Use of Software Components in Model Development

C.R. Maul

DNRE Tatura, Ferguson Rd, Tatura VIC 3616, Australia (Christian.Maul@nre.vic.gov.au)

Abstract: Software components do not stand alone but are designed to work as a indivisible part of a larger application. They open new horizons for modelling as they offer stability, reusability and adaptability. They can help to fill in gaps in modelling, to create shortcuts where one's own expertise or resources are lacking and they allow for larger and more comprehensive models. Model or functional components are of great interest to modellers. These could be mathematical, statistical, database and data manipulation components or ones that deal with general hydrological problems. To make component development work they must be useful and technical standards must be used, implemented and promoted. Even more important than issues of standardisation are issues of sound software engineering practices. Only when properly implemented do components redeem the promise of reusability of code and only well-designed components can speed up the development process considerably. Currently only a few languages such as Visual Basic, Delphi, Java, Eiffel and Oberon provide any technical component standards. In this paper, issues associated with component development, standards and engineering practice are described and discussed using the meteorology component of "FruitSim", a plant growth model, that I wrote in Java.

Keywords: Modelling; Climate; Software engineering; Java; Software components

1. INTRODUCTION

Software components merge two distinct perspectives: components as implementation of software and components as architectural abstractions. In the same way that builders use standardised door frames, beams or other parts from third parties to build a house, software architects look for recurring patterns, functions or processes that can be converted into building blocks for an software application. A component is:

- an opaque implementation of functionality that makes sense only in a larger application,
- subject to third-party composition, and
- conforms to a component model [Bachman et al. 2000].

The first components were GUI (graphical user interface) components. For instance, Java's user interface, the Java Foundation classes, are realised as components. Components can be bought from other companies such as IBM alphaworks (http://alphaworks.ibm.com/), KL-Group (http://www.klgroup.com/), Tidestone (http://www.actuate.com/) or Roguewave (http://www.stingray.com/). The industry standard ensures that they all work together satisfactorily. The typical GUI is a mixture of components produced by different companies. Second generation components contain both GUI and functional capabilities such as Formula One from Tidestone. It is a spreadsheet component with a visual part for reporting and charting purposes and a non-visual part that calculates, connects to databases, and reads and writes Excel file formats.

For modelling purposes in particular, non-visual components are more interesting. They can be mathematical components that realise neural networks, Bayesian belief systems, matrix algebra or a simple solution to linear equations. The next interesting class of non-visual components is that which provides database access or network services such as file transfer protocol (FTP), email or messaging services.

In addition to general purpose components such as those mentioned above, components that perform specific recurring tasks in hydrological models are

possible, provided they conform to an agreed interface. Components and interfaces enable a division in labour in modelling for the first time. To demonstrate the strengths, pitfalls and ramifications of design decisions on component development, the meteorology component of "FruitSim" is used. Using existing program parts for component development is the best way to design components because use precedes reuse.

2. METHODS

2.1 Assessment Criteria

Criteria against which components should be assessed are:

1. the ability to configure the properties of the component, such as simple and bounded properties, constrained and index properties;

2. the ability to react to events;

3. sufficient documentation on three levels

   a) introspection for use with the IDE (Integrated Development Environment), which means that the IDE actively looks into the component and displays all the available methods and their correct use

   b) component usage description for the user - developer such as BeanInfo in Javabeans (see http://java.sun.com/j2se/1.3/docs/api/index.html) and

   c) detailed documentation of the component interface such as the HTML documentation generated by the JavaDoc tool;

4. safe use of the component including error handling and instantiation;

5. customiser for correct use and instantiation of components;

6. persistence of the component or objects of the component; and

7. pre- and post-conditions for component use;

8. packaging standards.

These criteria are based on Pelegri-Llopart and Cable's [1977] and my own experiences as a user and producer of components.

A component designed to provide meteorology data for FruitSim will be assessed using these eight criteria.

2.2 "FruitSim" Meteorology Component

"FruitSim" is a tree growth model that calculates yield, water use and growth dependent on climate and management variables. Climate variables drive the model in its daily calculations. Because some values needed to run the program are not always provided by the METACCESS database, the meteorology component calculates the missing information. It comprises four classes: WeatherEvents, AirPropertics, Climate and SoilProperties.

WeatherEvents is a rather dumb data container which consists of encapsulated variables (properties) with their public methods for access and modification. The only function that requires some intelligence is the manipulation of dates, that is, the conversion of date strings to Julian day, determining day of the year depending on hemisphere, that is the start of the growing season, and taking into consideration time zones.

AirProperties inherits all the methods from WeatherEvents and has some methods of its own such as calculation of day length, twilight length, hourly temperature and degree day calculation. It fires an event if the date is changed and initialises the values such as day length, declination or sun angle accordingly.

The third class, Climate, inherits from AirProperties, which means that it comprises the functions of both previous objects. It extends the functionality to calculations of radiation and the sun's position. It overrides the propertyChange() method of AirProperties and configures the previously described attributes and the radiation attributes. Events are used in two different ways: if variables within the object are changed that have other dependent variables, events are fired to notify these dependent variables and to change them accordingly. Secondly, events are also fired to the outside world. They can be picked up by any other object that has a PropertyChangeListener registered. Through this mechanism the component can be integrated into any program, because any object yet to be created simply needs to register a listener. It will then be notified immediately and can change its processing accordingly.

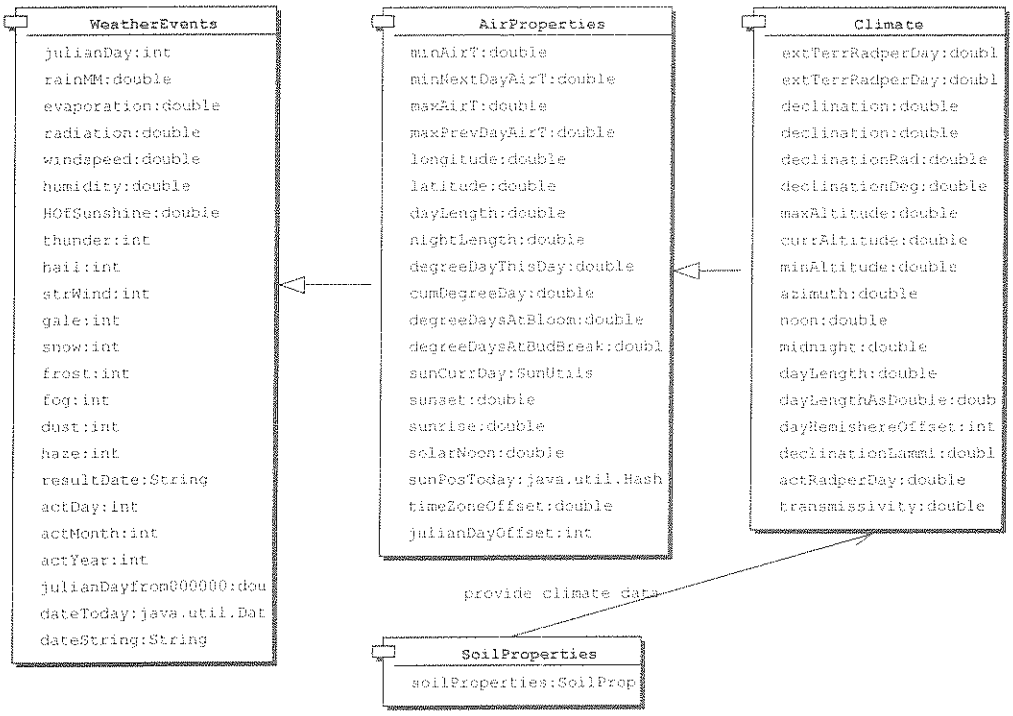

Figure 1. Class diagram of the meteorology component containing the properties of the classes.

SoilProperties calculates soil temperatures. Figure 1 shows a class diagram of the four classes and their properties. More detailed information can be found in Maul [1999].

Eighty seven properties represented by variables can be set or queried. The variables themselves are hidden either completely by defining them as private or partially by defining them as protected which prevents direct access.

A BeanInfo class provides information about the properties, their possible values and events for the IDE so that it can guide the user.

Apart from the BeanInfo class the JavaDoc documentation may be used to determine what each method does and the scientific basis of the particular algorithms.

Further support could be given by PropertyEditors which define ranges of values for variables when the component is used in an IDE. They are, however, not implemented in this example.

A customiser class which must be registered with the BeanInfo class could provide sensible values for properties. Technicalities, such as the use of a double as a parameter when it is required, could be forced upon the user by the compiler and the customiser could determine if a parameter makes sense when the parameter is dependent on another variable. Customisers are not implemented with the result that all values that are passed on to the constructor must be valid.

3. ASSESSMENT

How does the component rate against the eight assessment criteria?

There is certainly an abundance of set properties (criterion 1), the component is event driven (criterion 2 and it conforms to the Java packaging standard (criterion 8 and 3a). All public methods are documented (criterion 3c). Because the error handling is done elsewhere in the program in which it is used, there is no error handling in the component (fails criterion 4). This is definitely a weakness. The component expects to be initialised with sensible data for month, day, temperatures and so on (criterion 5).

The component implements the 'Serializable' interface, which means it can be stored (criterion 6).

Pre- and post-conditions (criterion 7) cannot be defined in Java. This is one of the major weaknesses of the language. Its first standard did contain pre- and post-conditions but it has since been abandoned by Sun [Coad and Mayfield, 1998].

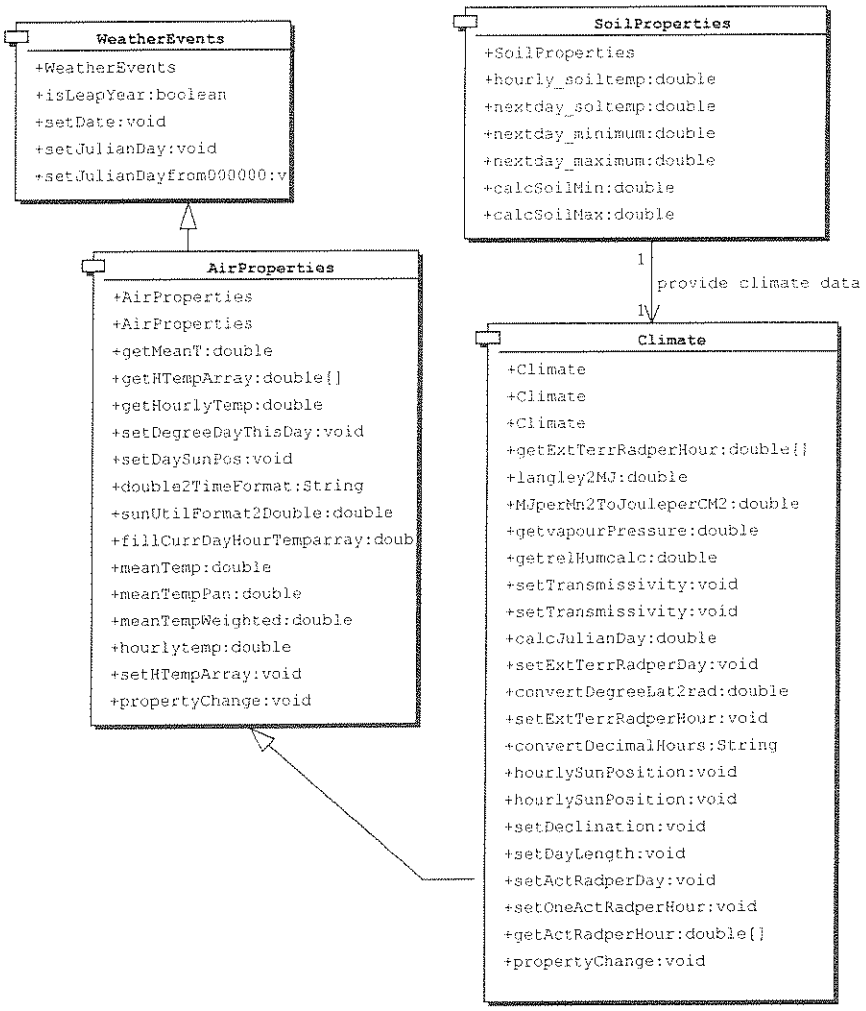

Figure 2. Class diagram of the meteorology component containing methods.

In this light the disadvantage of missing error handling has twice the impact because there are no safeguards against improper use. Pre- and postconditions could be used to enforce proper instantiation in particular.

A component can be a good component in theory but utterly useless in practice for programs. Figure 2 shows what you can actually do with the component.

It is not much if you only create a WeatherEvents object. You can convert date strings. An Airproperties object provides greater functionality such as calculating hourly temperatures from minimum and maximum temperatures, solar noon, sunset and sunrise, different kinds of twilight and daylength.

Climate extends the functionality of AirProperties as it contains many methods to calculate values of sun position at any real or solar time, extraterrestrial and ground radiation, humidity and vapour pressure. Apart from date, latitude and temperature most of the variables in the component can be set either from a database or calculated with different algorithms. Despite the naming convention of methods such as hourlytemp() the component can calculate any time resolution of radiation or temperatures because the time values are passed on as doubles. The component also provides converting and formatting methods. It is quite a powerful component that can generate a lot of values from a minimal dataset. Calculated values are certainly not as good as measured data but the JavaDoc explains the algorithms and restrictions.

The component is designed to provide values that are not provided by a particular weather station or the Australian Meteorology database, MetAccess. Its database structure, however, is similar to many other databases in the world. The component

could be used with nearly any other weather database.

Values that can be both calculated and set must be set after methods are called which fire events and calculate them automatically to prevent values being overwritten. This important order of initialisation would be a classical application for pre-conditions (criterion 7).

Three constructors provide safety and flexibility (criterion 4) for how the component is instantiated. There is a parameterless constructor, Climate(), to ensure that a climate object can be created using the Beans.instantiate() method from Java. The Java specification requires the user - developer to do so. I am, however, not convinced that this has been a good design decision because it leaves it to the user to initialise properly. If he fails to do so the program in which the component resides will not crash but it will be provided with "default" values. They will not make sense because the user wants to have data at a particular location at a particular day and time and not in Greenwich on the first of January at midnight.

4. DISCUSSION

The component described above does meet many of the assessment criteria. However, it falls down in two areas.

The first is the lack of appropriate error handling which is a significant weakness and reduces its industrial strength. Error handling is an important issue for the use of components. Many programmers have endured the painful experience of debugging function libraries or of using insufficiently documented function libraries. However, components contradict sound software engineering practices in this regard which demand a separation of the functional layer, error handling and graphical user interface.

The second is initialisation. Components are supposed to be safe, to come with their own manual and to enforce appropriate creation and initialisation. The meteorology component certainly has a manual which explains its use. However, the component itself does not prevent improper initialisation.

The design and implementation of the component have many strengths. Integration into other programs is simple because the component can be queried by IDE about its methods and properties. It is highly independent of the rest of the program because it simply requires the instantiation of a PropertyChangeListener that is registered with the component. A propertyChange() method must then query where the propertyChangeEvent is coming from and change the processing logic according to the changed climate data. The component makes good use of the flexibility provided by the Java standard.

Overriding methods can easily change the climate component. The interface that the component implements allows for the exchange of the entire component within existing programs. Extension is as easy as change because any new object can inherit from Climate and add new properties and methods.

The component standard can be used to share programming between institutions. The model is then finally assembled as a collection of black boxes that conform to a design agreed to at the start of the project.

There is a plethora of components conceivable. They can facilitate and speed up model development that clearly distinguishes tasks, separates groups or institutions and defines the responsibilities and duties of partners. Equally important as the use of component standards is proper project management. Only then can components deliver on their promises and lead to successful collaboration because they also allow for – and this is very important in science – a clear separation of merits.

5. REFERENCES

Bachman, F., F., Bass , L., Buhman, C., Comella-Dorda, S. Long, F., Robert, J., Seacord, R., K., Wallnau, Technical Concepts of Component-Based Software Engineering, Carnegie Mellon Software Institute http://www.sei.cmu.edu/publications/documents/00.reports/00tr008/00tr008title.html, 2000.

Coad, P. and M. Mayfield, Java Design. 2nd ed. Yourdon Press, Upper Saddle River, NJ, US, 1998.

Maul C.R., What would a reusable meteorology component for environmental models look like? *Environmental Software Systems*, Denzer, R. et al. (eds.), Vol. 3, pp 88–94, Kluwer, Amsterdam, NL, 1999.

Pelegri-Llopart, E. and L., P., G. Cable, How to be a Good Bean. Sun Microsystems, JavaSoft, Palo Alto, Ca, US, http://java.sun.com/products/javabeans/docs/goodbean.pdf, 1977.