Contents lists available at SciVerse ScienceDirect

Ecological Informatics

journal homepage: www.elsevier.com/locate/ecolinf

A universal simulator for ecological models

Niels Holst [*]

Dept. of Agroecology, Aarhus University, Forsøgsvej 1, 4200 Slagelse, Denmark

ARTICLE INFO

Article history:
Received 18 May 2012
Received in revised form 12 November 2012
Accepted 13 November 2012
Available online 21 November 2012

Keywords:
Software design
Simulation model
C++
XML
Agile
Object-oriented

ABSTRACT

Software design is an often neglected issue in ecological models, even though bad software design often becomes a hindrance for re-using, sharing and even grasping an ecological model. In this paper, the methodology of agile software design was applied to the domain of ecological models. Thus the principles for a universal design of ecological models were arrived at. To exemplify this design, the open-source software *Universal Simulator* was constructed using C++ and XML and is provided as a resource for inspiration.

1. Introduction

An ecological model is an instrumental summary of an ecosystem, written in the language of mathematics and logic. 'Der Satz ist ein Modell der Wirklichkeit, so wie wir sie uns denken' (Wittgenstein, 1922). However, most ecological modellers are not trained mathematicians, capable of formulating models in pure, analytical mathematics. They are rather computational modellers, who implement their models using simple mathematics implemented in software they often design themselves. This software, which can become quite complicated, then becomes both tool and object in the ensuing analysis. The design and implementation details of this software are routinely given little attention, but the focus of this paper is just this: the proper design and implementation of ecological models.

Ecological models seem to call for a design of naturally given building blocks, however diverse, which are suitably expressed in an object-oriented design. An object encapsulates a state and has a well-defined interface across which to communicate with other objects. It is easy think of individuals, populations or physical components of the ecosystem in terms of objects. As a further guideline for software design, Booch noted in Gamma et al. (1995), that 'all well-structured object-oriented architectures are full of patterns'; these patterns now form the basic engineering concepts in any software design.

The discussion of the proper design of ecological modelling software was initiated by Robertson et al. (1989), who experimented with different 'input languages' to aid biologists in formulating models. At the same time, Larkin and Carruthers (1989) presented HERMES, a fully-integrated, visual modelling tool for biologists developed in Smalltalk (the classic object-oriented programming language). Wenzel (1992) presented MOSES, a language to formulate ecological models consisting of 'a hierarchical structuring of models out of autonomous partial models residing in a model bank'. Thus, both HERMES and MOSES relied on the composite pattern of Gamma et al. (1995), but for different reasons neither tool was taken up: HERMES happened to be implemented on a platform that was falling out of use, while the syntax of MOSES may have been too demanding.

Silvert (1993) proposed object-oriented programming as the eminent tool for ecological modelling. Along the same line, Reynolds and Acock (1997) emphasised modular, generic model building blocks, as a design solution to the obvious lack of model re-use and the widespread duplication of efforts among ecological modellers. Papajorgji et al. (2004) exemplified an object-oriented framework for ecological models. However, its reliance on one particular software interface technology (CORBA) in the implementation may have played a role in the limited adoption of this design. Voinov et al. (2004) designed a framework for spatial, hydro-ecological models that combines modules written in STELLA (ISEE Systems, Lebanon, NH, USA) and C++.

In some cases, new models must be composed largely of existing models of various origins and designs. This problem was addressed by the programming frameworks ModCom (Hillyer et al., 2003) and CMP (Moore et al., 2007). With both frameworks, agro-ecological

* Tel.: +45 87 15 81 92; fax: +45 87 15 60 82.
E-mail address: niels.holst@agrsci.dk.

1574-9541/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ecoinf.2012.11.001

models, that were not necessarily conceived to work together, can be combined by wrapping existing models with code that make them compliant with the interfaces defined by the framework. Whereas ModCom depends on one particular technology (COM, Microsoft Inc.), CMP was designed more generically and can be implemented on any platform. An even more ambitious framework for the integration of highly diverse models was presented by Villa (2001). His solution involves a declarative syntax that uses XML (Bray et al., 2008) to describe model structures and interfaces.

The most ambitious designs of ecological models include formalism to describe both processes and semantic relations (Wenzel, 1992). Semantic modelling, however, is a demanding discipline whether expressed in mathematics (Uso-Domenech et al., 2006; and references cited therein) or a programming language, such as Prolog (Krivov et al., 2010). Villa et al. (2009) showed how semantic knowledge of a system can, in principle, be combined with a process model to simulate the system through 'semantic annotation of a model'. However, they also noted that 'the practicality of large-scale adoption remains to be fully understood'. In conclusion, it seems that ecological modelling is not yet ripe for the inclusion of semantics into model formalism.

Even though the history of ecological modelling is quite short, it has provided a multitude of models. On this background, we can look for the successful models and thus empirically characterise the best design. In systems biology, the SBML language soon became an international standard for the specification of cell biology models (Hucka et al., 2003). SBML, a dialect of XML, lets the user specify all details of the model, including its structure and parameter values. The original software was of limited capability but SBML files can now be handled by many common software packages. In individual-based modelling, NetLogo (Tisue and Wilensky, 2004) quickly won a large user group. It comes with a graphical user interface, a library of models and facilities for writing new models in the NetLogo language. The R software package (R Development Core Team, 2011) is highly generic. Operated from an old-fashioned prompt, it incorporates a programming language with access to a huge and extensible library of functions, some of which support modelling, e.g. 'popbio' for Leslie matrix models (Stubben and Milligan, 2007). All-purpose graphical modelling tools, such as STELLA, are also used for ecological modelling. They are useful for teaching and prototyping but are rather limited in their expressiveness and do not scale well to model large systems (Robertson et al., 1989; Villa, 2001). It is noteworthy that SBML, NetLogo and R are all communal, open-source development projects.

Silvert (1993) anticipated that, with object-orientated programming, future modellers would have available large libraries of standard ecosystem components which could be re-used, extended and combined at will. Arguably, this is not the case 20 years later. But the failure of object-oriented design to live up to its promise was already evident in 2001, when a group of industry experts created the foundation of 'agile' software development (Martin, 2006). The agile method defines a set of principles and practices which concur very well with the scientific process of ecological modelling (Holzworth et al., 2007). Even at a psychological level, the emphasis on 'motivated individuals' and 'self-organising teams' (Martin, 2006) should satisfy the outlook of most ecological modellers.

Given the review above, this paper aims to derive a universal design for ecological models. The envisioned design process is ideal. Thus, the design is not for uniting a collection of existing, disparate models but for developing new models the right way. The design should be modular and generic (Reynolds and Acock, 1997) and follow established design patterns (Gamma et al., 1995), as well as the agile design principles (Martin, 2006). Furthermore, the design should be implemented as open-source software, following common open standards and relying on no particular platform. First a universal design is derived, then, as a proof-of-concept, the design is implemented in a combination of C++ and XML and presented as the open-source software *Universal Simulator*.

2. The design

The design process (Parnas and Clements, 1986) went through five steps, designing first the computational (Section 2.1) and structural (Section 2.2) models, then the mechanisms for information flow (Section 2.3), modularisation (Section 2.4) and component creation (Section 2.5). Program listings are shown in C++-like pseudo-code and in XML. Class names are written with a capital first letter but only when this distinction is important.

2.1. Computational model

We envisage the simulation environment as a software construct, populated with objects representing simulation components, foremost the models which incarnate the dynamic behaviour of the ecosystem being modelled (Fig. 1). The Factory object (Abstract Factory pattern, Gamma et al., 1995) has the method 'create' used to populate the simulation environment with the Component objects needed for a simulation (Listing 1). This method takes a 'recipe' as input. The recipe describes the desired components and their configuration, such as parameter values. Upon creation each component's 'amend' method is called (Listing 1) to allow the component any additional self-configuration. For example, a component may itself need to create additional objects depending on its parameters.

```
// Listing 1 //
Factory::create(recipe) {
    createComponents();
    for each component[i]do component[i]->amend();
}
```

Once the environment is populated, a simulation can be carried out. For this we need a Simulation object with a 'run' method to execute the simulation in well-defined steps (Listing 2). Often a component will need to orient itself in the simulation environment, looking for other models with which to interact; this orientation phase is implemented in the component's 'initialize' method. When a simulation is run, it may involve replicate runs (iterations), for example if some model has stochastic behaviour. The first step in such an iteration is to 'reset' each component. For models, this involves setting all the state variables to their start value. Then in the innermost loop all components are repeatedly 'updated' in discrete time steps for the duration of the simulation. At the end of each model iteration, components may need some 'cleaning up', for example the closing of files, and at the very end a 'debriefing' may occur, for

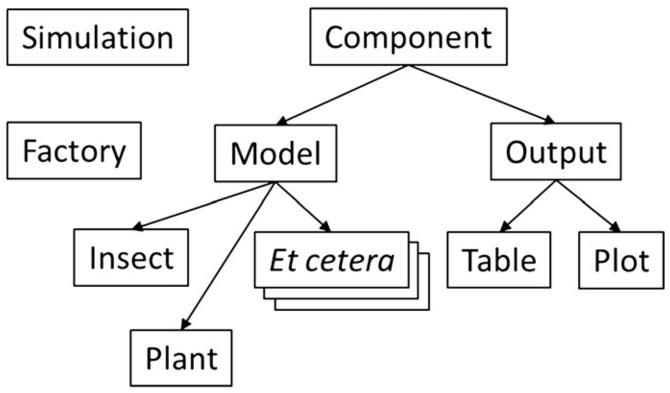

Fig. 1. Classes of the simulation environment. Simulation and Factory are Singleton classes (Gamma et al., 1995). Model and Output classes are derived from the Component base class and serve themselves as base classes for further derivation. '*Et cetera*' stands for additional classes derived from Model.