N + 1 the next, this behavior can be divided in three methods:

- Items migration from stage N − 1 to stage N, causing a decrease of N − 1 stage population and an increase of N stage population.
- Population increase by multiplication (for instance, in the case of mycelium increase).
- Population decrease due to mortality.

For each method, the number of items either added or subtracted depends on climate, phytosanitary protection, and plant sensitivity.

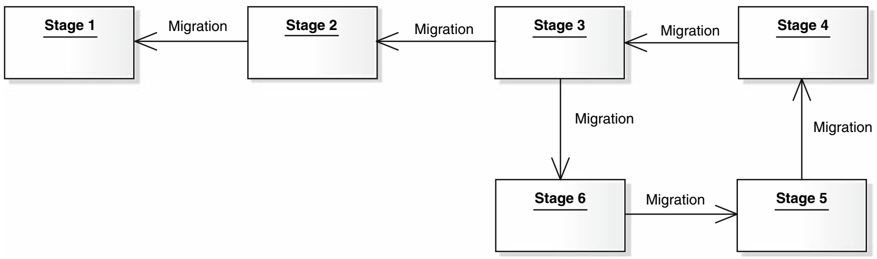

Fig. 14 Object diagram of the parasite model

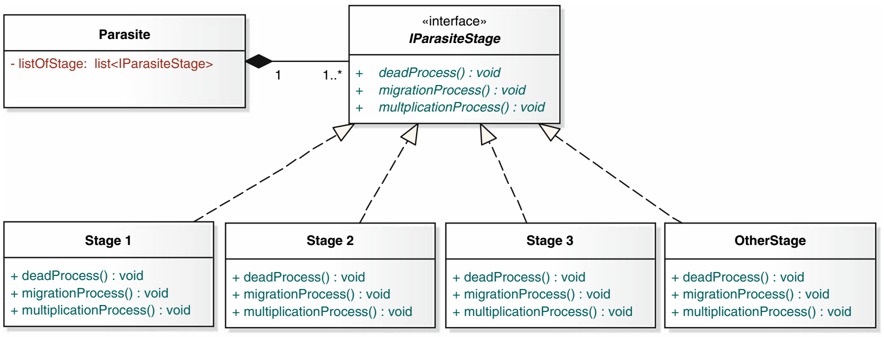
Parasite
- listOfStage: list<IParasiteStage>
1
1..*
«interface»
IParasiteStage
+ deadProcess() : void
+ migrationProcess() : void
+ multplicationProcess() : void
Stage 1
+ deadProcess() : void
+ migrationProcess() : void
+ multiplicationProcess() : void
Stage 2
+ deadProcess() : void
+ migrationProcess() : void
+ multiplicationProcess() : void
Stage 3
+ deadProcess() : void
+ migrationProcess() : void
+ multiplicationProcess() : void
OtherStage
+ deadProcess() : void
+ migrationProcess() : void
+ multiplicationProcess() : void

Fig. 15 Class diagram of the parasite model

The concrete classes modeling the developmental stages therefore share a common interface (Fig. 15). This interface has both the advantage of being generic to every developmental stage and flexible by allowing any developmental stages sequence.

5 The Application's Architecture

This section presents the application's software architecture and the different technologies used to ensure its cohesion and function.

5.1 *The Three-Tier Architecture and the Design Pattern "Strategy"*

As for many multiplatform Web developments nowadays, the three-tier architecture is chosen for the DSS software. In this kind of architecture, the generated application is divided into three separate layers:

- A "presentation" layer corresponding with what the software shows on screen
- A "business" layer where processes are performed
- A "data" or DAO (data access object) layer managing data access and storage.

The functioning of this kind of architecture is very simple, as the software user interacts with an element of the presentation layer, the latter calls the business layer to calculate the information asked by the user, and then the business layer asks the DAO layer for the data needed for the calculations. Communication fluxes then reverse. The DAO layer provides the business layer with the requested data, the business layer performs the calculations or more generally the asked service and transmits it to the presentation layer that finally displays the results. The “normal” operating mode thus induces to a round trip through the three layers. The presentation layer should never have direct access to the DAO layer and vice versa.

This organization can be easily adapted to the hardware architecture distributed on a network. Terminals on user-side present the information (corresponding with the presentation layer), whereas remote computers on server-side build this information (business layer) and manage data access (DAO layer).

The use of a software architecture based on this three-tier architecture type offers several advantages.

First of all, computer workload is dispatched. For instance, the client’s PC on the user side only supports the presentation layer, whereas servers support the business and data layers.

Another advantage is the possible use of multiplatforms. Computers on both client and server sides can run under different operating systems and interpret various programming languages, which is particularly adapted to the Web-applications context.

Moreover, the organization in layers can improve coupling quality and control between these modules. As the three-tier architecture distinguishes the presentation, the business, and the data modules, the code implementing these elements is also organized in three separate software layers. The coupling control induced by the three-tier architecture helps conserve the natural independence of the different layers.

Coupling control also facilitates work management inside the developing team. The computer specialists do not have to know all the software development aspects and can therefore be specialized in one layer (interface, business, or data), without being preoccupied by the other layers’ implementation details. Besides, dialogue between developers is more oriented on functionalities and less on purely technical problems concerning implementation. Communication between computer specialists is focused on the services to be provided by each layer and on the global system’s coherence regarding the requirements specification. Therefore, the organization and communication within the development team are improved, according to the interaction between the different software layers of the architecture.

This three-tier architecture is very close to the design pattern “strategy” as these two conception models aim at improving coupling quality between separate software agents. The design pattern strategy uses interfaces allowing one:

- to expose the services provided by software components without showing their implementation. Therefore, a component knows that another component is able to provide a certain service because this latter is exposed by its interface, but it does not know how this component is implemented to provide this service.
- to choose, when launching the application, the implementations to use for each software component by the mean of a configuration file. The whole

application is thus totally flexible. Moreover, an application can evolve by adding a new implementation in a component without constraints regarding the other components because they are all independent.

Thus, using the design pattern strategy inside a three-tier architecture provides interesting advantages [39]. The architecture is easy to maintain because, as each software module has an independent code, interleaving is minimum. This architecture is also globally reusable with respect to its components. Each component is upgradeable and specialized with little interleaving plug-in providing services. A component can thus be used in another application needing its services. Furthermore, this architecture is flexible by means of configuration files and can evolve easily, as components can be added or updated without impact on the rest of the system.

The three-tier architecture is developed with Spring,[4] which is an open source technical framework J2EE (Java 2 Platform, Enterprise Edition) simplifying implementation [40] and providing a complete and secure solution. Spring provides a lightweight container application for the objects used by the application with the Java Bean standard and enables working on simple POJOs (Plain Old Java Objects). These objects do not need to implement interfaces linked to the technical environment. Therefore, the generated code is easy to maintain and to reuse. In addition, Spring enables three-tier architecture programming in accordance with the design pattern strategy. On each layer, interfaces are used to configure the implementation of objects called JAVA beans in a totally simplified manner. This configuration includes object connection using a programming technique called "dependency injection" that will be explained later. During the application activation, objects implementation and dependencies between objects are chosen by means of Extensible Markup Language (XML) configuration files. The application is therefore flexible and upgradeable. During the development, Spring also facilitates the integration of Hibernate, a technology managing data persistence, inside the three-tier architecture.

5.2 The Three-Tier Architecture Layers and the Technologies Used

5.2.1 The Presentation Layer and Client–Server Communication

In our application, the presentation layer is relatively particular, as it is totally supported on the client side. It is designed with the Flash[5] software that also produces interactive graphical layouts facilitating interactions with potential users.

As the technologies and platforms usable for the architecture are various, Web services are chosen to enable information exchanges and more generally

[4] See http://www.springframework.org/.

[5] See http://www.adobe.com/fr/products/flash/.

client–server communications. Data flow is standardized using the XML markup language with the use of Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL), for example.

Web services have the major advantage of facilitating the building of architectures distributed on heterogeneous systems. Their conception is very close to the design pattern strategy and the three-tier architecture described above. Web services operate by exposing the JAVA beans methods of the business layer to the remote presentation layer. According to its needs, the latter layer can thus ask for services, considered as a Web service. For example, when the user interacts with the interface and asks for a plant growth simulation, the presentation layer asks to use the "SimulatePlantGrowth" method exposed by a bean of the business layer. Web services are therefore very interesting because clients of a service only need to know the URL where the Web service is exposed and also be able to read the XML WSDL file describing the technical specifications. Thus, the client can exactly know the operations supported by the Web service and the parameters needed for the operation.

The Xfire[6] environment has been chosen to expose the beans' methods of the business layer as Web services. It is an open source SOAP environment for JAVA from the CodeHaus community. It can be easily integrated to the Spring environment and allows the generation of WSDL files describing beans' available services.

5.2.2 The Business Layer and the *Dependency Injection* Design Pattern

The business layer exposes business logic–related services through interfaces. These services can be the computation of plant growth or disease development for a particular situation, for example, characterized by climate, variety, and crop practices. To access the data needed for calculation, the business layer calls methods exposed by beans belonging to the DAO layer. The business layer is thus linked with the DAO layer but, as explained in a previous paragraph, the Spring environment preserves the layers' independence.

The Spring configuration files allow the instantiation of the beans used by the business layer to provide services as a singleton. According to the singleton design pattern, the objects only exist in a single copy.

The configuration code also allows dependency injection between objects, no matter their location. This corresponds with the *dependency injection* design pattern [41], also called *inversion of control*. The two concepts of dependency injection and AOP (aspect-oriented programming) constitute the Spring environment's core. Among other things, AOP allows transaction management. These notions are illustrated with the following example, corresponding with an XML Spring configuration file extract:

[6] See http://xfire.codehaus.org.

```
<?xml version = "1.0" encoding = "UTF-8"? >
<beans xmlns: = "http://www.springframework.org/schema/beans"
          xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation = "
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd" " >

      <bean id = "simulation"
                class = "com.services.simulation.impl.SimulationMP1ServiceImpl" >
<property name = "plantDAO" ref = "plantDAO" / >
                <property name = "searchService" ref = "searchService" / >
      </bean >

      <bean id = "searchService"
                class = "com.services.search.impl.SearchServiceImpl" >
                 <property name = "plantDAO" ref = " plantDAO " / >
       </bean >

     [. . .]
```

The above code shows a bean in charge of the service called “simulation” (<bean id = "simulation"), logically located in the business layer. This bean has a reference to a bean located in the DAO layer (ref = "plantDAO") and another one referred to a bean located in its own layer (ref = "searchService"). The class implementing the bean "simulation" is class = "[. . .] SimulationMP1Service Impl">. This class has attributes referencing (meaning allowing access to) "plantDAO" and "searchService" also instantiated by Spring. To provide the object simulation with these two references, a dependency injection is needed. These dependencies are simply defined as properties (<property name = "plant-DAO") of the simulation bean "simulation." To have access to these properties, meaning injecting references to the objects "plantDAO" and "searchService," Spring uses getters/setters of the "simulation" object. Spring is also able to generate all the declared dependencies between the application objects. In concrete terms, the dependencies do not have to be hard coded any more and are externalized in easy-to-modify XML files. Therefore, to modify interfaces' implementation and dependencies between objects, it is not necessary to modify and recompile the classes' code.

5.2.3 The DAO Layer and Hibernate

Like the business layer, the DAO layer contains singleton beans instantiated by Spring. In this layer, an interface is defined for each object, such as PlantDAO, LeafDAO, ObservationDAO, and for each interface; N implementations are available such as PlantDAOImplA, PlantDAOImplB, LeafDAOImplA, Leaf-DAOImplB, LeafDAOImplC, and so forth.

In the architecture, the DAO objects aim at accessing a relational database management system (RDBMS), but they could access any type of system providing or storing data. This layer is decoupled from the technology used for data persistence, so that architecture is more generic and upgradeable. Persistence of JAVA/J2EE objects to a relational database is in charge of the Hibernate[7] environment. Indeed, this environment facilitates separation, or, more precisely, improves its quality [42, 43]. Spring is planned to integrate Hibernate and offers native classes that assist this integration. XML files called *mapping files* are used to establish a correspondence between JAVA beans and database. The link between objects and their backup in the database is called *object relational mapping* (ORM). A class, for example "Plant" in the object sense, corresponds with a table of the database in the relational sense. Each attribute of this class such as size or age corresponds with a column of this table. Each object is individually identified with an attribute "id" called the identifier, also recorded in a column of the table. The use of Spring and Hibernate allows the achievement of operations such as CRUD (Create, Read, Update, Delete) in a very simple way inside the database, although more complex queries can also be built. Regarding optimization, it is important to write queries that will exactly answer to the business layer objects' expectations that are usually accurate. These queries can be decoupled from the DAO classes' code. It is possible to store them in mapping files, which favors maintenance operations.

6 Conclusions

This chapter reviews the different design phases of a DSS intended for plant protection. The final product presented here is a Web software aiming at a large distribution among farmers. The creation of this software was shown to be complex and involved a multidisciplinary team. The first work consisted of identifying the different tasks in accordance with the requirement specifications. Four main tasks appeared: the design of the scientific model, the processable model, the interface, and the software architecture, respectively. As the specialists involved in this project belong to different professional domains, particular attention has been paid to the aspect of communication. The latter has been made simpler by adapted modeling tools such as UML. Concerning the scientific model, a major characteristic is the use of mechanistic models to simulate the biological system. This system is composed of modules, corresponding with the major actors of the system, evolving in interaction with each other. The behavior of each module, such as the plant, is precisely described in relation with its environment by means of equations. This aspect constitutes the basis of the

[7] See http://www.hibernate.org/.

tool relevance because the recommendations given by the model are scientifically based and reflect the system's behavior in the specified environmental conditions. Concerning the processable model and the software architecture, particular attention was paid to flexibility and evolutionary capacity so that applications can be easily upgraded or adapted to other systems.

This work is the beginning of an important agronomic modeling project. Even if the example taken concerns only the phytosanitary plant protection, the creation process is generic and can be applied to other DSSs. Moreover, the acquired experience will allow a quick design and set up of future DSSs. The potential use of this DSS is large. Indeed, new models concerning other pests and other phytosanitary products can be easily introduced in the current tool as new modules or new functionalities, for example. The integration of other disease models concerning the same crop opens new work prospects for the DSS improvement. This will indeed enable a more accurate assessment of crop damage and subsequently of phytonsanitary protection profitability. Moreover, it will be possible to take into account the competition between diseases. Furthermore, these tools also provide a pedagogical purpose aiming at good farming practices. For instance, concerning phytosanitary protection, the farmer can adapt his application program according to simple risk indicators that are easily understandable but are still scientifically based.

In the current context, environmental protection is an absolute priority, and concerning agriculture, it is essential to aim at more environment-friendly farming practices in the short-term future. This change is urgent because agriculture is going to be at the center of major stakes such as global human population growth and massive biofuel use, requiring huge production. All these conjunctural factors show that a rapid change in farming practices is necessary. The software presented in this chapter is designed to help this evolution. Indeed, the developed DSS allows a better diagnosis and short-term predictions of the current situation, and, furthermore, it can forecast the consequences of different scenarios tested by means of simulations. This system can also easily integrate new data or constraints to propose recommendations, always with a scientific justification. An increased use of these kinds of tools can allow a significant reduction of chemical products in agriculture without detriment to yield or production quality.

References

1. Rouzet J., Pueyo C., "Modèles de prévision et conseil phytosanitaire. Bilan des modèles en France, aperçu américain et perspectives";, Phytoma, 591: 32–36, 2006.
2. Decoin M., "OAD vus par la SdQPV, du côté des modèles", Phytoma, 603: 24–25, 2007.
3. Munier-Jolain N.M., Savois V., Kubiak P., Maillet-Mezeray, J. Jouy L., Quere L., "Decid'Herb : un logiciel d'aide au choix d'une méthode de lutte contre les mauvaises herbes pour une agriculture respectueuse de l'environnement", Proceeding AFPP – 19ième conférence du Columa – journées internationales sur la lutte contre les mauvaises herbes, Dijon, Décembre 2004.
4. http://www.invivo-group.com.

5. Booch G., Rumbaugh J., Jacobson I., “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.
6. Jones C.A., Kiniry J.R., “;CERES-maize, a Simulation Model of Maize Growth and Development”, A&M University Press, 1986.
7. Brisson N., Mary B., Ripoche D., Jeuffroy M.H., Ruget F., Nicoullaud B., Gate P., De-vienne-Barret F., Antonioletti R., Durr C., Richard G., Beaudoin N., Recous S., Tayot X., Plenet D., Cellier P., Machet J.M., Meynard J.M., Delecolle R., “STICS: a Generic Model for the Simulation of Crops and Their Water and Nitrogen Balances. I. Theory and Parame-terization Applied to Wheat and Corn”, Agronomie, 18: 311–346, 1998.
8. Luquet D., Dingkuhn M., Kim H.K., Tambour L., Clément-Vidal A., “ EcoMeristem, a Model of Morphogenesis and Competition Among Sinks in Rice : 1. Concept, Validation and Sensitivity Analysis ”, Functional Plant Biology, 33(4): 309–323, 2006.
9. Louarn G., “Analyse et modélisation de l’organogenèse et de l’architecture du Rameau de la vigne (Vitis vinifera L.)”, thesis PhD, école nationale supérieure agronomique de Montpellier, 2005.
10. Jallas E., Martin P., Sequeira R., Turner S., Crétenet M., Gérardeaux E., “Virtual COTONS®, the Firstborn of the Next Generation of Simulation Model”, NLAI 1834, pp. 235–245, Springer, 2000.
11. Costes E., Guedon T., “Modelling the Sylleptic Branching on One-Year-Old Trunks of Apple Cultivars”, Journal of the American Society of Horticultural Science, 122: 53–62, 1997.
12. Seleznyova A.N., Thorp T.G., Barnett A.M., Costes E., “Quantitative Analysis of Shoot Development and Branching Patterns in Actinidia”, Annals of Botany, 89: 471–482, 2002.
13. Buhlmann P., Wyner A.J., “Variable Length Markov Chains”, The Annals of Statistics, 27: 480–513, 1999.
14. Nozeran R., “Réflexions sur les enchaînements de fonctionnement au cours du cycle des végétaux supérieurs”, Bull. Soc. Bot. Fr., 125: 263–280, 1978.
15. Vanderplank J.E., “Plant Diseases: Epidemics and Control”, Academic Press, 1963.
16. Fleming R.A., “Development of a simple mechanistic model of cereal rust progress.“ Phytopathology, 73(2): 308–312, 1983.
17. Keeling M.J., Rohani P., “Estimating spatial coupling in epidemiological systems: a mechanistic approach”, Ecology Letters, 5(1): 20–29, 2002.
18. Passioura J.B., “Simulation models: snake oil, education, or engineering? ” Agronomy Journal, 88: 690–694, 1996.
19. http://www.hps-inc.com/edu/stella/stella.htm.
20. Makowski D., Hillier J., Wallach D., Andrieu B., Jeuffroy M.H., “Parameter estimation for crop models” in: “Working with Dynamic Crop Models – Evaluation, Analysis, Parameterization, and Applications”, pp. 101–140, Wallach D., Makowski D., and Jones J.W. (eds), Elsevier, 2006.
21. Saltelli A., Chan K., Scott E.M., “Sensitivity Analysis”. Wiley, 2000.
22. Saltelli A., Tarantola S., Campolongo F., Ratto F., “Sensitivity Analysis in Practice”, Wiley, 2004.
23. Wallach D., Makowski D., and Jones J.W, “Working with Dynamic Crop Models – Evaluation, Analysis, Parameterization, and Applications”, Wallach D., Makowski D., and Jones J.W. (eds), Elsevier, 2006.
24. Monod H., Naud C., Makowski D., “Uncertainty and sensitivity analysis for crop models”, in: “Working with Dynamic Crop Models – Evaluation, Analysis, Parameterization, and Applications”, pp. 55–96, Wallach D., Makowski D., and Jones J.W. (eds), Elsevier, 2006.
25. Wallach D., “Evaluating crop models” in: “Working with Dynamic Crop Models – Evaluation, Analysis, Parameterization, and Applications”, pp. 11–50, Wallach D., Makowski D., and Jones J.W. (eds), Elsevier, 2006.
26. Delannoy C., “Programmer en langage C”, Eyrolles, 1997.

27. Delannoy C., “Programmer en langage C ++ ”, Eyrolles, 1998.
28. Niemeyer P., Knudsen J., “Introduction à JAVA“, O’Reilly, 2002.
29. Martin M., “Programmeur Delphi 2005”, CampusPress, 2005.
30. http://www.mathworks.com.
31. http://www.r-project.org.
32. http://www.scilab.org.
33. http://www.wolfram.com.
34. http://www.maplesoft.com.
35. http://www.modelmakertools.com.
36. Cier P., Dorin R., “Excel 2000 en pratique”, Dunod, 1999.
37. Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.
38. Porter C.H., Braga R., Jones J.W., “An Approach for Modular Crop Model Development”, Agricultural and Biological Engineering Department, University of Florida, Gainesville, Florida, available at http://www.icasa.net/modular/downloads.html, 1999.
39. Papajorgji P., Pardalos P., “Software Engineering Techniques Applied to Agricultural Systems, an Object-Oriented and UML Approach”, Springer, 2005.
40. Johnson R., Hoeller J. “Expert One-on-One J2EE Development without EJB ”, Wiley Publishing, 2004.
41. Fowler M.,“Inversion of Control Containers and the Dependency Injection pattern”, http://martinfowler.com/articles/injection.html, 2004.
42. King G., Bauer C., “Hibernate in action”, Manning Publications, 2004.
43. Salvatori O., Patricio A, “Hibernate 3.0”, Eyrolles, 2005.