Object-oriented approach to crop modeling: Concepts and Examples

P. Papajorgji, R. Braga, Ch. Porter, J.W. Jones, H.W. Beck

Abstract

This research was initiated to learn what the crop modeling community could do to facilitate an evolution from existing crop models toward more modular ones that can meet increasing demands. Specific objectives are to 1) use UML to design and implement a modular crop model using an object oriented language, and 2) to compare the characteristics of this model and its development with a modular model written in FORTRAN. To better understand issues and challenges of using an object oriented (OO) paradigm relative to a Fortran-based model, we investigated the process with a simple, generic crop model. The complexity of the model is minimal hence allowing the focus to be more on model structure (the real issue) instead of the model itself. Two critical issues were considered throughout this study: 1) the exchangeability of modules and 2) the independence of the structure from a specific programming language.

From our experience of converting a simple crop model developed in FORTRAN into an object-oriented environment we found that UML can be useful to bring modelers to discuss modeling issues without considering any particular programming language. Modelers can exchange diagrams and convert them into particular programming languages.

Keywords: Object-Oriented, Crop model, UML, FORTRAN.

Introduction

In the late 1960’s and early 1970’s, researchers started developing computer models for simulating crop growth and yield (Duncan, 1972; Hesketh et al., 1972; Stapleton et al., 1973; Jones et al., 1974). Following the introduction of the personal computer, the number of

1 University of Florida, PO BOX Gainesville, Fl 32611
researchers developing models increased, as did the number of crop models. Researchers began to realize the need to make models compatible for more efficient applications by researchers and others outside the groups that developed them.

There have been different responses by researchers to these critical needs. One could argue that the models should be reprogrammed, however unless there is a compelling reason and resources to do so, this is not a practical solution. Some researchers have reprogrammed existing models using new object-oriented programming languages, but these efforts have mostly not achieved acceptance. Although the new languages have many advantages relative to software development, these efforts have not, by themselves, led to a move to convert existing models from FORTRAN to these new languages. Specifically, the exchange of data and expertise between researchers in different regions of the world does not yet seem to benefit from the advantages of OO paradigm.

**Modular Model Structure in a Procedural Language**

To discover the nature of problems we may face while converting our crop simulation models written in Fortran into an OO environment, we started with a small generic model (Porter et al. 1999). This model is available at [www.icasanet.org/modular](http://www.icasanet.org/modular). The model contains three main modules: Soil, Plant and Weather. Researchers at the Wageningen Agricultural University developed and extensively use this approach Kraalingen (1995). The basic idea is to create units or modules that would act as individuals, with their own data and subroutines. Each module should (Jones et al., 2000): read its own parameters, initialize its own variables, own its set of state of variables, compute rates of change for its state variables, integrate its state variables and be able to dialog with other modules in order to simulate the entire system.

Although this modular crop model was written in Fortran, conceptually it was a logical step towards creating independent units with data and behavior. Our OO approach largely used this modular structure to depict classes and provide them with the right data and behavior.
Design of Object Oriented Software using UML

To effectively build a complex system, the developers begin by looking at the big picture without getting caught up in details. Visual modeling has one communication standard: Unified Modeling Language (UML) (Booch et al. 1999). The UML provides a smooth transition between the business domain and the computer domain. The following are the UML diagrams used to build the system.

Use case diagram
A use case shows that the user should send to the system the message **simulate** and the system should provide the user with results of the simulation. Simulator is an object that controls the dialog of the system with the external environment. Therefore, this object should be provided with controlling or supervising behavior. The only important information that this diagram should present is that Simulator understands the message **simulate** and provides the user with simulation results. Figure 1 shows the use case diagram.

Class diagram
From a software engineering perspective, the first thing to do in developing software is to depict the participants (or entities) in a domain that will dialog between them to carry out some function. In our crop-modeling scenario, at the highest level of abstraction, we have depicted the following participants: **Plant**, **Soil** and **Weather**.

After depicting the participants, it is very important to agree on what the dialog should be between them. The dialog between the participants in our example crop model is shown in Figure 2.
Object **Plant** and object **Soil** are linked by an association that shows that **Plant** is planted in **Soil**. Both **Plant** and **Soil** have a well-defined role in this dialog. **Plant** should provide leaf area index data and **Soil** should provide water stress related data. The dialog between objects **Plant** and **Weather** is represented by another association that shows that **Weather** affects plant processes. In this association, the role of **Weather** is to provide weather data and the role of object **Plant** is to receive weather data. The dialog between object **Soil** and object **Weather** is represented by an association that shows that **Weather** affects soil processes and the role of object **Weather** is to provide weather data to the **Soil** object. The role of object **Soil** is to receive weather data. Special attention should be paid to establish the dialog between participants, as this is the foundation for discerning the behavior of future classes of the system.
Sequence diagram
The sequence diagram shows the messages that participants send to each other in order to achieve the desired results. In addition to Soil, Plant and Weather participants, we have introduced two other participants, the User and the Simulator as shown in Figure 3.

Figure 3. Sequence diagram.

The Simulator participant has a special role in the system. It is used to coordinate the messages that participants send to each other in a timely manner. Note that it follows the approach used by Kraalingen (1995) in initializing each module, then computing rates, integrating, and writing outputs from each module.

The user starts the simulation process by sending the simulate message to the Simulator object. From this point, the Simulator will
take control. It will start sending messages to other objects, obtain the simulation results and provide them to the User.

**Implementation**

In other Object-Oriented approaches to crop modeling that we have seen, authors use concrete classes such as **Soil**, **Plant** or **Weather**.

These classes are provided with attributes and methods so that they will have the required behavior. Code is written to describe the dialog between classes in a particular programming language. Classes are directly involved in this dialog; they send messages to each other to achieve results. Therefore classes have a high level of coupling. Coupling describes how much interaction two or more classes or components have. Coupling makes the system rigid and difficult to change. A change in one class easily can propagate errors in other classes or parts of the system. Furthermore, the classes are implemented in such a way that it is difficult for another researcher to use them without totally accepting the methodology used by the original author.

1. Use Interfaces instead of classes.

Instead of coupling classes in a rigid way, another concept, **interfaces**, should be used to express the general dialog between participants.

An interface is a contract of related services and a set of conditions that must be true for the contract to be executed. They can be considered as “pure” abstract classes that allow the user to establish the form for a class: method names, argument lists, and return types, but no method bodies. In other words, an interface defines how all classes that implement this particular interface will look. The user that decides to implement this interface will implement the method body. Interfaces are used to establish a “protocol” between classes.

**SoilClass** (Figure 3) does not offer its services to other classes directly. An interface **SoilIF** stands between **SoilClass** and other classes that need to dialog with it. The role of **SoilIF** is to define the
services that **SoilClass** offers but not their implementation. It is up to **SoilClass** to implement these services. **PlantClass** needs to use **SoilIF** interface to get access to **SoilClass**. The advantage of using interfaces is that the implementation of the method **getSoilWaterStress** in the class **SoilClass**, is irrelevant for **PlantClass**. Therefore changes of the implementation of this method will not affect **PlantClass** or other classes that use the result of this method. At this stage, it is irrelevant who is going to implement these services and how. Crop modelers only have to agree on what the dialog between participants should be. Although agreeing on what this dialog should be may be very difficult, it does provide a pathway for module independence and interchangeability.

![Figure 3. Dialog between classes through interfaces.](image-url)

### 2. Separate interface form implementation

At this level of analysis, it is premature to decide whether a **SimplePlant** class or a **ComplexPlant** class is needed. A **SimplePlant** class would have only attributes, be they attributes that belong to the stem, leaves or roots or to the plant itself. A **ComplexPlant** class may have two kinds of attributes: simple attributes such as integers (for plant age, for example) and complex
attributes that could be classes. For example a ComplexPlant class may be defined as a composition of Root, Stem, Leaf that are classes on their own and simple attributes such as age, cultivar etc. What is important is the fact that whatever the structure of the class Plant (simple or complex), it has to know how to respond to the message getLeafAreaIndex defined by the interface that both classes implement. Examples of SimplePlant and ComplexPlant classes are shown in Figure 4.

Figure 4. Different class implementations interacting with the same interface.

Both implementations of the same class (SimplePlant and ComplexPlant) can be used interchangeably as long as they implement the same interface PlantIF. Separating interface from implementation makes the system flexible and gives developers freedom to implement classes based on their particular architectural needs.
3. Reverse engineering

Another important feature provided by UML tools is the capability to generate code from diagrams and vice versa. This allows modelers to have a common language to discuss modeling issues without having to refer to a particular programming language. After building diagrams, modelers can implement their models using different programming languages. UML tools can convert diagrams into class skeletons in different programming languages. Examples of converting the same diagram, the **ComplexPlant** class, into code in two different languages: Java and C++ are shown in Figure 5 and 6 respectively. TogetherSoft (TogetherSoft, Inc. 2001) modeling tool is used to produce class skeletons through the reverse engineering process.

```java
/* Generated by Together */

public class ComplexPlant {
    public void getLeafAreaIndex() {
        Leaf getLeafAreaIndex();
    }

    /**
     * @link aggregationByValue
     * @clientCardinality 1
     * @supplierCardinality 0..*
     */
    private Leaf lnkLeaf;

    /**
     * @link aggregationByValue
     * @clientCardinality 1
     * @supplierCardinality 1
     */
    private Stem lnkStem;

    /**
     * @link aggregationByValue
     * @clientCardinality 1
     * @supplierCardinality 1
     */
    private Root lnkRoot;
}
```

**Figure 5.** Java code generated from ComplexPlant class diagram.
### 4. Object-Orientation and the Fortran Legacy

While the object design approach has many advantages, many existing crop models are already written in FORTRAN. Rather than rewrite the existing FORTRAN code using an object oriented language such as C++ or Java, it may be desirable to keep the existing FORTRAN code but move it into an environment where object oriented design principles can still be used. One way to do this is through the use of wrapper objects. Wrapper objects appear from the outside to be standard objects (with usual features such as callable methods), and behave just like any other object. But internally, the methods can be implemented using existing legacy programs, such as crop models written in FORTRAN.

---

```cpp
/* Generated by Together */

#ifndef COMPLEXPLANT_H
#define COMPLEXPLANT_H  

class Root;
class Leaf;
class Stem;

class ComplexPlant {
    private:

        /**
         * @link aggregationByValue
         */
        Stem * lnkStem;

        /**
         * @link aggregationByValue
         */
        Leaf * lnkLeaf;

        /**
         * @link aggregationByValue
         */
        Root * lnkRoot;
};
#endif //COMPLEXPLANT_H
```

---

**Figure 6.** C++ code generated from ComplexPlant class diagram.
Thus to implement an existing FORTRAN module inside a wrapper object, the interface for the module would need to define the methods that would be called in this module. A class would be created to implement this interface, and inside this class would be calls to FORTRAN programs. Values returned by these calls would be stored in member variables defined in the class.

The first thing to do while creating a wrapper object representing a FORTRAN component, is to create an interface. Interface’s methods define all functionalities provided by the FORTRAN component. The actual implementation of the methods will be done by the classes implementing this interface. As an example, the following is an interface written in Java for the modules in CropModule:

```java
interface CropModule
{
    void initialization();
    void rateCalculations();
    void integration();
    void output();
    void close();
}
```

Every module (Plant Growth, Soil/Water, etc.) in the crop model implements these 5 basic behaviors. So, for example, a Plant Growth module would implement this interface via a Java class definition such as:

```java
class PlantGrowth implements CropModule
{
    float height;
    float weight;

    native void initialization();
    native void rateCalculations();
    native void integration();
    native void output();
    native void close();

    double getHeight(){return height;}
}```
double getWeight(){
    return weight;
}

In addition to implementing the 5 basic behaviors, additional methods specific to PlantGrowth class are added. For example, current plant height and weight are stored in member variables, and can be accessed by other objects by calls to getHeight() and getWeight() respectively.

The key to wrapping an existing FORTRAN module inside this class comes from the “native” method implementations. The keyword “native” indicates that the actual code for these methods is not written in Java, but in a language native to the operating platform, such as FORTRAN. The technical details of calling FORTRAN from Java involve using the Java Native Interface (JNI). In fact, JNI cannot call FORTRAN directly, but it can call C programs, and the C program can call FORTRAN. Unfortunately this necessitates a intermediate C program. But the C routine simply passes information between FORTRAN and Java, so the performance is not significantly impacted. The compiled FORTRAN code is contained within a dynamic link library (DLL), that is called from C. These member variables in class PlantGrowth are set internally though interaction with the FORTRAN program.

Thus the existing Plant Growth module written in FORTRAN remains entirely unchanged, yet an environment is created where the Plant Growth module behaves as an object. Several advantages in redesigning CropModule are obtained by this approach:

- CropModule is cast into an object-oriented framework without having to rewrite the FORTRAN code.
- CropModule is better modularized by using objects to describe each module.
- CropModule modules now have formal methods for the 6 main behaviors (the FORTRAN implementation used a control variable and “if” statements to implement this behavior because the FORTRAN language does not support methods).
- Intramodule communication is much cleaner. Data values are passed directly between modules by using method calls. The original crop model FORTRAN required a complex “Main
Program” to handle all intramodule communication and manage memory.

Discussion

From our experience of converting a simple crop model written in FORTRAN into an object-oriented environment we can recommend the following:

- Switching a programming paradigm should be addressed carefully by advancing by reasonable steps.
- Interfaces are useful to define generic behavior for classes and to apply the principle of separating interfaces from implementation.
- UML can be useful to bring modelers to discuss modeling issues without considering any particular programming language.
- Reverse Engineering can speed up the process of creating code in different programming languages from common diagrams.
- Modelers can exchange diagrams and convert them into their particular programming language.
- Wrappers are useful tools to bridge the very valuable expertise accumulated in legacy systems to object-oriented paradigm.

References


