Frontiers in Ecology and the Environment

Increasing the reliability of ecological models using modern software engineering techniques

Robert M Scheller, Brian R Sturtevant, Eric J Gustafson, Brendan C Ward, and David J Mladenoff

Front Ecol Environ 2009; doi:10.1890/080141

Increasing the reliability of ecological models using modern software engineering techniques

Robert M Scheller[1*], Brian R Sturtevant[2], Eric J Gustafson[2], Brendan C Ward[1], and David J Mladenoff[3]

Modern software development techniques are largely unknown to ecologists. Typically, ecological models and other software tools are developed for limited research purposes, and additional capabilities are added later, usually in an ad hoc manner. Modern software engineering techniques can substantially increase scientific rigor and confidence in ecological models and tools. These techniques have the potential to transform how ecological software is conceived and developed, improve precision, reduce errors, and increase scientific credibility. We describe our re-engineering of the forest landscape model LANDIS (LANdscape DIsturbance and Succession) to illustrate the advantages of using common software engineering practices.

Front Ecol Environ 2009; doi:10.1890/080141

At its core, scientific knowledge is a collection of conceptual models that attempt to describe how the natural world works, and that have resisted repeated attempts to find empirical data to contradict them. However, solutions to broad-scale and complex environmental challenges often lie beyond the domain of traditional experimental or empirical approaches. Ecological models provide powerful tools to demonstrate or test the consequences of assumptions and to conduct virtual experiments to gain insight into complex ecological systems.

Ecological models (and by extension, any software tool used in ecological research) are typically developed as small, self-contained research projects. Beginning as relatively simple programs, many ecological models grow as funding opportunities arise, research needs evolve, and personnel change. This approach meets immediate research needs and produces quick (2–3 years) results. Over the past decade, however, there has been an increased demand for models capable of simulating multiple interacting processes and making ecological forecasts (Clark *et al.* 2001). Consequently, the scope and complexity of ecological models have increased. Because of the intellectual investment required to build complex tools, their longevity and distribution have also increased. Managing the long-term (> 5 years) growth of models – while maximizing their reliability – is becoming an important challenge. However, ecological models are typically built by ecologists, not computer scientists. Although many ecologists can write computer code, they typically lack software engineering literacy (Wilson 2006). As a result, ecologists often build or modify models in an ad hoc manner, neglecting essential software-building stages, such as testing and the documentation of the underlying architecture (the underlying internal design; Wilson 2006).

The adoption of rigorous approaches to building (or "developing") models also has the potential to substantially increase scientific rigor and confidence in the results. The scientific precision expected of models should equal that expected for any ecological research (Scholten and Udink ten Cate 1999). Furthermore, when models are used in decision making, model developers must minimize logical errors and maximize the reliability of the output.

Our objective is to introduce ecologists to state-of-the-art software development approaches. Ecologists cannot be expected to undertake the additional training necessary to become proficient in computer science. Instead, we hope that by highlighting some of the common shortcomings of model development, along with common solutions, the scientific value of ecological models can be improved. We also maintain that exposing ecologists to established processes for developing robust software can aid them as they manage complex software development projects. Finally, we relate these concepts to our own

In a nutshell:

- Ecological models and other software tools are typically built in an incremental, ad hoc manner, such that they can no longer address new hypotheses, and must therefore frequently be rebuilt
- Modern software engineering techniques can substantially increase scientific rigor and confidence in ecological software
- LANDIS-II, a newer version of the LANDIS forest landscape dynamics model, provides an illustration of an ecological model re-engineered using modern software development techniques

[1]*Conservation Biology Institute, Corvallis, OR* [*]*(rmschell@pdx.edu);* [2]*US Department of Agriculture Forest Service, Northern Research Station, Rhinelander, WI;* [3]*Department of Forest Ecology and Management, University of Wisconsin, Madison, WI*

experience in re-engineering an ecological model.

Although system design and architecture are major components of any project, we do not discuss either in detail here. There exist parallel suites of software engineering practices for designing system architecture. One example is "Object Oriented Design", which has been successfully applied to ecological tools (Mladenoff *et al.* 1996; Sequeira *et al.* 1997; Lorek and Sonnenschein 1998). However, beyond a limited degree of complexity, it is not necessary for most ecologists to become proficient in designing software architecture – rather, ecologists should become proficient in the process of developing a model. Improving the process will ultimately lead to more robust model architecture that can be readily adapted to new questions and hypotheses.

Common shortcomings of ecological model implementation

We begin by distinguishing a conceptual model from the computer program that implements it (henceforth, the "implementation"). A conceptual model consists of the science, including the formal logical and mathematical relationships, state variables, input data, and the sequence of calculations; it is typically peer-reviewed. The implementation is a translation of the conceptual model into code that a computer can read. There can be multiple implementations (ie code translations) of a conceptual model. Programming a model involves additional issues and challenges, unrelated to the underlying science. Approaches and methods for generating, verifying, and validating a conceptual model have been previously reviewed (Oreskes 1994; Aber *et al.* 2003; Gardner and Urban 2003). Approaches to more robustly link conceptual models and their implementation have to date received limited attention in ecology.

There are three principal shortcomings of model implementations: (1) failure to correctly or fully implement the conceptual model, (2) the inability to maintain the implementation through time, and (3) the inability to adapt the implementation to allow new hypotheses and questions to be addressed. Although failure to correctly implement the conceptual model can be identified, ease of maintenance or ability to readily expand the capability of a model is more difficult to measure. These are related to the management of complexity: how does one implement, maintain, and adapt a sufficiently complex ecological model?

A number of factors contribute to inadequate model implementations, including: (1) the complexity of the conceptual model, (2) the scale and scope of the project, (3) schedule and budget constraints, and (4) changing requirements for the model implementation (Foote and Yoder 2000). Many of these factors are beyond the control of the model developers. However, we believe the following general practices could readily improve the model development process and aid scientists when managing complex software development projects. These practices will have the added benefit of removing the dependency of a model on a single person or group, thereby increasing model longevity and the potential for replicating the methods.

Generating requirements

Requirements are the criteria that a software system must meet, and include implementation of the conceptual model and the expected interface between the user and the tool. Ideally, requirements are set before the beginning of the development process and remain stable throughout, but this is rarely the case in practice (Larman 2004). Requirements are particularly dynamic in a research setting, where the answers to a set of questions can lead to different directions of inquiry. These new directions may regenerate requirements that differ considerably from the original. For example, simple models often evolve into larger, more complex models (Minar *et al.* 1996). Even the most robust tool can erode over time as a result of changing requirements (Foote and Yoder 2000).

Before a model is developed or substantially revised, three sets of documents should be generated to clarify the requirements: (1) a project charter, (2) a conceptual model description, and (3) the expected model–user interface description. The project charter should include the reasons for developing the new model or tool, the objectives and expectations of the software over the next 5–10 years, constraints under which the implementation will take place, and the identities of the main stakeholders (Lewis 1995; Panel 1). The conceptual model description is written for both ecologists and computer scientists, and provides both the conceptual details of any requirements and a clear specification for implementing the model as software. This is distinct from the scientific literature that documents the conceptual model and algorithms, and should be geared toward enabling a programmer to generate the model implementation. Later, it can serve as an introduction to the model for both ecologists and programmers. Finally, the expected model–user interface description should define the desired interaction between the user and the particular tool, including the graphical or command tools necessary for operating the software.

Iterative and incremental development

Iterative development (also known as "incremental" or "iterative and incremental" development) is an approach to building a software system through multiple iterations (Larman 2004). Each iteration is a discrete mini-project, consisting of requirement definition, design, coding, testing, and assessment (Figure 1). Each iteration has discrete tasks that can be objectively evaluated. Working code is produced and documented at each iteration, with itera-

Panel 1. LANDIS-II project charter

Purpose

Our research objective is to produce a forest landscape simulation model, LANDIS-II, designed to study the effects of interacting natural (eg fire, wind, insects, deer) and anthropogenic (eg harvesting, climate change) disturbances on forest succession in large, heterogeneous landscapes. The model will produce predictions that can be tested empirically, in whole or in part. LANDIS-II will have the capability to be parameterized for multiple ecosystems and conditions.

LANDIS-II will comprise three main ecological components: succession, seed dispersal, and disturbances. The succession component will be designed to work with multiple disturbance components that modify the landscape, such as windthrow, harvest, insects, and fire. Tree species will be represented in LANDIS-II as cohorts; each cohort is defined by species and age range (eg all sugar maples, *Acer saccharum*, 1–10 years old). In addition, LANDIS-II will allow additional information to be associated with each cohort. A succession routine that tracks biomass accumulation (aboveground net primary productivity–mortality) for each cohort has been designed and will be available as an optional succession component. Each disturbance type will be a component that can be modified and replaced as necessary. The model will allow new disturbance components to be added at will. Succession and seed dispersal components can be replaced but are required. Finally, users will be able to select a time step that best fits the temporal scale of succession and disturbance within their ecosystem. Each ecological process will have the capacity to operate at a unique time step.

Project objectives

LANDIS will be redesigned into LANDIS-II from the "bottom up", resulting in an integrated but flexible landscape succession–disturbance model with flexible cohort information (species, age, and, initially, a biomass option) and a variable time step capability. The model will consist of a controller component and multiple, separate components that can be added as needed. Each component will be designed in the form of dynamic linked libraries. The model will require succession and seed dispersal components as a minimum and these will be designed and implemented first. Inputs and outputs will also be designed as components that will be available for user modification. LANDIS-II will be designed to accept additional disturbance components, so that other researchers will be able to create their own components and add them to LANDIS-II to suit their particular research needs. This will allow expanded usage of LANDIS-II, as other research facilities will be able to design custom disturbance components for their particular needs.

The redesign of LANDIS offers us the opportunity to (1) improve computational efficiency through improvements to the code, (2) remove both known and unknown sources of model error, (3) enhance the re-usability of code through superior design, and (4) provide seamless and simplified integration between model components.

Physical scope

- Spatial extent: large landscapes ~ 10^4 to 10^8 ha.
- Spatial resolution: 10 m x 10 m to 500 m x 500 m.
- Temporal extent: 50 to 2000 years.
- Temporal resolution: 1 to 40 years.
- Number of species: unlimited.
- Operating system: initially Windows OS.
- Programming language: C#, although the use of dynamic linked libraries (dlls) allows individual components to use other languages.

tions taking about 2–8 weeks to complete (Larman 2004). During iteration assessment, the entire project team reviews the tasks completed, assesses progress, identifies lessons learned, and defines the requirements for the next iteration. During assessments, requirements can be modified, allowing for flexible priorities and functionality. Testing – described in detail below – should be applied during every iteration. Testing and correcting a model or tool are most efficient when done regularly, because discovering problems early mitigates their impact on subsequent code. In addition, testing core components and isolating errors become increasingly difficult as additional complexity is layered onto the model.

Iterations can be grouped into four phases: conceptual modeling, elaboration, model building, and model release (Jacobson *et al.* 1999; Figure 2). During the conceptual modeling phase, scientists and programmers focus on gathering information about short- and long-term requirements. Scientists envision how the model will be used in 5 years, 10 years, and beyond (this vision is incorporated into the project charter). The elaboration phase identifies, develops, and tests the most challenging components of the underlying design. This phase is where most of the software design occurs and should result in a robust architecture that meets the defined requirements. The model building or construction phase implements the system and iteratively adds more complexity. Lastly, the model release phase finalizes implementation and testing of the complete system. Most of these activities will occur during multiple phases and iterations.

Iterative development encourages regular, consistent, and open communication among ecologists and programmers, thereby enabling frequent revision of model requirements. Iterative development also encourages risk taking and risk management: new ideas can be tested during a single iteration, and failure is therefore limited in scope while providing critical feedback. For example, if a model contains a new scientific understanding or paradigm, these features can be encapsulated within an early iteration and defined, built, and tested before additional complexity is added to the project. If the iteration fails (ie the new scientific understanding cannot be represented or the requisite architecture is not workable), the cost to the larger project is minimized.

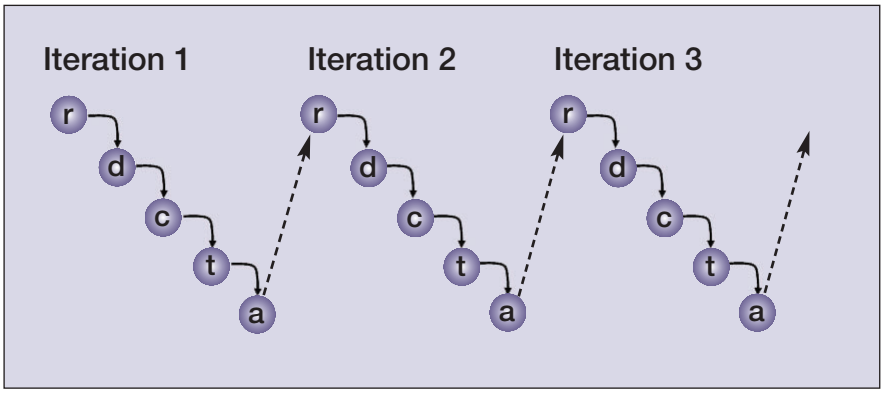

Figure 1. *Multiple development iterations consisting of writing requirement (r), design (d), coding (c), testing (t), and assessment (a). This is the most simplistic formulation of the software development life cycle.*

■ Software testing

Software testing is required to ensure consistency between the conceptual model and the implementation, and is therefore essential to maintaining scientific rigor. Unfortunately, software testing is typically undervalued within scientific model building; often, formal testing is neglected to save time, although ecologists may ultimately expend much more time correcting poorly implemented software.

Software testing includes unit testing, integration testing, and system testing (Figure 3). Unit testing is the testing of individual units (eg components, modules) of the software system, as opposed to system testing. A unit test is a pass–fail test for a function (a single component, such as an algorithm) or an interface (an interaction among components or between the user and the model). Unit tests can be defined prior to implementing the associated component. Indeed, generating unit tests prior to implementing a component can help refine the component's requirements and expected behavior. For example, a simple unit test could pass a controlled input value to a component containing a mathematical function, and verify that the output value matches the expected value. Another type of unit test would verify proper behavior when an illogical or out-of-range value is encountered. Unit tests are designed to be automatically repeated after any substantial changes to coding.

Integration testing combines components and tests their behavior and performance as a group. Integration tests typically occur after unit tests have been completed, and should be performed sequentially, from lower to higher levels of organization, such as testing a few components initially, then testing many components later. Integration testing focuses primarily on the validity of the communication among components. For example, an integration test could combine an input component, a processing component, and an output component, and use a controlled set of inputs to verify that the correct outputs were produced. This ensures that data were correctly shared among the three components, and that each component correctly interpreted the data passed from the other components.

System testing (also called "model verification"; Rykiel 1996) tests whether all the components are working together correctly, as a cohesive package. System testing evaluates the behavior of the entire system within the context of the specified requirements, and can be performed without knowledge of the underlying details of the implementation. System testing can also serve as a means of evaluating the requirements specified for the system, and can therefore provide a means of further refining requirements for later phases of development.

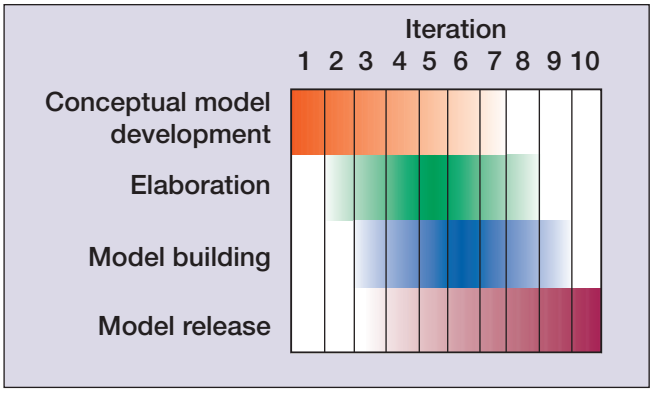

Figure 2. *The four phases of software development using the Unified Process. The actual number of iterations and the number of iterations in any given phase will vary widely, depending upon the project and personnel.*

■ Version control/configuration management

Version control is a system for tracking changes to source code over time, saving source code at set times, and resolving conflicts if multiple programmers are working on the same project. A "snapshot" of an iteration can be recorded before moving on to the next, allowing programmers to roll back development to a previous version if the exploration of risky options during an iteration is unsuccessful. Version control also enforces documentation of changes and facilitates co-development for geographically distributed teams. Free tools (eg Subversion, www.subversion.tigris.org) and web sites (eg SourceForge, www.sourceforge.net; Google Code, www.code.google.com) are available to quickly deploy version control.

■ Sustaining documentation

One of the greatest challenges for scientists developing models with long life spans is the maintenance of documentation. Ecological models evolve and change as the questions being asked change. Although a project charter

may have been written and peer-reviewed manuscripts published, documentation of the model implementation and components can quickly become out of date. Peer review of algorithms is unlikely, given the limited budgets for most projects. There are no easy solutions, and we recommend that developers: (1) Maintain documentation separate from the published descriptions so that the documentation can be readily updated. These can take the form of technical reports or user guides available online. (2) Use existing tools for documenting code. Many programming environments provide tools that can assist with the production of implementation-specific documents, such as descriptions of functions and relationship diagrams of model components. (3) Practice "literate programming" by writing code that maximizes readability (for humans) and includes descriptions of algorithms and associated full citations within the source code.

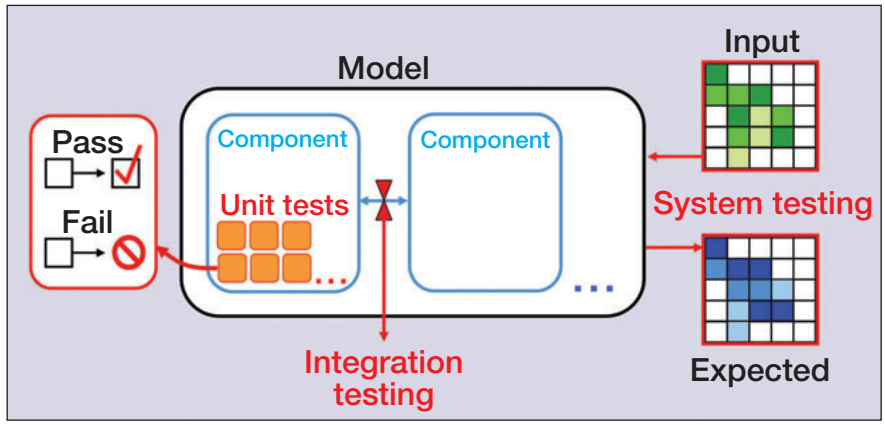

Figure 3. *The three principal types of software testing: (1) unit tests define pass/fail tests that are frequently applied; (2) integration testing assesses communication among components; and (3) system testing compares actual to expected outcomes.*

■ Software development processes

Over the past two decades, many groups have defined consistent suites of software development practices, including project management, programming styles, and tools to use (eg the Unified Modeling Language). These are called "software development processes" (Table 1), and each variant highlights different features. For example, some processes emphasize many short iterations, team development (eg paired programming, code review, common project rooms), and minimal documentation (Larman 2004). Others are architecture-centric and emphasize tackling the riskiest or most novel components of a project within the first few iterations. The appropriate software development process to adopt is dependent upon the available resources, the size and scope of the project, and the experience of the team members. Each process is generally intended as a guideline, and each project must identify which elements to emphasize or discard.

■ The LANDIS-II re-engineering experience

LANDIS (LANdscape DIsturbance and Succession) was originally conceived to scale the insights provided by forest gap succession models to landscapes to investigate interactions between fire, wind, and forest dynamics (Mladenoff 2004). The success of the LANDIS conceptual model is evident in its longevity and its use in different research efforts over time (Mladenoff *et al.* 1996; He and Mladenoff 1999; Gustafson *et al.* 2000; Sturtevant *et al.* 2004; Scheller and Mladenoff 2005). However, while the LANDIS *conceptual model* evolved as a research tool, the *program* that implemented the model could no longer support emerging requirements. As the number of represented processes expanded (eg Gustafson *et al.* 2000; Scheller and Mladenoff 2004; Scheller and Mladenoff 2005; Sturtevant *et al.* 2004), requirements also changed and expanded. Although the implementation matched

Table 1. Suggested process models, from most accessible (top row) to most advanced (bottom row)

Process	*Description*	*Reference or Uniform Resource Locator (URL)*
Software development life cycle	The development process is broken into components, including: requirement, design, coding, and testing. In its most basic form, these components progress linearly (Figure 1). Iterations are not used.	Post and Anderson (2006)
Unified process	Emphasizes iterative development, uses cases, development of a robust architecture, and risk assessment.	Jacobson *et al.* (1999); http://en.wikipedia.org/wiki/Unified_Process
Agile	Agile development uses very short iterations, frequent communication, and minimal formal documentation. Requirements can change often.	Post and Anderson (2006); http://en.wikipedia.org/wiki/Agile_software_development
Capability maturity model integration (CMMI)	A general framework, including a structured set of relevant best practices.	www.sei.cmu.edu/cmmi

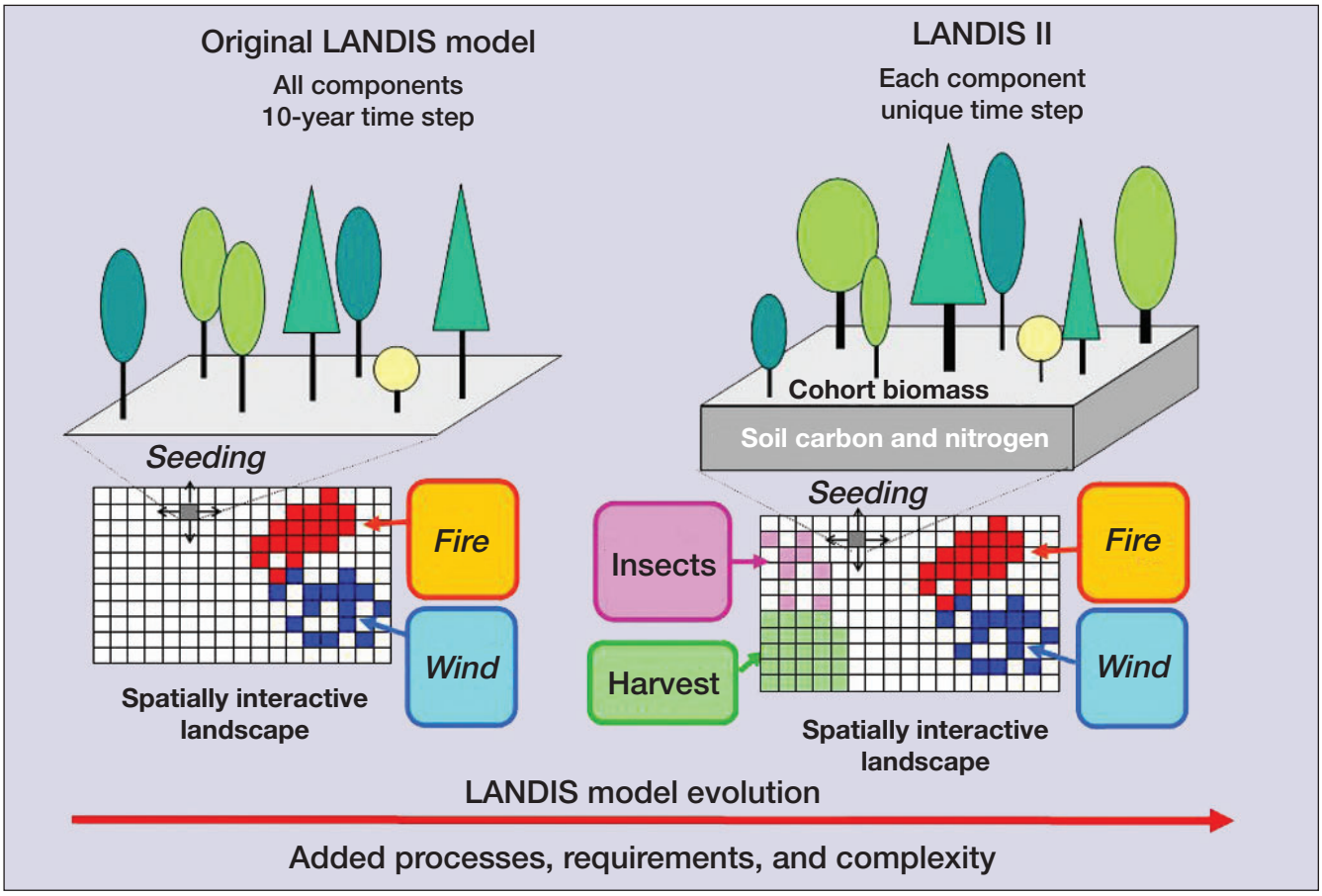

Figure 4. *The evolution of the LANDIS conceptual model through time.*

the conceptual model (ie the model behaved correctly), we could no longer adapt the model to meet our research needs (eg adding biomass accumulation, soil carbon, and nitrogen; Figure 4). We concluded that a complete re-engineering of the LANDIS program was required to meet these needs (Scheller *et al.* 2007).

Our first step was to formalize a vision for the next generation LANDIS model within a project charter (Panel 1). Key requirements identified included: (1) building upon the scientific assumptions of the original model, (2) performance optimization for large landscapes (ie millions of spatially interactive cells), (3) ability to accommodate multi-scale processes through process-specific time steps and variable cell resolution, and (4) enhanced opportunity for scientific collaboration through a flexible and extensible model architecture that would allow new ecological processes (implemented as extensions) to be added or removed, without altering the core model (Scheller *et al.* 2007).

Building upon the science of the original LANDIS model required a clear description of its underlying conceptual model. Like many ecological models, the conceptual model was not adequately described in a distinct document and existing published manuscripts did not contain sufficient information to re-implement the model. We therefore documented the conceptual model within a model description document, adding in new requirements defined within the LANDIS-II charter (Scheller *et al.* 2007).

Next, we adopted the Unified Process (UP) of software development (Jacobson *et al.* 1999). Distinguishing aspects of UP are that it is iterative and incremental and emphasizes risk mitigation. Important risks are addressed early in the development cycle (Jacobson *et al.* 1999). Model development required 13 iterations (Table 2). Because we addressed the riskiest component first – the extensible architecture – we quickly discovered advantages to using the C# programming language and switched away from C++, potentially saving considerable time. In addition, the iterations greatly facilitated the management of such a complex model. Communication among team members was frequent and we were able to follow an "adaptive management" approach, learning and documenting which development and design approaches were working and which needed to be changed or discarded. The testing that occurred at each iteration gave us confidence that the model was robust, and many problems were identified and fixed throughout the process. In general, of all the practices we used, iterations had the most immediate and tangible benefits.

Our testing initially verified that the core model correctly recognized and loaded the various extensions and correctly managed the variable time steps. Later, hundreds of simple pass/fail unit tests were written to ensure

that the model accurately identified user input errors. After the initial suite of extensions was written, we applied integration testing to individual extensions to compare outcomes with and without other extensions. Later, we performed system testing to ensure that model behavior matched that of the conceptual model. Our system tests used highly simplified and small landscapes that would allow us to quickly evaluate model behavior; only at the end of the development process was system testing applied to large, complex landscapes. We have since developed several additional extensions that have also been tested using unit tests, integration tests, and system tests, on both simple and complex landscapes.

Table 2. Major iterations used in the development of LANDIS-II (version 5.0)

Iteration phase type	*Number*	*Description*
Conceptual modeling	1	Defining the LANDIS-II charter
Conceptual modeling	2	Defining LANDIS-II features and requirements
Conceptual modeling	3	Assessing project risks
Elaboration	4	Landscape module: site variables and iterators
Elaboration	5	C++ and Plug-n-Play (dynamic linked libraries)
Elaboration	6	Investigating C#
Elaboration	7	Rounding out the core
Model building	8	Wind disturbance extension and species data manager
Model building	9	Age-only succession, continue work on species data
Model building	10	Ecoregions and reproduction
Model building	11	Alpha 1 release
Model building	12	Performance testing
Model building	13	Performance tuning
Model release	14	Beta 1 release
Model release	15	Release candidate 1
Model release	16	Official release

Finally, the LANDIS-II development process was considerably enhanced and augmented through open-source, online tools. We created a web site (www.landis-ii.org) containing detailed iteration plans and reviews, enabling our geographically distributed science team to track progress and coordinate efforts. Version tracking software (a Subversion code repository with TortoiseSVN client software) accessed through the Internet allowed teams of programmers to coordinate and share code in real time.

Recommendations for getting started

We recommend easing into these software development practices. Ecologists should learn how to write requirements, plan iterations, and use version control before choosing a particular process (Wilson 2006; Table 1). When implementing best practices, focus first on those that fit your needs and budget. Deploy additional concepts and processes incrementally, as you learn what works and what does not. If you choose to adopt a defined process, it is not necessary to follow the defined approach exactly. Instead, remain adaptive and use those elements that are attainable and valuable. We advocate using all the model testing approaches available – writing unit tests, testing components after every iteration, integration testing, and system testing. We recommend that iterative and incremental development should always be employed. If the tool being developed is a model, we recommend exploring some of the many existing frameworks for model implementation (eg Simile [Muetzelfeldt and Massheder 2003], Stella [Costanza and Voinov 2001], SELES [Fall and Fall 2001]), although the approaches outlined should still be deployed even when working within an existing framework. Finally, we recommend further research on software design and architecture, recognizing that it may eventually be necessary to consult with or employ computer scientists.

Conclusions

Software fails when the implemented code is not consistent with the conceptual model or the relationship between the conceptual model and the code can no longer be ascertained. Failure is not uncommon in software development and we should expect that such failures occur routinely during the development of ecological tools. Because of the need for scientific rigor and the high potential for failure, developers of ecological tools have a strong motivation for adopting standardized processes and best practices.

Complex ecological models represent a substantial investment in time and money. Some of the more sophisticated and general models may be used for a decade or more, to address a variety of questions, and often by a user community that did not develop the model. Early investment of time and resources in the underlying vision and architectural design of a model not only provides the flexibility to adapt to new questions, but also saves many more times that investment in software maintenance and user support.

A key challenge for the future of ecological modeling is the management of complexity. The methods and processes described here, adopted from software engineering, are proven techniques that will enable ecologists to develop the next generation of models with the same rigor and reliability expected of our statistics and experimental designs. If we allow system and model complexity to overwhelm us, the potential contribution and promise of ecological models will be lost.

Acknowledgements

B Cummings first introduced us to software development processes and challenged us to adopt software engineering techniques. J Domingo implemented the

core LANDIS-II architecture. J Fisk, T Spies, M Moran, and others provided critical feedback on the manuscript. Some key concepts underlying the LANDIS-II architecture, including modularity and the use of dlls, were elucidated through a series of LANDIS development workshops. In addition to the authors, key contributors to the workshops included H He, W Li, ZB Shang, J Yang, D Lytle, and S Shifley. Funding for LANDIS-II was provided by the US Department of Agriculture Forest Service, Northern Research Station, and the National Fire Plan.

References

Aber JD, Bernhardt ES, Dijkstra FA, *et al.* 2003. Standards of practice for review and publication of models: summary of discussion. In: Canham CD, Cole JJ, and Lauenroth WK (Eds). Models in ecosystem science. Princeton, NJ: Princeton University Press.

Clark JS, Carpenter SR, Barber M, *et al.* 2001. Ecological forecasts: an emerging imperative. *Science* **293**: 657–60.

Costanza R and Voinov A. 2001. Modeling ecological and economic systems with STELLA: part III. *Ecol Model* **143**: 1–7.

Fall A and Fall J. 2001. A domain-specific language for models of landscape dynamics. *Ecol Model* **137**: 1–21.

Foote B and Yoder J. 2000. Big ball of mud. In: Harrison N, Foote B, and Rohnert H (Eds). Pattern languages of program design 4. Reading, MA: Addison-Wesley.

Gardner RH and Urban DL. 2003. Model validation and testing: past lessons, present concerns, future prospects. In: Canham CD, Cole JJ, and Lauenroth WK (Eds). Models in ecosystem science. Princeton, NJ: Princeton University Press.

Gustafson EJ, Shifley SR, Mladenoff DJ, *et al.* 2000. Spatial simulation of forest succession and timber harvesting using LANDIS. *Can J Forest Res* **30**: 32–43.

He HS and Mladenoff DJ. 1999. Spatially explicit and stochastic simulation of forest landscape fire disturbance and succession. *Ecology* **80**: 81–99.

Jacobson I, Booch G, and Rumbaugh J. 1999. The unified software development process. Boston, MA: Addison-Wesley Professional.

Larman C. 2004. Agile and iterative development: a manager's guide. Boston, MA: Pearson Education Inc.

Lewis JP. 1995. Fundamentals of project management. New York, NY: American Management Association.

Lorek H and Sonnenschein M. 1999. Object-oriented support for modelling and simulation of individual-oriented ecological models. *Ecol Model* **108**: 77–96.

Minar N, Burkhart R, Langton C, and Askenazi M. 1996. The swarm simulation system: a toolkit for building multi-agent simulations. Santa Fe, NM: Santa Fe Institute. Working Paper 96-06-042.

Mladenoff DJ, Host GE, Boeder J, and Crow TR. 1996. LANDIS: a spatial model of forest landscape disturbance, succession, and management. In: Goodchild MF, Steyaert LT, Parks BO, *et al.* (Eds). GIS and environmental modeling: progress and research issues. Fort Collins, CO: GIS World Books.

Mladenoff DJ. 2004. LANDIS and forest landscape models. *Ecol Model* **180**: 7–19.

Muetzelfeldt R and Massheder J. 2003. The simile visual modelling environment. *Eur J Agron* **18**: 345–58.

Oreskes N, Shraderfrechette K, and Belitz K. 1994. Verification, validation, and confirmation of numerical-models in the earth-sciences. *Science* **263**: 641–46.

Post G and Anderson D. 2006. Management information systems: solving business problems with information technology, 4th edn. New York, NY: McGraw-Hill Irwin.

Rykiel Jr EJ. 1996. Testing ecological models: the meaning of validation. *Ecol Model* **90**: 229–44.

Scheller RM, Domingo JB, Sturtevant BR, *et al.* 2007. Design, development, and application of LANDIS-II, a spatial landscape simulation model with flexible spatial and temporal resolution. *Ecol Model* **201**: 409–19.

Scheller RM and Mladenoff DJ. 2004. A forest growth and biomass module for a landscape simulation model, LANDIS: design, validation, and application. *Ecol Model* **180**: 211–29.

Scheller RM and Mladenoff DJ. 2005. A spatially interactive simulation of climate change, harvesting, wind, and tree species migration and projected changes to forest composition and biomass in northern Wisconsin, USA. *Glob Change Biol* **11**: 307–21.

Scholten H and Udink ten Cate AJ. 1999. Quality assessment of the simulation modeling process. *Comput Electron Agr* **22**: 199–208.

Sequeira RA, Olson RL, and McKinion JM. 1997. Implementing generic, object-oriented models in biology. *Ecol Model* **94**: 17–31.

Sturtevant BR, Gustafson EJ, Li W, and He HS. 2004. Modeling biological disturbances in LANDIS: a module description and demonstration using spruce budworm. *Ecol Model* **180**: 153–74.

Wilson GV. 2006. Where's the real bottleneck in scientific computing? *Am Sci* **94**: 5–6.

