FOREST: A System for Developing and Evaluating Ecosystem Simulation Models

Heather K. May*
John S. Conery
Department of Computer and Information Science
University of Oregon
hmay@fs.fed.us

Keywords: ecology, individual-based, design pattern, Java.

Abstract

FOREST is a software environment for designing, implementing, and evaluating ecosystem simulations. Written in Java, it encourages modelers to use an object-oriented design to promote extensibility and model reuse. The paper introduces a new design pattern, called the Event Handler Pattern, that allows modelers to extend an existing class hierarchy without modifying existing code.

INTRODUCTION

Computational models have the potential of becoming valuable tools for environmental managers to evaluate alternative management strategies for an ecosystem. Models provide a way to perform controlled experiments that would be difficult or impossible to conduct in nature. As a result, mathematical models have contributed to the development of some central theories in ecology, including competition theory and the paradox of enrichment [5].

Traditional computational models have used a high level of aggregation, where the state of the system is defined by the collective properties of the populations [2]. Since many details about individuals are abstracted away, these models achieve a high level of generality. Aggregated models try to capture the essential dynamics that account for the observed behavior of a system, without identifying which details are directly responsible for that behavior. Specifically, aggregated models have been used to predict the future state of a system in the absence of certain sources of variation, including demographic stochasticity and a spatially or temporally heterogeneous environment.

Predictions from aggregated models are often not applicable to a specific ecosystem since natural populations rarely satisfy the conditions assumed in these models. In contrast, individual-based models (IBM) explicitly represent each individual in a population, allowing for the inclusion of variation between individuals due to genetic and environmental influences [13]. Since individual-based models do not abstract away essential differences among individuals, they can potentially provide a more accurate representation of a particular ecosystem. Such models can also help ecologists understand how collective properties of a population or community emerge from individual interactions.

Despite the potential benefits, there are several biological and computational challenges to applying an individual-based approach to ecosystem modeling. Biological challenges include quantifying traits and behaviors, with a corresponding increase in the amount of information needed to formulate and parameterize individual-based models [12] [13]. Modeling requires knowledge about the characteristics of each individual and the factors affecting its behavior, which may not be possible to collect through observation. The computational difficulties stem from the high level of detail and numerous interactions included in the representation of individual-based models: efficient methods for storage and retrieval of large quantities of data, multiple techniques for analysis and communication of results, and minimizing the execution time [8].

An ecosystem modeling environment could facilitate the widespread development of ecological models by helping researchers manage the significant challenges in the formulation and implementation of this type of model. Such a system would promote communication and collaboration between model developers. Our project, FOREST (Forest Object-oriented Reusable Ecosystem Simulation Template), was designed to be the foundation for such an environment, providing tools for defining classes of individuals, infrastructure for building IBM simulations, and graphic visualizations for interpreting and analyzing results. We used concepts from object-oriented programming — class hierarchies and design patterns — to maximize reuse of model components and to allow model developers to focus on the biological rather than computational challenges.

This paper begins by addressing the challenges to implementing efficient IBM ecological simulations, and then outlines the necessary components of an ecological modeling environment. We then describe the structure of FOREST and its implementation in Java, followed by the results from a test simulation.

---

* Current address: USDA Forest Service Dorena Genetic Resource Center, Cottage Grove, OR.

CHALLENGES FOR INDIVIDUAL-BASED ECOSYSTEM MODELS

The computational challenges in individual-based ecological models can be attributed to the higher level of detail than in traditional aggregated models and a large number of interactions between model components. Detailed individual-based models generate very large data sets based on representations of many thousands of individuals, leading to more complexity in the analysis and communication of results, and a much higher overall difficulty involved in the formulation, implementation and calibration of the models. Although the calculations used to determine the effects of interactions between individuals tend to be relatively simple, the number of interactions in a system can be quite large [14], so the infrastructure that supports these models must be efficient, scalable, and potentially parallelizable.

One of the first challenges that needs to be faced is how to set the initial attributes of each individual in the population. Two methods for initialization are to use data collected about each individual or to use statistical distributions based on aggregated field data. To parameterize the model from actual observations requires an extensive amount of information to be collected about every individual represented in the simulation [16]. However, it may not be possible to collect all of the data needed to fully populate the model due to biological difficulties in making the observations and the amount of time required for data collection. Some attributes may have to be estimated using randomly generated variates from statistical distributions if collecting the necessary information about each individual is not feasible. For example, in simulations of a specific wildlife reserve, the exact location of every individual at the beginning of the simulation may not be known from field data. The initialization routine must provide a method to disperse individuals throughout the model environment in patterns that match their natural dispersal patterns.

The large number of interacting factors makes the results from individual-based models more difficult to analyze and interpret than results from state variable models [4][5]. Also, since most individual-based models include some stochasticity, the simulation needs to be run multiple times with the same parameters and initial conditions. Statistical analysis becomes necessary in order to assess the probabilities of various outcomes resulting from each parameter set and to determine the robustness of the model in terms of its sensitivity to initial conditions. Second, due to the large number of variables and the possible estimation of parameters, there is no way to test the validity of a highly detailed ecological model [17]. Instead, these models must be evaluated in terms of their level of corroboration, which is a measure of how well a model meets its objectives. The level of corroboration is determined through the comparison of model output with field data over the range of conditions for which the model is to be used [17].

Due to the large data sets and number of simulations required, effective communication of the results often requires graphical displays. Graphical visualization of the data can facilitate understanding of complex relationships, but only if the data are presented in a such a way that patterns are easily recognizable to the human eye. Recently there has been an increased awareness of the crucial role that visualization plays in the analysis and interpretation of results from ecological simulations [8].

Resulting from the inherent complexity of ecological systems, an important step in building ecosystem models is determining which details to include. As a general guideline, a model should include only those aspects of the system that affect the problem under investigation, but should contain enough detail to permit valid conclusions to be drawn about the real system [4]. However, no complete methodology exists for determining which details in a complex ecological system control particular aspects of the larger system [9].

Levin et al. [12] warn against creating models which include more detail than can be measured or parameterized. The output from such models may appear realistic, but does not represent any real system. Many modelers suggest developing simple individual-based models with as few parameters and assumptions as possible [5]. Additional details can be added and removed in an attempt to identify which local interactions affect the broader scale patterns and which are noise [12]. This approach requires extensive simulation programs that allow users to vary the parameters and level of detail in the model. However, this approach is necessary to improve the realism of the representation of the system and the corresponding accuracy of the model's predictions.

The absence of the widespread development of individual-based ecosystem models as tools for environmental management can be attributed both to the computational complexities associated with building and analyzing these models and to the lack of communication and collaboration between modelers. Development has been limited by the ability of any single team of researchers to deal with the conceptual complexities involved in the formulation, implementation, calibration and debugging of ecological models. Some models are so complicated that they are comprehensible only to the developers, making it virtually impossible to communicate the structure of the model to others. The inability to explain the model structure has prevented the use of these models as decision making tools for ecosystem management, since policy makers (and the public) are unlikely to trust a model they do not understand. Furthermore, these models have typically taken teams of experts from two to five years to develop, which may be too long to wait to make a management decision. If the goal is endangered species

protection, creating a model to test management strategies is impractical since the population may be irreparably harmed in the time it takes to develop the model. Additionally, the lack of communication and collaboration between modelers has resulted in different groups of researchers encountering similar challenges in the design and implementation of ecological models. The similarities between ecosystems and hierarchical nature of ecosystems suggest the possibility of creating models with reusable and extendable components, to reduce the amount of redundancy in model creation.

The FOREST modeling environment introduced in the next section was designed specifically to address the issues listed above that are important for ecosystem modeling. FOREST is essentially an integrated development environment (IDE) designed specifically for developing and evaluating individual-based ecological simulations. The major components of FOREST are:

- Graphical interface: FOREST presents users with a window containing several different panels. Some of these allow the user to enter simulation parameters, control execution, and interact with the model. Output panels display simulation results graphically; for example a 2D plot can show the location of individuals in a spatial model. Screen shots of the FOREST interface are shown in figures later in this paper.
- Stochasticity: FOREST has a set of random number generators to support a variety of distributions, including Gaussian, Binomial, and Poisson distributions.
- Initialization: Both methods described earlier for creating the initial population are supported: users can load descriptions of individuals from a file, or use the random number generators to assign values for selected attributes in each individual.
- Object-oriented models: In order to promote better communication between modelers, FOREST adopts an object-oriented approach that will enable programmers to reuse code developed for one simulation and adapt it for other models. Our approach is the one commonly used in object-oriented programming: we provide a set of abstract base classes for important elements of the simulation, and guidelines for how to implement concrete objects built upon these base classes. We believe the inherent modularity and hierarchical nature of object-oriented models will also help modelers better explain the models they produce when they present the design and results to non-specialists.

FOREST

FOREST is a discrete-event ecological simulation environment written in Java. By adopting an object-oriented framework we expect models created in this model development environment to be more easily designed and then adapted and extended for other models (e.g. [7]). FOREST is based on the premise that a fundamental step in the process of building an ecological model is determining which details to include. Therefore, FOREST's class hierarchy was designed to allow the model developer to determine the entities and events in the model, as well as the implementation of each event.

The discrete event simulator based at the core of the FOREST system uses several of Java's predefined classes and interfaces. The event list is implemented as a `TreeMap` class, and events are ordered with `GregorianCalendar`. To represent events, we defined a base `Event` class and `TimeDependentEvent` interface.

The following classes are provided as base classes from which more specific classes can be derived for each simulation model:

- `LivingOrganism`: This is the abstract base class for all living entities in the model. The attributes in this class are the individual's birthdate and species.
- `Species`: This class is just a data structure to contain all of the species-specific information for `LivingOrganism` objects.
- `Population`: Objects in this class contain a collection of `LivingOrganism` objects and methods that return aggregate information about the individuals (e.g. total size, average age).
- `Environment`: The environment is implemented as a two-dimensional array of cells. In the current version each cell can hold at most one organism at a time.

In order to assist the modeler in identifying the appropriate level of detail, FOREST's model hierarchy was designed to allow the model developer to change the implementation of certain events and examine the effects of such changes.

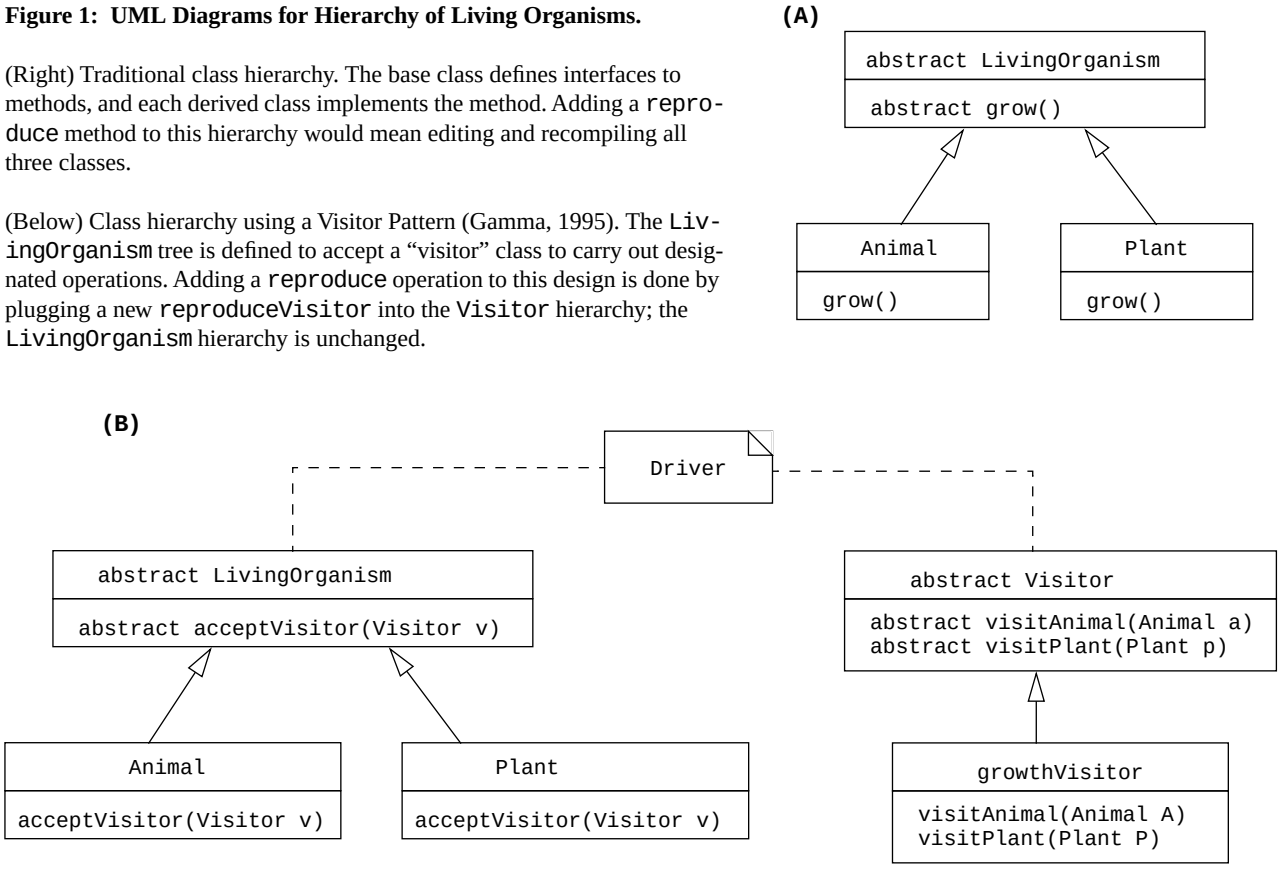
In a traditional object-oriented hierarchy, adding new functionality to a base class often requires adding a new method to every class below it in the hierarchy. Figure 1 shows a simple example of a class hierarchy for organisms, where `grow` is a method defined in the base class and implemented in each derived class (Figure 1A). In order to add reproduction as a behavior of every `LivingOrganism` object, the base class would be extended and then derived classes would have to implement a new `reproduce` method. This would require modifying and recompiling several classes in the hierarchy, a time-consuming process in large systems.

Gamma et al [10] described an object-oriented design pattern, known as the *Visitor* pattern, which was created to allow additional functionality to be added to a system at a later date without the need to modify the existing components of the system. This is accomplished by implementing additional methods outside of the original class hierarchy, in

Figure 1: UML Diagrams for Hierarchy of Living Organisms.

(Right) Traditional class hierarchy. The base class defines interfaces to methods, and each derived class implements the method. Adding a `reproduce` method to this hierarchy would mean editing and recompiling all three classes.

(Below) Class hierarchy using a Visitor Pattern (Gamma, 1995). The `LivingOrganism` tree is defined to accept a "visitor" class to carry out designated operations. Adding a `reproduce` operation to this design is done by plugging a new `reproduceVisitor` into the `Visitor` hierarchy; the `LivingOrganism` hierarchy is unchanged.



`Visitor` classes. The Visitor pattern requires implementing two class hierarchies: one for the elements in the system and one for the visitors that define operations on the elements (Figure 1B). New operations can be added by creating a new subclass in the `Visitor` hierarchy.

Visitor patterns use a technique known as double-dispatch, where the method that gets invoked depends on the kind of request and types of receivers. All classes in the hierarchy are prepared to accept a `Visitor` object via an `accept` method. To process an event, the simulation program ("driver") first matches the event with an `accept` method in the organism that will be updated. The second dispatch occurs when the organism object invokes the visitor, the object that actually does the state updates. The object being visited passes itself as a parameter. Invoking this method gives the visitor a reference to the visited object and access to all of its public methods. Inside the `Visitor` classes, there is a `visit` method for each object in the hierarchy.

The value of this organization can be seen when it is time to extend the simulation to include a new behavior. In the example shown in Figure 1(B), adding a "reproduce" behavior to the objects can be accomplished by simply creating a new `Visitor` subclass called `reproduceVisitor`. Inside this new class, `visitAnimal` and `visitPlant` methods would use public "getter" and "setter" methods of the respective `Animal` and `Plant` classes to implement the reproduction operations. The class hierarchy for the original `LivingOrganism` tree would not need to be modified or recompiled.

Although Visitor patterns simplify the addition of new functionality to existing classes, there are two drawbacks to using this approach. The first stems from the fact that operations defined by visitors are implemented outside of the object. Classes that provide enough information for visitors often expose more of their internal state, which might compromise encapsulation [10]. Additionally, Visitor patterns actually complicate the addition of new classes into the hierarchy since each visitor class must be modified to visit the new class. For example, if we wanted to add the class `Fungus` to the hierarchy, a `visitFungus` method would have to be added to every visitor class. Even though this example has only one concrete visitor class, there may be many visitors in an actual implementation. For this reason, Visitor patterns have been recommended only for systems whose class hierarchies are unlikely to change.

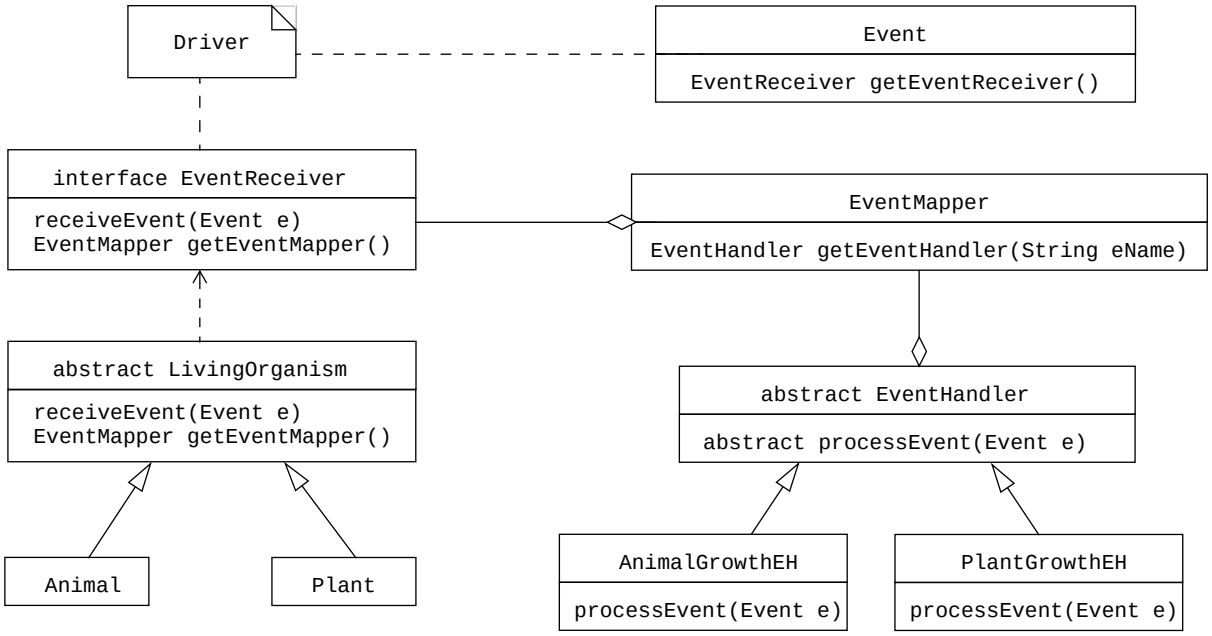


Figure 2: LivingOrganism classes implemented with the EventHandler design pattern.

The appeal of the Visitor pattern for ecological modeling is in the fact that the methods of an object can be added or removed without modifying the model class hierarchy. The implementation of a method can also be easily redefined by removing the current visitor and creating a new visitor for this function. However, two consequences of using the Visitor pattern conflict with the goals of FOREST. The requirement of a static class hierarchy is too strict for an environment designed to facilitate the calibration of ecological models. Modelers should be able to decide which entities to include in their models, since this will depend on the specific goals of the model and the corresponding appropriate level of detail. Additionally, the implementation of a certain behavior for all objects in the hierarchy must be placed in a single Visitor class. It should be noted that the grouping of related behaviors is usually considered one of the advantages to using Visitor patterns. However, this implementation complicates the process of changing the behavior of just one particular entity (or a subset of the entities) in the hierarchy, which may be desired in the process of calibration.

To address these problems, we developed a variation on the Visitor design pattern that will make it easier for modelers to add new classes. In this new *EventHandler* pattern, the modeler can specify which entities and behaviors are implemented in the model. Accomplishing this goal required a higher level of modularization than commonly used in object hierarchies. As a result, the entities in the model are used only to encapsulate data, rather than both methods and data. Instead of grouping the implementation of a behavior into one class, FOREST creates a separate class to implement this behavior for each object in the hierarchy. FOREST assigns an EventHandler object for each behavior an entity participates in during the simulation (Figure 2).

The `Event` class is simply a data structure to hold all information needed to process the event. The `EventReceiver` is the object the event affects. In order to accommodate interactions between organisms, events can contain multiple objects as long as one of the objects is designated as the receiver. Since `EventReceiver` is an interface, any object in the model can receive events. Each receiver must contain an `EventMapper`, which is just a hash table that maps events to their appropriate `EventHandlers`. In order to process an event, the receiver must be queried to get the appropriate handler for this particular event. The `processEvent` method in the handler class is then invoked, and the actual processing of the event takes place in the `EventHandler` class.

In FOREST, each species has its own set of `EventHandlers`, which implement all of the behaviors for individuals of that species. This implementation allows for species-specific behavior to be incorporated into the model, although it does not require a different `EventHandler` for each species.

By associating a different `EventHandler` for each species, reusable modules can be developed for the behavior of each species. A new model could be created that uses components from more than one existing model. This design facilitates reuse more than the traditional use of inheritance through class hierarchies, since a new model could reuse or extend parts of an existing model without having to extend the class hierarchy. Another advantage of this design is that it allows the user to choose how events are implemented at runtime. This means that modelers may change the implementation of a model each time it is run in order to investigate the effects of the changes on the dynamics of the system.

Although FOREST's use of the EventHandler pattern alleviates the need for a static class hierarchy in the traditional application of visitor patterns, this implementation results in some new tradeoffs. This design results in a large number of `EventHandler` classes since a new class may have to be created for each (species, behavior) combination. The methods of an object are dispersed throughout the `EventHandler` hierarchy, instead of being encapsulated within the object. Additionally, invoking the appropriate event handler for an object is slightly more complicated since the association is between EventHandler *classes* and objects rather than Visitor *methods* and objects. This require an additional step in processing an event, since the events must be mapped to their appropriate handlers by the event receiver.

The flexibility in model development comes at the expense of increased complexity in the design and implementation of the base class hierarchies. However, the process of developing and calibrating new models is simplified once the `EventReceiver` and `EventHandler` hierarchies are implemented. The modeler just needs to implement the entities (in `EventReceiver` classes) and events (in `EventHandler` classes), and define the associations between the two types of objects. Once some entities and events have been implemented and added to the library, these components may be used in new models by creating the desired associations. To facilitate the development of realistic ecological simulations, the benefit of modular development outweighs the tradeoff of increased complexity of the base class hierarchy.

EXAMPLE

To demonstrate the ability to create simulation models in FOREST, a simple model (GrowthModel) was developed to simulate tree growth. The model represents an ecosystem with three species of trees which differ in life history characteristics. The behaviors simulated in the model are growth, seed dispersal and death. Although this model was not parameterized from field data, this example shows how to initialize a FOREST model when the parameters are known or can be estimated. The model consists of entities and two types of events: state events and scheduling events. State events represent the behaviors in the model and are used to update the state of the entities, and scheduling events are used to schedule the occurrence of the state events.

The only entities in GrowthModel are the trees, implemented in the class `Tree`. The `Tree` class stores the height and diameter of the tree, and provides methods for getting and setting these values. Since `Tree` extends `LivingOrganism`, it implements `EventReceiver` and inherits the `receiveEvent`, `setLastUpdate` and `getLastUpdate` methods. The interface for the `Tree` class consists of: the following attributes and methods:

```
Attributes:
    double height;
    double diameter;
Methods:
    double getHeight();
    setHeight();
    double getDiameter();
    setDiameter();
```

The species-specific parameters needed to process the events are stored in the `Species` class. Each species contains attributes for the mean and maximum number of seeds produced during a reproductive event. The `Species` class also contains the mean and maximum values of the dispersal range of seeds, as well as the growth rate and death rate for individuals of the species:

```
Attributes:
    double meanOffspring;
    double maxOffspring;
    double meanDispersalDistance;
    double maxDispersalDistance;
    double growthRate;
    double deathRate;
Methods:
    double getMeanOffspring();
    double getMaxOffspring();
    double getMeanDispersalDistance();
    double getMaxDispersalDistance();
    double getGrowthRate();
    double getDeathRate();
```

Since the state events correspond to the behaviors in the model, the state events in this model are seed dispersal, growth and death.

When a dispersal event occurs, the number of seeds to disperse is determined by generating a random number from a binomial distribution, using the appropriate parameters for this species. For each seed, the dispersal distance is determined by generating a random number from another binomial distribution. A uniform random variate is then used to determine the direction of dispersal. The seed will survive only if this location is not currently occupied by another tree.

GrowthModel provides two event handlers with different implementations of tree growth to show that the implementation of an event can be easily redefined. The first implementation calculates growth using a linear equation with a species-specific slope. The second implementation takes shading into account by decreasing growth for trees that have neighbors in adjacent locations. This version is implemented by querying the environment for the number of trees surrounding a certain tree, and reducing the tree's growth by 5% for each adjacent tree. The user may choose which implementation is applied in the simulation when the model is initialized.

Since growth is a continuous process, the class implements `TimeDependentEvent`. Since the size of all of the individuals in a population should be updated at the same time, `Growth` events are scheduled for the entire population. The event handler first iterates through the population and updates each individual's size, and then schedules the next `Growth` event for this population for the current time plus the time interval between updates. If the time of the next occurrence is outside the growth period for this species, the event is not added to the event list.

Growth events should be scheduled at fixed time intervals throughout the growth period, seed dispersal events need to be scheduled once a year, and death should be scheduled once for each tree. Since GrowthModel schedules events based on statistical properties of the population, these events are scheduled for the entire population. All of these events implement `InitialEvent`, and are used to initialize the event list at the start of the simulation.

Dispersal events are scheduled for each population at the beginning of that species' reproductive time of year. This event handler iterates through all of the reproducing individuals in the population and schedules a seed dispersal event for each one. The time of dispersal for each individual in the population is determined by generating a random variate from an exponential distribution. After scheduling the dispersal events for all of the individuals, the next occurrence of this event is scheduled for the beginning of the reproductive season of the next year.

Since `Growth` events are self-generating (i.e., the next occurrence of the event is scheduled during the processing of the event), this class only needs to schedule the occurrence of the first event of the growth season.

The model is initialized from a data file, which creates the populations and sets the initial values for all of the attributes of the trees. The model uses the random dispersal method provided by FOREST to disperse the individuals in all of the populations throughout the model environment.

The output of the simulation consists of information about the populations (e.g. total size vs. time, average height) and about the simulation itself (e.g., current time, number of events processed). The position of each individual tree within the environment is displayed on a two-dimensional grid (Figure 3).

At the beginning of the simulation, no events have been processed so most of the values shown on the screen are zero. After 100 days of simulated time, the number of trees in each population has changed, with an overall net gain in the number of trees. The size of each population over time can be shown graphically (as in Figure 4), or the second plot can be replaced by a table (not shown) if the user wants to see exact values.

Even though the implementations of the behaviors in GrowthModel is fairly simple, this model shows that users can define the level of detail in their simulations. This model uses few details on the individual level, but the amount of detail can be increased by extending the Tree class and adding more attributes and by implementing additional `Event` and `EventHandler` classes.

SUMMARY AND FUTURE WORK

FOREST takes an object-oriented approach to ecosystem modeling. Model entities are defined according to a class hierarchy, where classes lower in the hierarchy can inherit attributes of previously defined classes. By itself this organization should help make it easier for modelers to create new models by reusing components of earlier models. Another aspect of the FOREST design that will promote modularity and reuse is the use of a new EventHandler design pattern that makes it easier to adapt existing models without having to modify current class hierarchies.

FOREST was designed to accommodate users with different amounts of programming experience. In addition to the abstract class hierarchy, FOREST could promote reuse of model components by providing a library of concrete implementations of the entities and events commonly included in a forest simulation. Therefore, users with minimal programming experience could create models with the existing components, while experienced programmers could extend these components to create more detailed models. The library would be a collaborative effort, where programmers could add the components they implemented so that other modelers may use these components in their models.

Since FOREST's main goal is to assist the modeler in identifying the appropriate level of detail, the focus was on building the simulation engine and a class hierarchy which would support the development of reusable model components. Much less emphasis was placed on developing techniques for initialization from user input, analysis and visualization of the results, and communication between entities in the model. As a result, FOREST would need to be extended to include these features to provide a complete environment for developing ecological simulations.

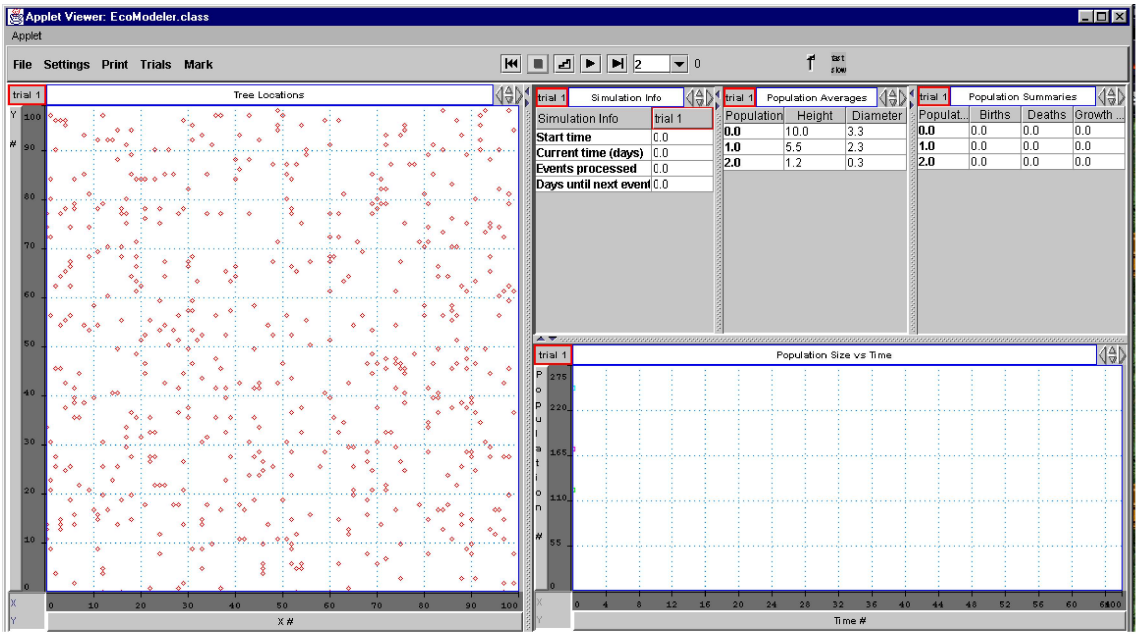
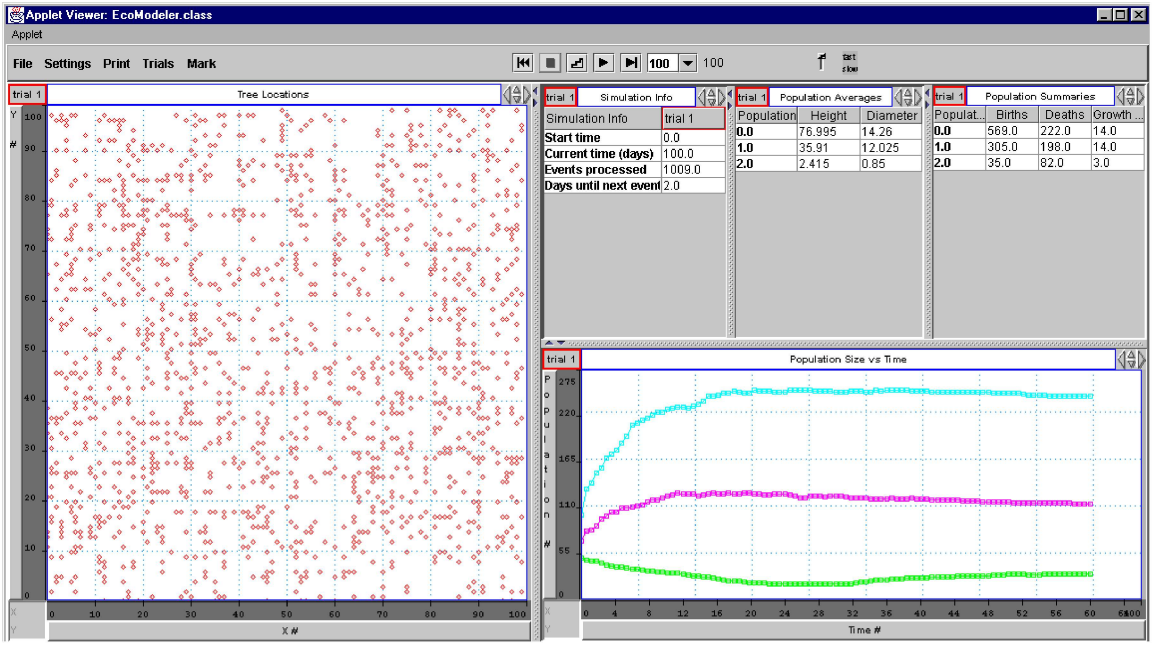


Figure 3: Initial model state (above). Trees have been set at random locations, as shown in the scatter plot on the left. The tables in the upper right show simulator parameters.

Figure 4: Growth model after 100 days (below). The chart on the lower right tracks the size of each population over time. Since the total number of trees has increased, the scatter plot is more dense than it was in the original state.



FOREST was able to reduce the time needed to implement an individual-based forest simulation (GrowthModel), by providing the discrete-event simulation components, extending these components to facilitate the development of ecological models, and implementing a model class hierarchy that supports modular development. The development time was reduced since the model only had to implement the entities and their behaviors, rather than an entire simulation and class hierarchy from scratch. However, the full benefits of using the EventHandler implementation would only be realized once a library of component modules had been developed, and additional techniques implemented for initialization of the model and analysis and visualization of the results. A new model could be created that uses or extends components from more than one existing model, decreasing the time needed for formulation and implementation of the model. Additionally, modelers could investigate the issue of relevant detail by varying the type and implementation of the events in the simulation and examining the effects of the alterations on the simulation results.

Two limitations in the current design of FOREST are that it does not provide any support for modeling interactions between organisms and it does not support movement of organisms between cells. Simple interactions, for example two species who compete for the same food source, could be mediated by the environment. But direct interactions, such as those in predator-prey models, would require an extension to the modeling framework to allow interactions between simulated objects. Allowing objects to interact introduces several interesting challenges, including methods for handling conditional events [3].

Another avenue for future development includes support for parallel execution. Parallelism within a model, especially a spatially explicit model, is a very challenging prospect [6][11]. But in stochastic models, where the same system is simulated many different times, a simple form of parallelism would be to run several instances of the model at the same time. This sort of parallel execution would be very straightforward using the underlying support for parallel threads in Java.

ACKNOWLEDGEMENTS

Much of the FOREST modeling environment was constructed with the help of software and visualizations developed by Tom Conlin and Mark Felt under the direction of Prof. Daniel Udovic at the University of Oregon [18].

REFERENCES

[1] Banks, J., J.S. Carson II, B.L. Nelson, D.M. Nicol. 2001. *Discrete-Event System Simulation.* New Jersey: Prentice-Hall, Inc.

[2] Begon, M., J.L. Harper, C.R. Townsend. 1996. *Ecology.* Oxford: Blackwell Science Ltd.

[3] Bolte, J.P., J.A. Fisher, D.H. Ernst, 1993. "An Object-oriented, Message-based Environment for Integrating Continuous, Event-driven and Knowledge-based simulation." In *Application of Advanced Information Technologies: Effective Management of Natural Resources.* ASAE. June 18-19, Spokane, WA.

[4] Caswell, H., and A.M. John. 1992. "From the Individual to the Population in Demographic Models." In *Individual-Based Models and Approaches in Ecology,* eds. D.L. DeAngelis and L.J. Gross. New York: Chapman and Hall.

[5] DeAngelis, D.L., and K.A. Rose. 1992. "Which Individual-Based Approach is Most Appropriate For a Given Problem?" In *Individual-Based Models and Approaches in Ecology,* eds. D.L. DeAngelis and L.J. Gross. New York: Chapman and Hall.

[6] Deelman, E., Caraco, T., and Szymanski, B. K. "Simulating Lyme Disease Using Parallel Discrete Event Simulation." In *Proc. Winter Simulation Conference*, 1996.

[7] Downing, K. "An Object-oriented Migration Model Driven by Biological Field Data." In *Proc. Pacific Symp. Biocomputing*, 1996.

[8] Deutschman, D.A., C. Devine, L.A. Buttel. 2000. "The Role of Visualization in Understanding a Complex Forest Simulation Model." In *ACM SigGraph 34*(1).

[9] Deutschman, D.A., SA. Levin, C. Devine, L.A. Buttel. 1997. "Scaling from Trees to Forests: Analysis of a Complex Simulation Model." *Science On-Line* (http://www.sciencemag.org/feature/data/deutschman).

[10] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[11] Glass, K., Livingston, M., and Conery, J.S. "Distributed Simulation of Spatially Explicit Ecological Models." In *Proc. 11th Workshop on Parallel and Distributed Simulation*, 1997.

[12] Levin, S., B. Grenfell, A. Hastings, and A.S. Perelson. 1997. "Mathematical and Computational Challenges in Population Biology and Ecosystems Science." *Science 275*: 334-341.

[13] Lomnicki, Adam. 1992. "Population Ecology from the Individual Perspective." In *Individual-Based Models and Approaches in Ecology,* eds. D.L. DeAngelis and L.J. Gross. New York: Chapman and Hall.

[14] Matsinos, Y.G., W.F. Wolff, and D.L. DeAngelis. 2000. "Can Individual-Based Models Yield a Better Assessment of Population Variability?" In *Quantitative Methods for Conservation Biology,* eds. S. Ferson and M. Burgman. New York: Springer.

[15] Maxwell, T., F. Villa, R. Costanza. *Spatial Modeling Environment.* http://www.uvm.edu/giee/SME3.

[16] Slothower, R.L., P. Schwarz, and K.M. Johnston. 1996. *Some Guidelines For Implementing Spatially Explicit, Individual-Based Ecological Models Within Location-Based Raster GIS.* http://www.sbg.ac.at/geo/idrisi/gis_environmental_modeling/sf_papers/slothower_roger/sf23.html

[17] Swartzman, G.L., and S.P. Kaluzny. 1987. *Ecological Simulation Primer.* New York: Macmillan Publishing Co.

[18] Udovic, D., T. Conlin and M. Felt. 2001. *Java Demography 2.1. Computer software and manual.* Biology Software Lab web site (http://darkwing.uoregon.edu/~bsl/). In Jungck, J., Peterson, N. S., and Calley, J. (eds.) *BioQUEST: Quality Undergraduate Educational Simulations and Tools VI*, Academic Press (CD-ROM).