NATURAL RESOURCE MODELING
Volume 15, Number 1, Spring 2002

SOFTWARE ENGINEERING CONSIDERATIONS FOR INDIVIDUAL-BASED MODELS

GLEN E. ROPELLA
The Swarm Development Group
624 Agua Fria
Santa Fe, NM 87501
E-mail: gepr@acm.org

STEVEN F. RAILSBACK
Lang, Railsback & Associates
250 California Avenue
Arcata, CA 95521
Email: LRA@Northcoast.com

STEPHEN K. JACKSON
Jackson Scientific Computing
McKinleyville, CA 95519
E-mail: Jackson3@Humboldt1.com

ABSTRACT. Software design is much more important for individual-based models (IBMs) than it is for conventional models, for three reasons. First, the results of an IBM are the emergent properties of a system of interacting agents that exist only in the software; unlike analytical model results, an IBMs outcomes can be reproduced only by exactly reproducing its software implementation. Second, outcomes of an IBM are expected to be complex and novel, making software errors difficult to identify. Third, an IBM needs 'systems software' that manages populations of multiple kinds of agents, often has nonlinear and multi-threaded process control and simulates a wide range of physical and biological processes. General software guidelines for complex models are especially important for IBMs. (1) Have code critically reviewed by several people. (2) Follow prudent release management practices, keeping careful control over the software as changes are implemented. (3) Develop multiple representations of the model and its software; diagrams and written descriptions of code aid design and understanding. (4) Use appropriate and widespread software tools which provide numerous major benefits; coding 'from scratch' is rarely appropriate. (5) Test the software continually, following a planned, multi-level, experimental strategy. (6) Provide tools for thorough, pervasive validation and verification. (7) Pay attention to how pseudo-random numbers are generated and used. Additional guidelines for IBMs include: (a) design the model's organization before starting to write code, (b) provide the ability to observe all parts of the model from the beginning, (c) make an extensive effort to understand how the model executes—how often different pieces of code are called by which objects, and (d) design the software to resemble the system being modeled, which helps maintain an understanding of the software. Strategies for meeting these guidelines include planning adequate resources for software development, using software professionals to implement models and using tools like Swarm that are designed specifically for IBMs.

KEY WORDS: Individual-based model, observability, software, validation, verification.

Introduction. Individual-based models (IBMs) simulate systems of independent agents interacting with each other and their environment. The technique of building and using IBMs is an essential tool for understanding complex systems and is increasingly applied to ecological, economic and sociological issues related to natural resource management.

While most natural resource models are implemented in computer software, IBMs are uniquely dependent on their computer implementation. The root cause of this dependence is that the results of IBM simulations we are interested in are the complex, emergent behavior of a system of interacting agents, and these agents exist only in the software. A conventional analytical model can be solved many ways (e.g., by using calculus or a variety of numerical methods to solve the equations) all of which should produce the same results. An IBM, however, has no intrinsic "solution" and its results are completely (and often subtly) dependent on the computer methods used to implement the model. The only way to exactly reproduce an IBM simulation is to exactly reproduce its software implementation. Because they represent complex systems and abandon such concepts as equilibrium or convergence, IBMs are expected to produce novel and dynamic outputs, making the artifacts of software mistakes difficult to distinguish from valid results.

The close dependence of the results of an IBM on its software is undoubtedly one of the main reasons why many scientists and managers remain wary of the technique. Therefore, using appropriate software engineering methods is a critical part of advancing the IBM technique to acceptance as an essential tool for ecology and natural resource management. Individual-based simulations will not, and should not, be accepted as credible unless the software issues addressed here are given due attention.

The importance of software design for IBMs has been recognized, yet software engineering remains inadequately addressed in many individual-based modeling projects (Grimm [1999]; Grimm et al. [1999]; Lorek and Sonnenschein [1999]; Railsback [2001]). To date, little specific guidance on implementing IBMs in software has been published in the natural resource and ecological modeling literature.

The goal of this paper is to identify important software engineering issues for IBMs and provide general recommendations for addressing these issues. Our intent is to provide guidelines that will help an IBM project develop software that (1) does not contain important undetected errors, (2) helps keep the project efficient and cost-effective, and (3) provides the tools necessary to conduct valid science or responsible management with the model. Early detection of mistakes is crucial to the cost-effectiveness of any software project. Failure to detect mistakes leads to spending much of the project's time and budget on work that must be discarded after mistakes are found or (worse yet) the promulgation of "research" results or management decisions based on erroneous simulations. Likewise, failure to provide key software features can make it difficult or impossible to understand and learn from an IBM. These guidelines are based mainly on the authors' extensive experience implementing IBMs and designing software libraries for individual-based simulation.

We address three specific objectives. The first is to provide a set of software issues and recommendations that are applicable to modeling projects in general. The second objective is to describe why IBMs have different software considerations than conventional natural resource models. Finally, we provide recommendations specifically for developing IBM software.

2. General software guidelines. This section briefly presents recommendations on seven software issues that apply to any modeling project of nontrivial complexity.

2.A Conduct critical reviews. Software is like other products of science that greatly benefit from critical review. Review is especially important in the unfortunately common situation in which one person both designs the model and implements it in software. As with review of written work, software reviews are most likely to be successful in improving the code if they are conducted by multiple reviewers that are adequately qualified and interested. Reviews should address all aspects of a model—its objectives, written specification and software. We find that a large majority (but not all) of software mistakes can be found in a careful review process.

An efficient way to produce quality code is for the model to be implemented by a team of modeler and programmer. The modeler (a scientist that understands the system being modeled and has primary responsibility for the model's formulation) writes the model's full specification out and provides it to the programmer. The programmer writes software implementing the model specification, at the same time reviewing the specification for consistency and completeness. The modeler then reviews the software to ensure that it faithfully implements the model. Our experience has been that this process cost-effectively produces both code containing few undetected mistakes and a complete and accurate written model description of the model.

An important secondary benefit of the software review process is that it promotes a clear and understandable coding style. For reviewers to understand and check software, the code must be written in a relatively simple and understandable fashion, and simple, understandable code is more likely to be error-free. Even if copious comments are included, the actual code statements must be understandable to reviewers and largely self-documenting. Using variable names that fully and accurately describe the variables' meaning is helpful; we recommend key variable names be assigned by the modeler instead of the programmer. It is also usually beneficial to use simple and clear coding constructs instead of overly clever algorithms that may be harder to understand and more subject to obscure errors. Clear and simple code is desirable even if it reduces execution speed because it typically reduces the time spent writing, checking and verifying the code and therefore shortens the software development time (our experience with IBMs is that computer execution time rarely limits the rate at which science is conducted).

2.B Use release management. Keeping careful control over the evolving software is critical for avoiding mistakes and wasted effort. Common, prudent software engineering practices include:

- Using version control software that automatically documents who made what changes in the software and when. Version control is essential for keeping track of exactly what is implemented in each piece of code, thus allowing changes to be undone easily.
- Packaging and archiving different versions of a model as changes and additions are made.
- Using automated processes to build and distribute software packages; for example, providing tools like installation software. Such tools minimize the opportunities for software users (or developers) from unintentionally losing, corrupting or misusing code and input files.
- Careful tracking of when changes are made to, and errors removed from, each version of a model.
- Assigning specific responsibilities for conducting software maintenance tasks; even if only one person is responsible for a code, maintenance tasks should be scheduled and documented.

2.C Create multiple representations of a model. It is essential that a model be described in several ways, including a full written description as well as its software implementation. Models that are completely described only by their software, and not in written text, mathematics, diagrams, etc., are of little value for science or resource management. The lack of such multiple representations of a model makes software reviews much less valuable because there is nothing to check the software against. Multiple representations make a model easier to communicate and reproduce, both of which are critical for conducting science. Models of complex systems (as most IBMs are) are especially important to represent in multiple ways because complex systems require more effort to understand.

It is also valuable to represent the model's software in multiple ways. Flow charts, class hierarchy and class relationship diagrams for object-oriented codes, and various kinds of code maps help in designing and understanding software for complex models. We often write a verbal description of the function of each important method (or subroutine) in a complex code, which makes future code changes (even by the same programmer) easier and less error-prone. Software is available to automate some of these code documentation processes.

2.D Use widespread tools. There are many software development tools available for modelers, and using them has many advantages. For IBMs and natural resource models, available tools include high-level simulation languages and graphical environments, code libraries (several of which are available specifically for IBMs; see Section 4), and existing codes (Lorek and Sonnenschein [1999]). Using these tools can greatly reduce the amount of code that needs to be written from scratch, reducing costs and error potential. The right software tools can also provide the kinds of interfaces (graphs, animations, etc.) needed to fully observe and test models. Using existing tools also reduces the amount of documentation needed for a model, since the existing code should already be fully understood and documented, with its design justified. Searching for appropriate development tools should be an early step in any software project, and those with large projects should continually watch for useful new tools becoming available. Given the tools now available, it is difficult for us to envision any situations where building code from scratch in a base computer language (e.g., C++, Java) would be a wise decision.

There is an important secondary advantage to using widespread tools: the benefits of cooperation and sharing with fellow modelers. Just as science advances by individual researchers building on the work of others, simulation toolboxes grow in power and quality as more people use them. Model developers should be active in the user community for their chosen software tools, reporting any errors found and contributing a new code that is potentially useful to others. The result of this community activity is a continual increase in the reliability and usefulness of the software tools.

2.E Test continually. Treating software testing as a low, late or optional priority is a sure sign that a modeling project is in trouble. Software testing is a verification process—showing that the software accurately implements the model. (Note that software testing therefore requires that the model be represented separately with a written or other description—software cannot be verified if there is no other description of the model to verify it against.) Software mistakes are inevitable in complex models, and finding as many mistakes as early as possible is critical to project success. Therefore, comprehensive testing measures must be built into the software development process.

A critical element of successful and efficient software testing is applying a clear experimental strategy to the process. Testing should be treated as a scientific enquiry, with the testers designing experiments, predicting the outcomes of the experiments and then running the code to compare actual outcomes to the predicted outcomes. Documentation is also an essential part of the testing process—there is little value in testing a code without also providing the documentation to show future developers, users and clients exactly what testing was completed.

The following sequence of methods provides a minimum appropriate level of testing for complex codes. (Note that these methods do not include the all-important task of testing the model's design; they only test whether the software faithfully implements the model.)

- Code reviews (Section 2.A above).
- Spot checks of key model subcomponents. For a small set of example cases, the input and output of specific routines can be obtained and hand-checked. This helps find code errors that have widespread effects. However, spot-checks are not adequate by themselves because they cannot test all the model's possible combinations of variables and conditions.
- Pattern tests. The modeler executes example simulations and observes patterns of behavior via graphical outputs. Observing unexpected or unrealistic behavior often quickly leads to the identification of a mistake in the code or formulation.
- Systematic tests against an independent implementation. Because an IBM typically has many potential combinations of input variables, control paths and intermediate results, the only way to test software thoroughly is by implementing it independently in two codes and looking for differences in the output of the two codes. These tests are best conducted separately for each major component of the model as they are added to the code. Testing software must be written to execute the model component over a wide range of inputs, recording the inputs and results for comparison to another implementation of the same part of the model. This testing software is often a separate driver program, or it can be written into the full model code and switched on and off. We often program our models' major components in spreadsheets and compare spreadsheet calculations to intermediate results printed

out from the whole model's code, over a wide range of input values. This approach can be an easy and inexpensive way to test the model for thousands of cases, finding the inevitable mistakes that only occur in rare circumstances.

2.F Provide tools for pervasive validation and verification. To thoroughly verify that software is error-free, and to thoroughly validate a model's formulation, it is necessary to be able to collect data from throughout the model. A model may produce quite believable output even if many of its subcomponents have major errors in design or implementation. Therefore, software should be designed to provide easy access to all of its parts—during verification and validation, model users should be able to observe the state of all model components. The need for pervasive validation should be an important consideration in selecting software tools; at least some widely used simulation platforms provide facilities for observing all parts of a model. Grimm [2002] discusses design and use of graphical tools for validation and verification.

An important element of verification is identifying and preventing run-time errors. A run-time error occurs when the software reaches an invalid or erroneous state during execution as a result of input values instead of mistakes in code syntax. Run-time errors due to invalid or uninitialized parameter values and input data errors are common; these can be prevented by including code to check for errors or detected by using the tools provided for pervasive validation and verification. Some widely used programming languages do not necessarily abort when division by zero occurs (for example, we found variables given the value "Not a Number" following a division operation in which the denominator was e raised to a large negative number) or when messages are sent to null objects and nil methods. Without pervasive verification, such run-time errors can cause significant undetected errors.

2.G Pay attention to pseudo-random number generators. Many simulators use pseudo-random numbers to represent some processes. In most cases, pseudo-random numbers are used to represent the outcome of processes that occur at too fine a resolution to be represented mechanistically, providing diversity to the model's results without excessive model detail.

Pseudo-random numbers are provided by generator software, and pseudo-random number generators vary widely in quality. Generators vary in their execution speed, but much more importantly, in their cycle length or the number of values they produce before starting to repeat the sequence. The built-in generators in some computer languages and spreadsheets can have short cycle lengths that can induce artifacts into simulation results. Substandard generators can also have tendencies to produce occasional sequences of values having non-random trends. Such non-random tendencies can depend on the computer architecture (e.g., a generator that passes standard tests for randomness on a 64-bit machine may not pass on a 32-bit machine). These problems are of additional concern if a model's design is such that the random number sequence affects the sequence of events in the model; such model designs are generally questionable.

Modelers need not become experts on this issue (which has been studied extensively by computer scientists), but need access to expertise. Wilson [2000] discusses this issue in an ecological modeling context. Random number generation is an issue for which choosing the right software tools is a good solution. Some simulation platforms provide a variety of high-quality generators and documentation on the advantages of each, but if a software platform does not include documentation discussing the quality of its random number generators, it may not be a good tool for developing a complex model.

3. Why IBMs are different. While the above issues apply to complex models in general, IBMs have unique characteristics that make several other software issues important (Section 4, below). This section identifies those unique characteristics.

The primary difference between IBM software and code implementing conventional natural resource models is that IBM software is essentially "systems software," not a monolithic or single-purpose code. An IBM represents systems of heterogeneous, interacting agents and the software must manage these systems in addition to coding the characteristics of each type of agent. Some properties of IBMs resulting from this system's nature are:

- An IBM represents populations of (often) several types of agent, instead of single instances of each module. A conventional ecological model may have several modules, each representing the population of one species or habitat conditions. An IBM, instead, may have code representing an individual of one or more species, and code representing the habitat patches the organisms use; in addition the IBM has code to manage the populations of multiple copies of both organisms and habitat patches. The software for managing these populations can be as important as the software defining each type of agent.

- The modules within an IBM are heterogeneous at several levels. In a typical IBM there is a hierarchy of agent types, including habitat units and organisms of one or more species and life stages. At the higher levels, each broad category of agent type (organism, habitat unit) has a completely different structure and rule set. At intermediate levels, organisms of similar species or of different life stages within a species may share some rules but not others, or share rules but use different parameter values. At the lowest level are individuals of the same type that all use the same rules and parameter values, but each has its own state variable values (location, age, size, energy reserves, etc.). Only in the simplest models might there be multiple model modules that are exactly the same. At the other extreme are IBMs where new model structures emerge during simulations: emergent flocks or herds of animals, symbiotic structures, or patterns of spatial distribution may be as important to model results as the modules written into the code. One of the primary benefits of IBMs is that they can represent variation among individuals and organizational hierarchies; however, the heterogeneity of model modules means that the modeler's perception and understanding of the model is highly dependent on the software tools for collecting data from the model's diverse modules.
- Process control in an IBM is often multi-threaded and variable, instead of strictly procedural and linear. The sequence in which program statements are executed often depends on the events occurring in an IBM. This is especially true when an "event-based" software architecture is used; this approach allows model agents to trigger the events that occur next (for example, when an animal enters a habitat patch, it may trigger the other animals

in the patch to execute their code that determines whether they stay or leave).

- An IBM typically includes a wider range of dynamic processes than do conventional models. Multiple ways in which individuals can adapt (e.g., via behavior, evolution, coevolution) are often included in IBMs. This is another reason why understanding the cause of IBM outcomes can be a challenge.

4. Guidelines for IBM software development. As a consequence of the properties of IBMs identified in Section 3, IBM software requires additional engineering to assure that models are complete and coherent. We present four key issues in IBM software engineering, discuss how one software platform for IBMs (Swarm) helps address these issues and provide additional general recommendations for implementing IBMs.

4.A Key issues in IBM software engineering. There are four general issues that developers of IBMs need to consider from the start of a project. Understanding these issues and designing the software in consideration of them will greatly increase the odds that an IBM project will be efficient and successful.

Process. The model's process is the sequence of events that occur during a run. In an IBM, the process design can have a great influence on results—executing the same events in a different order can produce different outcomes. The modeler needs to carefully consider the desired order of events, and the software developer must be able to enforce the modeler's desired process.

Causality. This issue is the need to understand *why* a model produces the results (both intermediate and final) that it does. Little can be learned from an IBM that provides only outcomes with nothing known about why the outcomes occurred. The model and software should be designed so that the events causing a specific model state can be determined. This concept applies both to final model outcomes and to intermediate states of model individuals.

Constituency. The constituency of a model code is the classification of the various kinds of things in the model. For example, in designing the constituency of an IBM one might consider such questions as whether two similar species (or habitat types) have separate code or share code, but use different parameter values. A logical and consistent constituency makes an IBM easier to understand and makes the software simpler and less error-prone.

Observability. Testing software, conducting pervasive validation and verifying and understanding a model's causality all require that many parts of an IBM be observable to users. In our experience the failure to use software that makes the models sufficiently observable, and therefore testable, is one of the main reasons why few IBMs have contributed successfully to ecology and natural resource management.

Because IBMs are by definition driven by the actions and interactions of individuals, they are not truly testable unless the individuals can be observed. Observing individuals' state variables such as age, size and growth rate is necessary for verifying the software that calculates these variables and for validating the assumptions used to model them.

The interactions among agents, and among agents and their environment, must also be observable for an IBM to be truly verified and validated. Graphical user interfaces (GUIs) are an essential tool providing this kind of observability (Grimm [2002]). For spatial models, observing the location of individuals in space as a model executes, is especially informative. Using GUIs to observe the "locations" of individuals in other model dimensions besides space can also be helpful. The GUIs attached to conventional models are sometimes merely a nonessential add-on; however, the users of IBMs with GUIs typically consider the GUIs to be absolutely essential for model verification and validation. In every model we have developed, the GUI showing animal locations over time has allowed us to rapidly identify and correct at least one important error in the software, the model design or the input data. Many of these errors had major effects on model results, but almost certainly would not have been detected without the GUI. Our experience indicates that IBMs that lack GUIs are very unlikely to be free of important errors.

Observability is also an issue in how an IBM reports population-level results. If IBM software reports only the number and mean state (size, age, etc.) of the population, then important information on variation among individuals is unavailable to the user. The variability, uncertainty and observer-bias concerns that apply to studies of real natural systems also apply to how an IBMs software reports population data to the model user.

4.B A software platform that addresses IBM needs: Swarm. One good way to address the issues raised above concerning IBM software is to use a software platform designed specifically for the purpose. The Swarm simulation system is a software library originally developed at the Santa Fe Institute to facilitate agent-based simulation in the study of complex adaptive systems (Minar et al. [1996]). (Several other similar platforms are also now available.) Swarm is currently maintained by the non-profit Swarm Development Group, with information available at `www.swarm.org`. Because Swarm is a software library, model developers write the code specific to their model and its agents (e.g., what individual organisms do; how habitat is represented) in a common programming language. Most of the software needed to build, maintain and observe the individual agents and to control the process is provided by the Swarm libraries. In addition to software libraries, Swarm also provides a set of conventions for organizing IBMs.

An additional, very valuable, part of Swarm is its user community. A diverse and active group of scientists working on agent-based simulation provides ideas and code of great benefit to IBM developers.

Following is a summary of the ways that Swarm is designed to address the specific concerns of implementing IBMs.

Process. Swarm includes a variety of scheduling mechanisms that provide control over the execution of model actions. A “swarm” is defined as a collection of model agents and a specific schedule of actions that the agents execute. A Swarm model has a hierarchy of swarms, each with its own schedule. Typically, the highest level swarm is an observer swarm, and its schedule may (1) execute the model swarm for one time step, then (2) update the graphical and file outputs. The model swarm contains the actual model agents and its schedule typically includes (1) updating habitat conditions, (2) telling the model individuals to execute their schedule and (3) performing bookkeeping and clean-up tasks like removing dead individuals from the model. The individuals can have their own schedule in which they are told the order to conduct such actions as moving, feeding, reproducing and undergoing mortality. Swarm also allows event-based actions, in which a model maintains a queue of events waiting to be executed; the agents or the model itself can add events to the queue. Swarm provides a "logical concurrency model", an explicit definition of the sequence in which the events treated in the model as occurring concurrently are actually executed by the computer.

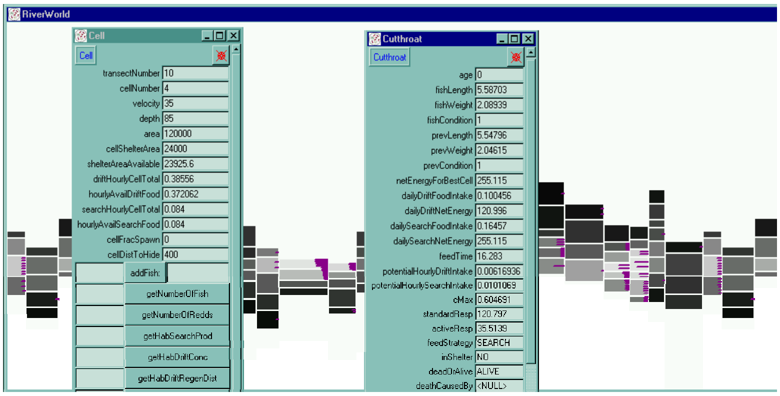
Causality. Understanding the cause of events in a model is greatly facilitated by Swarm's process control and observer tools. The first step in understanding why a particular model state occurred is understanding the sequence of model actions that preceded the state; as discussed above, Swarm provides easy and explicit control over event scheduling. Swarm's observer tools (below) allow modelers to trace the state of individuals and aggregations as a model executes, which is often essential for understanding causality.

Constituency. Swarm follows the object-oriented paradigm for organizing a model and provides several additional levels of constituency. Object-oriented programming is a natural fit to individual-based modeling. In the object-oriented paradigm, components of a model are individual objects, with one or more objects to a class and hierarchies of classes. In a typical object-oriented IBM there may be several species of organism that are subclasses of a general organism class. Each species can inherit some code from the general organism class but also has some species-specific code. Similarly, all the individuals of a species have the same code, but each individual has its own state variables. Likewise, habitat units may be organized in subclasses (e.g., for meadow, forest, lake). With its hierarchical organization of individual objects, object-oriented programming resembles the natural systems modeled with IBMs, so it is a natural approach to the constituency of an IBM.

Via the concept of a "swarm," a collection of objects and their action schedule, Swarm provides a second level of organization. Designing the exact contents of a swarm is an important step in designing a model.

Observability. Providing a high level of observability is one of primary benefits of Swarm. Observer tools that can be built into a Swarm model with a few lines of code include:

- A control panel that allows model execution to be stopped and restarted at will.
- "Probe displays," windows displaying the state of any model



transectNumber	10
cellNumber	4
velocity	35
depth	85
area	120000
cellShelterArea	24000
shelterAreaAvailable	23925.6
driftHourlyCellTotal	0.38556
hourlyAvailDriftFood	0.372062
searchHourlyCellTotal	0.084
hourlyAvailSearchFood	0.084
cellFracSpawn	0
cellDistToHide	400

age	0
fishLength	5.58703
fishWeight	2.08939
fishCondition	1
prevLength	5.54796
prevWeight	2.04615
prevCondition	1
netEnergyForBestCell	255.115
dailyDriftFoodIntake	0.100456
dailyDriftNetEnergy	120.996
dailySearchFoodIntake	0.16457
dailySearchNetEnergy	255.115
feedTime	16.283
potentialHourlyDriftIntake	0.00616936
potentialHourlySearchIntake	0.0101069
cMax	0.604691
standardResp	120.797
activeResp	35.5139
feedStrategy	SEARCH
inShelter	NO
deadOrAlive	ALIVE
deathCausedBy	<NULL>

FIGURE 1. Example Swarm animation window and probe displays, from a stream fish model. The habitat cells are shaded by depth, with deeper cells being lighter. Fish appear as horizontal line segments, with segment length proportional to fish size. The probe displays provide access to the state variables and code methods of organisms (in this case, a cutthroat trout) and habitat units (a stream habitat cell).

object. These windows can be used, for example, to continually display the simulation date, abundance and average characteristics of a population and habitat conditions. The windows also allow users to change an object's variable values during a simulation and to manually execute parts of an object's code.

- An animation window that can show the location of each individual in an IBM, overlaying a display of habitat conditions.
- The ability to click on the animation window and open a probe display to objects shown on it. A mouse click can be used, for example, to open windows showing the state of individual organisms and habitat units (Figure 1).
- Line graphs and histograms that can be attached to collections of model objects to report data such as population size, animal size distributions and habitat conditions, as the model executes.

4.C General recommendations for software implementation of IBMs. The following are practices that we find helpful in ensuring that software faithfully implements an IBM and promotes understanding of, and learning from, the model.

- Build a constituency (organization of model components and objects) before starting to write code. Having the model organization in hand helps make coding efficient and clean.
- Build verification and validation mechanisms and the necessary observability into the model from the beginning. These tools will help find mistakes early and help guide the design and implementation of additional parts of the model.
- Make extensive effort to understand how the model is executing. Use tools such as code profilers (which report how much processor time is spent on each code statement), and trace how often model objects have their methods called by which other objects.
- Design the code so it resembles the system being modeled, an important way to promote understanding and avoid subtle errors. Use additional metaphors to relate the code to real systems. Thinking about the code as if one were inside the simulation helps find subtle mistakes and create good designs.

5. Conclusions. An unfortunately large number of IBM projects has produced little scientific understanding, and software inadequacies have been a major contributor to this lack of productivity (Grimm et al. [1999]; Lorek and Sonnenschein [1999]). This situation is not surprising, considering (1) the newness of the IBM technology, (2) the lack of software engineering expertise and training among the natural resource scientists that typically undertake IBMs and (3) the fact that software design is much more important and challenging for IBMs than it is for conventional ecological and natural resource models. Software practices traditionally used for conventional models are not likely to succeed with IBMs.

We hope that the primary conclusion readers draw from this paper is that software engineering is a very important consideration in developing IBMs and using them for research and resource management. Compared to conventional models, the results produced by an IBM are much more closely linked to its software implementation. This characteristic, and the unique complexity of IBM simulations, demand a much greater engineering effort to assure that errors are eliminated and that tools are provided to make IBMs observable and understandable.

Several important strategies can make the software engineering for an IBM project manageable. First is to recognize the importance of the software process and to provide the resources for it. Whereas a conventional ecological modeling project may spend little of its budget on software, an IBM project should budget a much greater amount for software design and verification. Planning and budgeting for adequate software implementation will make an IBM project much more likely to succeed and therefore much more cost-effective.

Second is to involve software professionals in the project, instead of expecting modelers to handle the software themselves. All the issues raised in this paper are widely understood among computer scientists and software engineers, and using such professionals frees the modeler to concentrate on designing the IBM and conducting science with it (which still requires being thoroughly familiar with the software). The traditional practice of scientist writing both a model and its software has a poor track record with IBMs; the consequence too often has been the scientist's spending too much time writing inadequate software and therefore having neither time nor tools to do science with the IBM (Minar et al. [1996]).

The third strategy is to use appropriate software tools to implement a model. Swarm is one of a number of platforms that can reduce coding and documentation effort, reduce errors and provide the tools needed to observe and understand an IBM. The benefits of an active user community are not to be overlooked.

Our conclusions regarding the importance of software engineering for IBM projects are important not just for those conducting IBM projects but also for research managers and ecology instructors. Managers in charge of funding IBM-based research should expect to see significant resources dedicated to software and should encourage researchers to take software considerations seriously. A proposal to build and use an IBM that mentions little about the software issues and techniques mentioned in this paper is not likely to be a good investment. Finally, the (as yet, few) instructors teaching students how to do individual-based ecology and resource management need to instill in their students an understanding of the relation between an IBM and its software.

Acknowledgments. These thoughts were originally presented by the first author to the special symposium, *Advancing the Individual-*

Based Modeling Approach: *New Tools and Concepts*, at the 2000 Ecological Society of America annual meeting. Funding for the authors' participation in the conference was provided by Columbia Basin Research, University of Washington.

REFERENCES

V. Grimm [1999], *Ten Years of Individual-Based Modelling in Ecology*: *What Have We Learned and What Could We Learn in the Future?* Ecological Modelling **115**, 129–148.

V. Grimm [2002], *Visual Debugging*: *A Way of Analyzing, Understanding, and Communicating Bottom-Up Simulation Models in Ecology*, Natur. Resource Modeling **15**, 23–38.

V. Grimm, T. Wyszomirski, D. Aikman and J. Uchmanski [1999], *Individual-Based Modelling and Ecological Theory*: *Synthesis of a Workshop*, Ecological Modelling **115**, 275–282.

H. Lorek and M. Sonnenschein [1999], *Modelling and Simulation Software to Support Individual-Based Ecological Modelling*, Ecological Modelling **115**, 199–216.

N. Minar, R. Burkhart, C. Langton and M. Askenazi [1996], *The Swarm Simulation System*: *A Toolkit for Building Multi-Agent Simulations*, Working Paper 96-06-042, Santa Fe Institute, Santa Fe, NM. Available at `http://www.santafe.edu/sfi/indexPublications.html`.

S.F. Railsback [2001], *Concepts from Complex Adaptive Systems as a Framework for Individual-Based Modelling*, Ecological Modelling **139**, 47–62.

W. Wilson [2000], *Simulating Ecological and Evolutionary Systems in C*, Cambridge Univ. Press, Cambridge, England.