Domain-based Reengineering of a Spatially Explicit Individual-based Population Modeling and Simulation Research Tool

by

Sachen Mary Margaret Macdonald
B.Sc., University of Victoria, 1999

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. M. Storey, Supervisor (Department of Computer Science)

[redacted]

Dr. H. Muller, Departmental Member (Department of Computer Science)

[redacted]

Dr. I. Traore, Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. P. Keller, External Examiner (Department of Geography)

Supervisor: Dr. Margaret-Anne Storey

ABSTRACT

This study is based on the reengineering of a research tool for spatially explicit individual-based population modeling and simulation. Reengineering is typically based on various program analysis methods such as parsing and data flow analysis of the existing code base. While these methods can provide some insight as to the functionality of the program, they rarely capture the intention or purpose of the program. This is particularly true with respect to programs designed for research since they generally evolve to accommodate new research ideas and the intention or purpose of the program is obscured. Even though this domain is fairly well known and there are many examples of similar applications, there lacks clear and prescriptive guidance on how a program in this class should be designed. In this study, we used a domain engineering methodology to understand common requirements in this domain and an object-oriented framework to document that understanding. This framework is then used to guide the reengineering process.

Examiners:

[redacted]

Dr. M. Storey, Supervisor (Department of Computer Science)

[redacted]

Dr. H. Muller, Departmental Member (Department of Computer Science)

[redacted]

Dr. I. Traoré, Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. P. Keller, External Examiner (Department of Geography)

Table of Contents

Table of Contents iii
List of Tables vi
List of Figures vii
Acknowledgments ix
Chapter 1 Introduction 1
1.1 Motivation 1
1.2 Problem 1
1.3 Approach 2
1.4 Solution 3
1.5 Thesis Outline 4
Chapter 2 Background on Target System 5
2.1 Reengineering Problem 6
2.2 Overview of the Architecture 8
2.2.1 Architectural Characteristics 10
2.3 Overview of the Code 12
2.3.1 Code Structure 13
2.4 Summary 15
Chapter 3 Background on Domain-based Reengineering 16
3.1 Reverse Engineering 17
3.1.1 Reverse Engineering Tools 17
3.1.2 Manual Strategy 19
3.2 Domain Engineering 20
3.3 Object-oriented Framework Design 22
3.3.1 Framework Development vs. Traditional Object-oriented Design 23
3.3.2 Black Box Frameworks vs. White Box Frameworks 24
3.3.3 Class Libraries vs. Frameworks 25
3.3.4 Polymorphism and Dynamic Binding in Framework Design 25
3.4 Summary 26
Chapter 4 Domain Analysis 28

4.1 General Ecosystem Modeling and Simulation 29
4.1.1 ATLSS 29
4.1.2 OO-IDLAMS 30
4.2 Individual-based Population Modeling and Simulation 31
4.2.1 ECLPSS 32
4.2.2 ECOSIM 34
4.3 Domain Characteristics 36
4.3.1 Modeling the Individuals 37
4.3.2 Modeling the Environment 37
4.3.3 Modeling the Relationships 38
4.3.4 Modeling the Processes 40
4.3.5 Scheduling of Processes 42
4.4 Summary 43
Chapter 5 A Framework to Support Spatially Explicit Individual-based Population Modeling and Simulation 46
5.1 Domain Requirements 46
5.2 Framework Description 49
5.2.1 Overview 50
5.2.2 SimulationManager Class 51
5.2.3 SimulationScheduler Class 52
5.2.4 SimulationObject Class 53
5.2.5 Grid Class 53
5.2.6 Cell Class 54
5.2.7 Individual Class 54
5.2.8 Condition Class 55
5.3 Framework Extension 55
5.3.1 Design Considerations in the SimulationManager 55
5.3.2 Design Considerations in the SimulationScheduler 58
5.3.3 Creating and Extending SimulationProcess Classes 59
5.3.4 Creating and Extending SimulationObject Classes 60
5.4 Summary 61

Chapter 6 Framework Instantiation 62
6.1 Framework Instantiation 63
6.1.1 SimulationObject Classes 64
6.1.2 SimulationProcess Classes 69
6.1.3 SimulationScheduler Class 73
6.1.4 SimulationManager Class 74
6.2 Summary 77
Chapter 7 An Agent-based Approach 79
7.1 Agent-based Modeling and Simulation 80
7.2 CORMAS Framework 81
7.2.1 Building an agent-based model with CORMAS 83
7.3 Research in an Agent-based Environment 85
7.3.1 Sustainable Resource Management Aspects 87
7.3.2 Prototype Architecture 88
7.4 Summary 93
Chapter 8 Conclusions 94
8.1 Theoretical Implications 94
8.1.1 Benefits of Simple Manual Reverse Engineering Techniques 95
8.1.2 Domain Analysis to Support Reverse Engineering 96
8.1.3 Domain Engineering to Support Reengineering Activities 96
8.1.4 Frameworks to Capture and Discuss Domain Assets 97
8.1.5 Applicability of Agent-based Technology in this Domain 97
8.2 Practical Implications 98
8.2.1 The SORTIE Reengineering Project 98
8.2.2 Future Projects in the Same Domain 99
8.3 Limitations 100
8.4 Future Work 101
Bibliography 102
Appendix A SORTIE Instantiation Supplemental 105

List of Tables

5.1 Summary of Domain Requirements.....47

List of Figures

2.1 Screen Shot of SORTIE..... 5

2.2 User Tasks in SORTIE..... 9

2.3 SORTIE Model..... 10

2.4 Pipe-and-Filter Style Data Flow..... 11

3.1 Reverse Engineering Process..... 19

3.2 Domain Engineering..... 21

3.3 Framework Development versus Traditional Object-oriented Design... 23

3.4 The Shape Class Hierarchy..... 26

4.1 DIAS Framework..... 31

4.2 Overview of the ECLPSS Environment..... 33

4.3 EcoSim Class Hierarchy..... 35

4.4 Group Relationship..... 39

4.5 ProximityRelationship..... 39

5.1 Framework Class Diagram..... 50

5.2 Sequence Diagram of Main Tasks in the SimulationManager..... 51

5.3 Diagram of SimulationScheduler..... 52

5.4 SimulationProcess Targeting Individuals via Grid Object..... 59

6.1 SimulationObject Class Hierarchy..... 64

6.2 BorderedCell Relationships..... 66

6.3 SimulationProcess Class Hierarchy..... 70

6.4 Process Coordination Example..... 71

6.5 SimulationManager States..... 74

7.1 CORMAS Class Hierarchy..... 84

7.2 Consensus in the Harvest Process Agent..... 88

7.3 Integration of Agents with Framework..... 89

7.4 Sequence Diagram of Harvest Process..... 92

Acknowledgments

I am grateful to the people at the Institute of Ecosystem Studies for their hospitality during site visits and patience during this study, especially Charles Canham and Dave Coates for giving us the opportunity to collaborate. I could not have finished this thesis without the support and advice of my supervisor Peggy Storey and my colleagues at the University of Victoria. In particular I would like to thank my friends and family. It is with your love and support that I can get through anything.

Chapter 1

Introduction

1.1 Motivation

This study is based on a legacy research tool for spatially explicit individual-based population modeling and simulation called SORTIE [21]. Population modeling and simulation tools are used to model population dynamics in simulated ecosystem environments. These tools are primarily used for predictive experimentation by biologists and resource managers. SORTIE was built to support scientific research in forest dynamics. It has been in use since 1992 and has evolved to support new research ideas that apply to forests of varying species composition. It is also being used in sustainable resource management to predict the effects of planting and harvesting regimes.

SORTIE has evolved over a number of years under the direction of multiple programmers and while still quite useful has become unstable. The original architecture was not designed to support the maintenance of this application. We are working with a small group of researchers and programmers at the Institute of Ecosystem Studies to create a reengineering solution that increases maintainability, understandability and usability of SORTIE while maintaining similar performance characteristics.

1.2 Problem

Our challenge is to come up with a new architecture that supports the original application requirements as well as the reengineering goals and constraints. A key constraint on the

new design is that the existing development environment cannot be replaced and where possible, small-scale reuse of code is desired.

A good architecture takes into account how various requirements interact and considers trade-offs between architectural attributes [17]. Maintainability and understandability often come at the expense of performance, therefore it is important in this study to clearly identify where performance is a high priority and to assess the points in the architecture where maintenance is expected to occur.

The new architecture should be flexible and provide for extension at the maintenance hotspots. It is also important to consider the programmers who will be maintaining the system. Designing for flexibility can lead to an overly complex architecture that compromises understandability and is difficult to maintain. Once a design is determined, it is important to provide clear and prescriptive guidance for redesign and future extensibility so that the programmers can implement and maintain the new system.

1.3 Approach

A first step in addressing this problem is to understand the existing code base. Discovering which features the program actually does support and which parts of the system experience a high degree of modification is essential in predicting how the program might evolve. We were given the existing code base at the onset of this study and used both automated and manual reverse engineering techniques to uncover application knowledge. Since we are geographically separated from the primary stakeholders in this project, our communication mechanism is primarily through email.

To create an architecture that meets the reengineering goals we used a domain engineering methodology. Domain engineering involves analyzing domain legacy and capturing domain assets that can be applied to applications in the same class [4]. For adequate coverage, the domain analysis in this study included the broad domain of ecosystem modeling and simulation environments and was not limited to the type of model, however applications for spatially explicit individual-based models were the primary focus. The design process was collaborative and involved several site visits for brainstorming sessions with the application stakeholders. We used an iterative prototyping technique to uncover design problems and test out architectural approaches. We communicated our results through report style documentation and used presentation sessions as a discussion platform. The final design solution is a result of the lessons learned and insights gained through this process.

1.4 Solution

Our solution is presented in terms of an object-oriented framework. Frameworks represent the core part of an application that is reusable for other applications in the same problem domain [3]. They provide architectural guidance by partitioning the design into abstract classes and defining the responsibilities and collaborations of those classes. Our framework provides a solution that meets the requirements of the domain and prescribes how the SORTIE application can be reengineered and extended.

This study focuses on the core characteristics of the domain. These are the characteristics that support spatially explicit individual-based population modeling and simulation. However, it is also common for tools in this domain to allow researchers to explore ideas about how to manage ecosystems that contain a wealth of natural resources.

With this in mind, it becomes obvious that it is not only important to consider how an application for ecosystem modeling and simulation can be designed for maintainability, but also how to integrate these systems and the models contained therein with models for sustainable resource management. As part of this study we also looked at how current research in agent based technologies to support sustainable resource management can be integrated into the framework.

1.5 Thesis Outline

This thesis documents the reengineering project. In Chapter 2, we detail the research problem and provide an overview of the SORTIE application including an overview of what we learned from the reverse engineering process about the existing SORTIE architecture. In Chapter 3 we present our methodology and provide background on the techniques we used, including reverse engineering, domain engineering, object-oriented framework design and iterative prototyping. A summary of the domain analysis and the characteristics of the domain are given in Chapter 4. Our solution to the problem is presented in terms of an object-oriented framework that supports a set of common requirements in the domain. This framework is detailed in Chapter 5. In Chapter 6 we discuss how the framework can be applied to the reengineering of SORTIE. Our results from integrating agent-based technology into the framework are presented in Chapter 7. Finally, the implications and limitations of these research findings are summarized in Chapter 8.

Chapter 2

Background on Target System

Our study is based on SORTIE, a spatially explicit, individual-based model of the succession dynamics of mixed-species forests [21]. It is a research tool primarily used and maintained at the Institute of Ecosystem Studies in Millbrook, New York. The tool has become increasingly unstable over the last few years and was considered for our study in domain based reengineering. Figure 2.1 is a screen shot of the existing application interface. The visualizations are windows containing simple graphical representations of the data including a plot map, data table, bar charts and line graphs. The plot map represents the area of study.

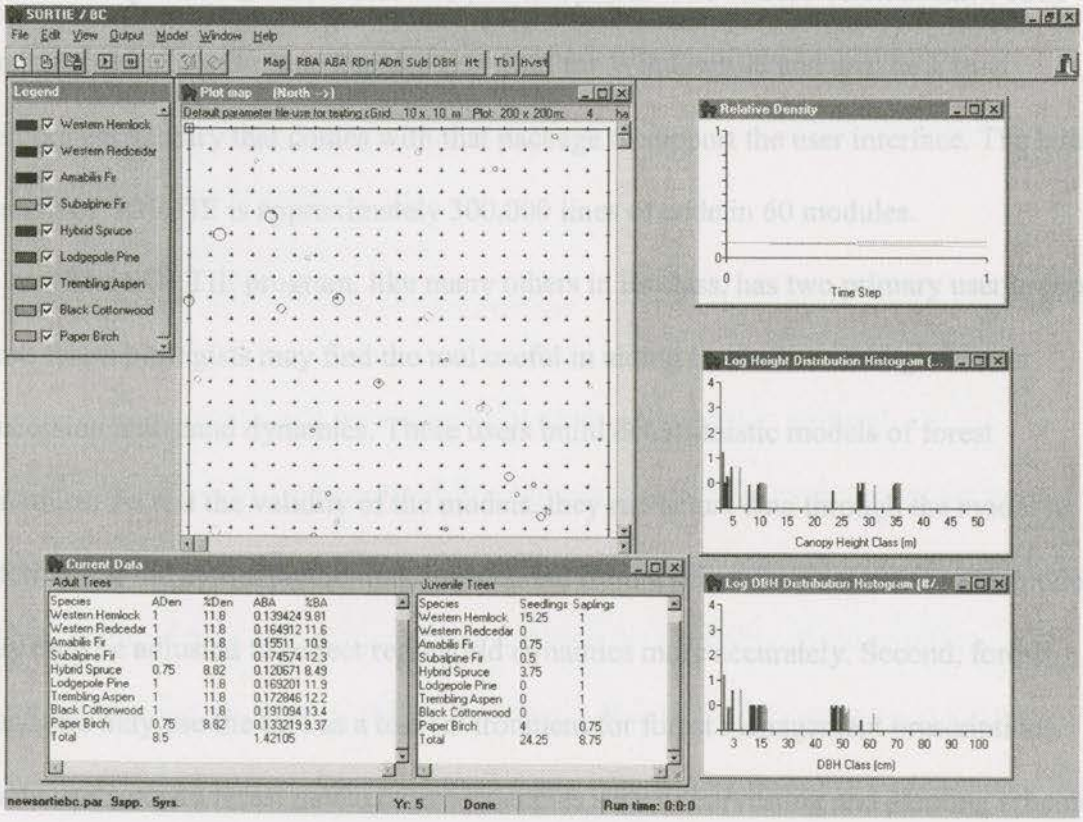


Figure 2.1 Screen Shot of SORTIE

In this chapter we provide details of the research problem and an overview of what we learned about the existing architecture from the reverse engineering process and through communication with application stakeholders. We give an overview of the primary tasks supported by the program, and discuss the architectural characteristics and code structure.

2.1 Reengineering Problem

SORTIE is a legacy application originally developed in C on a Unix platform and was ported to a C++ desktop environment. Although it was ported to an object-oriented language, the original architecture was largely unchanged and the code has a distinct C feel to it and does not take advantage of the benefits inherent in object-oriented designs such as encapsulation, inheritance and polymorphism. The current versions have been implemented using Borland Builder C++ [22] for Windows 98 and use the Visual Component Library that comes with that package to support the user interface. The latest version of SORTIE is approximately 300,000 lines of code in 60 modules.

The SORTIE program, like many others in its class, has two primary user groups. First, forest biologists may find the tool useful in aiding the understanding of forest succession and stand dynamics. These users build deterministic models of forest dynamics. To test the validity of the models, they run actual data through the model to determine if the model predictions are accurate from a scientific perspective. The models may then be adjusted to reflect real-world dynamics more accurately. Second, forest managers may use the tool as a test environment for forest management prescriptions. These users create forest management strategies such as harvesting and planting schemes

and apply them to the model to see their effect on the simulated forest. This can be very helpful in choosing strategies for sustainable resource management.

Over the last several years, the basic SORTIE program has evolved to incorporate new models and has been adapted for application to forests in the pacific northwest as well as the atlantic northeast. The application programmers have been adapting the source code according to the needs of the SORTIE researchers on an as-need basis. The existing architecture was not designed with this high degree of program evolution in mind and subsequently, the code base is quite complicated and difficult to maintain. The resulting program is somewhat fragile and is prone to memory leaks and crashes.

Despite the program weaknesses, it is still useful in modeling and simulating forest dynamics. The users of the software would like to reengineer the code to meet four high level reengineering goals:

1. Increase code maintainability
2. Improve code extensibility
3. Increase application usability
4. Maintain current levels of performance

In this context, code maintainability can be measured in terms of understandability [25]. Code extensibility refers to the architectural support for program evolution. The new design should support the modification of existing models and the incorporation of new ones without compromising the reliability of the program. Understandability refers to increased modularity and reduced complexity so the code can be maintained in a conceptually straightforward manner. While SORTIE does allow the modification of some forest variables and sub-model algorithms, the current architecture

fails to achieve the level of modularity required to allow for program extensibility. The main problem is that many of the objects that implement the core model functionality also perform other tasks such as file input/output and user interaction. The key usability concerns of this application are stability and performance. A researcher should be able to interact with the application without being concerned that their actions will cause application errors. Many of the models contain computationally intensive algorithms and the design must support efficient execution of these algorithms.

2.2 Overview of the Architecture

Through code inspection, program execution and communication with application stakeholders we discovered that the application supports four main user tasks:

- **Load Input Parameters**

  The Load task allows a user to specify an input file from which to load parameters and allocate program memory to store the information. The parameters characterize the forest being modeled in terms of number of species, forest plot size and slope, substrate content, species behavior.

- **Edit Parameters**

  The forest characterization can be changed during the Edit task through a series of user interface dialogs. The submodels may also be modified in a limited fashion. For example, the user can select absolute or relative juvenile growth and can select a gap or non-gap seed dispersal.

- **Run Simulation**

  The Run task executes the model over a series of time-steps. Each time-step is composed of a series of algorithms that drive the succession dynamics within the forest ecosystem. Each algorithm belongs to one of the sub-models.

- **View Outputs**

  At each time-step, if selected by the user, the successive forest transformations are graphically displayed in terms of spatial, height and density distribution through the use of two-dimensional colored graphs and plots.

A diagram of the user tasks and the activation of the core submodels is shown in Fig. 2.2 below.

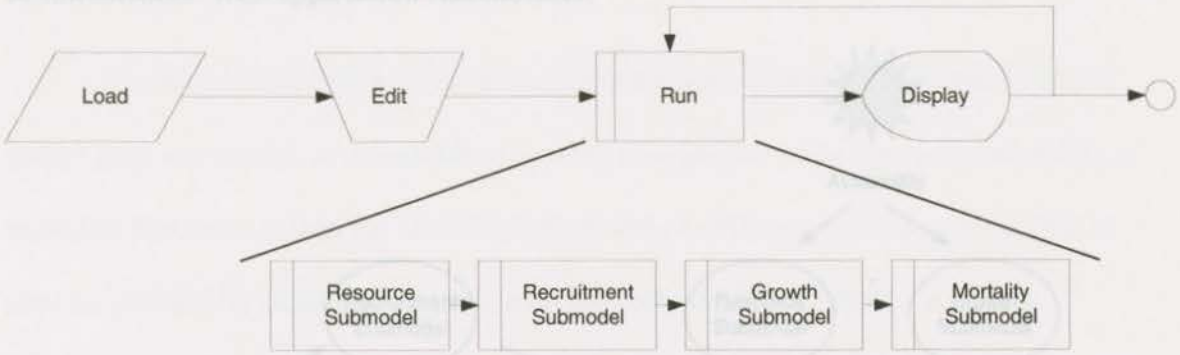


Figure 2.2 User Tasks in SORTIE

The basic SORTIE population model consists of four submodels that are executed during the run phase of the program. These submodels contain the algorithms that drive the simulation.

- **Resource Submodel**

  The resource submodel predicts the availability of resources for each seedling, sapling, and adult tree in a simulated stand. Light is the primary resource considered in the current model and is used to predict spatial variation.

- **Recruitment Submodel**

  The recruitment submodel predicts the distribution and density of new seedlings as a function of the sizes and distribution of parent trees.

- **Growth Submodel**

  The growth submodel uses the light level predictions of the resource submodel to predict radial growth on each individual in the forest.

- **Mortality Submodel**

  The mortality submodel uses empirical relationships between growth rates and survival of under-story seedlings and saplings to calculate the probability of survival.

An illustration of the submodels and their interactions is provided in Fig. 2.3 below. This representation of the SORTIE model is based on program inspection and communication with application stakeholders.

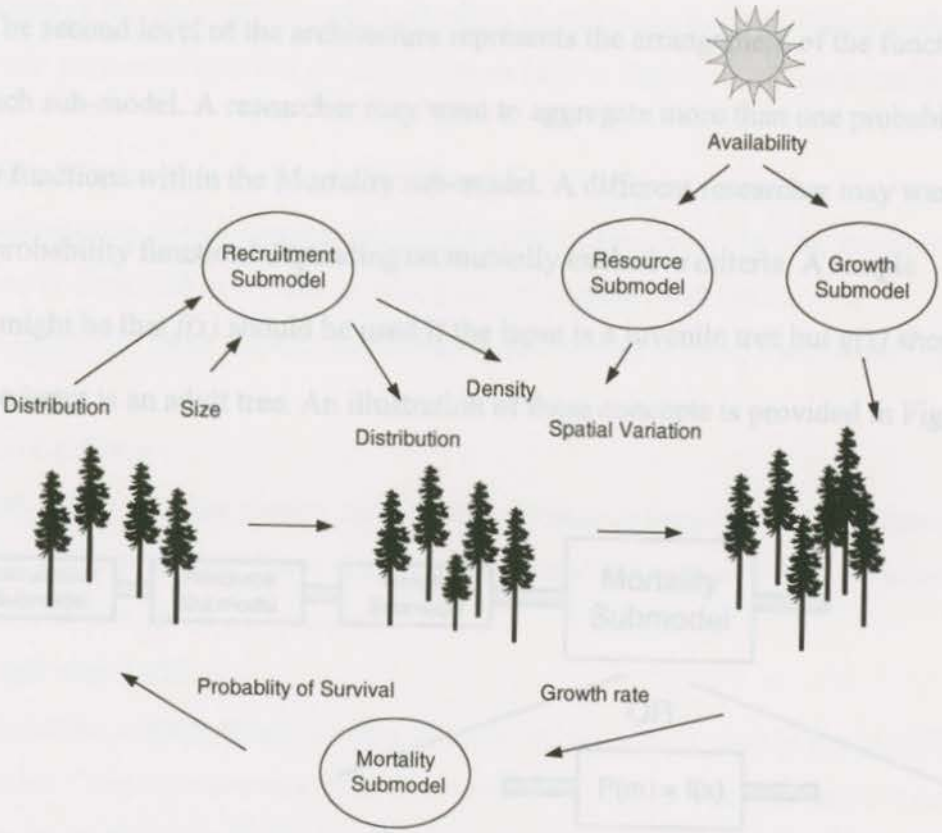


Figure 2.3 SORTIE Model

2.2.1 Architectural Characteristics

The existing SORTIE program has the characteristics of a data flow architecture [1]. The system performs an incremental transformation of data by successive components. The

SORTIE application can be further characterized as a pipe-and-filter style data flow architecture since data flow in the program is unidirectional and sequential [30]. Conceptually there are two levels of pipes and filters. This first level represents the high level relationships in the SORTIE model:

- Resources are a function of recruitment.
- Growth is a function of resources.
- Mortality is a function of growth.

The second level of the architecture represents the arrangement of the functions within each sub-model. A researcher may want to aggregate more than one probability of mortality functions within the Mortality sub-model. A different researcher may want to use two probability functions depending on mutually exclusive criteria. A simple example might be that *f(x)* should be used if the input is a juvenile tree but *g(x)* should be used if the input is an adult tree. An illustration of these concepts is provided in Fig. 2.4 below.

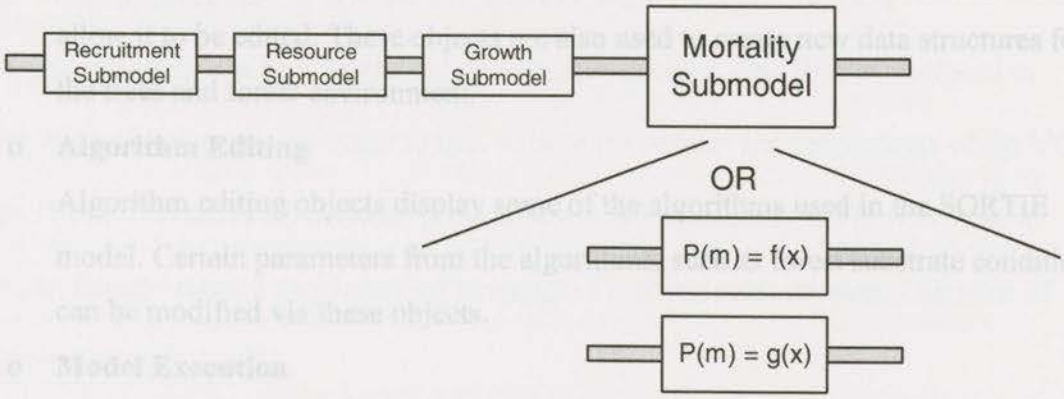


Figure 2.4 Pipe-and-Filter Style Data Flow

Currently, these variances in research strategies are integrated into the SORTIE code in a relatively *ad-hoc* fashion. The design of the program does not support the encapsulation of these differences. When a researcher wants to test new relationships and

theories, the core program source code must be modified, an expensive process that jeopardizes the integrity of the program.

2.3 Overview of the Code

Through code inspection, we created a class hierarchy of the existing code base. This exercise was valuable since we were able to see that the hierarchy was flat and that there is minimal use of inheritance. To gain a better understanding of how the system supports the user tasks, we analyzed the code from the perspective of the main user tasks. We stepped through the code and documented which classes were invoked during each task and for what purpose. From this process we were able to identify six support categories:

- **Data Display**

  Data display objects graphically display information from the data structures. The data is displayed in various graphs, plots, histograms and tables. This functionality helps the user understand how the data is being transformed by the system.

- **Data Editing**

  Data editing objects display modifiable information from the data structures and allow it to be edited. These objects are also used to create new data structures for the trees and forest environment.

- **Algorithm Editing**

  Algorithm editing objects display some of the algorithms used in the SORTIE model. Certain parameters from the algorithms, such as forest substrate conditions can be modified via these objects.

- **Model Execution**

  Model execution objects contain the core SORTIE model code and some of the data structures used by the model.

- **Data Control**

  Data control objects handle all input/output to and from the program input/output and contain many of the data structures used by the model.

- **Program Control Objects**

  Program control objects handle basic program operations such as user dialog and program options.

However, the actual classes in the program do not uniquely coincide with a single category. Many classes possess characteristics of more than one category. For example, there is a class called TMainWindow that handles most of the program control and contains the main application interface, however, it has many other functions such as clearing memory allocated to model data structures. Forest data is primarily stored in the TSortieIO class, the main class for handling data control, however there are no methods for providing external access to the forest data structures and many other program modules interface with the TSortieIO data structures directly.

2.3.1 Code Structure

The SORTIE program is implemented using Borland's Visual Component Library (VCL) [22]. VCL encapsulates the Windows Graphics Device Interface, or GDI.
GDI programming can be a subtle and complicated process. The VCL is designed to make this technology much easier to use. 75% of the objects are descendants of the VCL class TForm. TForm is a persistent, scrollable window control. SORTIE is written in C++, an object-oriented programming language. The three most important benefits of object-orientation are [3]

1. Encapsulation of data and operations
2. Code reuse through inheritance
3. Code extension/adaptation through polymorphism

However, the objects in SORTIE don't follow standard object-oriented programming guidelines and the code is more procedural in nature. First, the data objects are not strictly encapsulated. Data members of one object can be and are directly accessed by other objects. In an object-oriented design, data members of the source object should only be externally accessible using public methods. For the most part, this problem comes from the way the code has been evolved to incorporate new relationships. This kind of evolution tends to degrade the structure of software over time.

Second, there is only minimal use of inheritance in the code. Many of the graphical display objects have common characteristics and are descendants from a parent object. It would be useful for code reuse purposes if inheritance were used to manage new relationships. For example, the Mortality sub-model in SORTIE can use many different internal functions. Functions of this kind would have the same interface as well as the same external dependencies. If a parent "probability of mortality" class could be inherited, all of the common code could be reused and the task of incorporating a new relationship would be much simpler.

Third, the code doesn't use polymorphism. Polymorphism simplifies communication among objects by allowing the same general effect to be triggered by different objects without having to worry about the details of that effect [9]. Further to the example above, polymorphism could be used to override the "calculate probability" method of the object. The "calculate probability" method has the same interface in each inherited object, however the implementation details within each instance of the method could be different to reflect the particular calculation details.

2.4 Summary

The results of our reverse engineering efforts were beneficial in many ways. First, we were able to identify the points in the software that experience a high degree of maintenance. We described these points in the context of a pipe-and-filter style data flow architecture. The filters are the objects that contain algorithms to process data and the pipes are the pieces that pass data to each filter and connect it to the architecture. Second, we gained an understanding of the high-level user tasks in the system and the support the system provides the user in the completion of those tasks.

We have stated the application and reengineering requirements fairly broadly and at this level of inspection there are a number of architectural design candidates. However, there are constraints within the domain that need to be uncovered. These constraints drive the elimination of alternatives and help to narrow down design decisions. The reverse engineering methods we used failed to create a clear picture of the constraints of the domain so we turned to domain based reengineering methodology that combines a number of software engineering techniques.

Chapter 3

Background on Domain-based Reengineering

In this chapter we present background on the domain based reengineering methodology that was applied during our study. We discuss the blend of software engineering approaches that were used to solve our research problem. Our approach includes reverse engineering, domain engineering and object-oriented framework development through iterative prototyping.

The intent of domain engineering is to come up with a design that applies to a set or class of applications rather than a single application. One of the primary goals of this technique is to determine the range of variability that the design model must encompass so that it can be used to support the instantiation of many applications with the same core characteristics [4]. The domain analysis included collaborative work sessions with the SORTIE stakeholders to uncover domain knowledge, and the examination of legacy application code and documentation. For adequate coverage, we also looked the broad domain of ecosystem modeling and simulation environments and did not limit our scope to applications specifically for spatially explicit individual-based models. Existing systems and toolkits within this domain were analyzed and compared to determine common requirements, design and development strategies.

Frameworks map well onto the domain engineering process since they represent the core part of an application that is reusable for other applications in the same problem domain. A framework captures the design decisions that are common to its application domain thus emphasizes design reuse over code reuse [3]. They can be used to document

the design decisions that result from uncovering the domain constraints and provide architectural guidance by partitioning the design into abstract classes and defining the responsibilities and collaborations of those classes. We have designed a framework in collaboration with Institute of Ecosystem Studies that supports the general characteristics of the domain while still meeting the specific SORTIE application requirements as documented. The final solution is a result of iterative prototyping and framework development in collaboration with the application stakeholders.

3.1 Reverse Engineering

The goal of reverse engineering is to describe the existing system as-is, typically for the purpose of reengineering [26]. There are many tools available that aid in program understanding and reverse engineering. Two such tools were used in the early stages of this project, however most of our results came from the use of a manual reverse engineering strategy.

3.1.1 Reverse Engineering Tools

Rigi

Rigi [23], a product developed by researchers at the University of Victoria, is a visual tool for understanding legacy systems. Rigi enables the engineer to model the application domain using a general Rigi graph model, and supporting graph editor. The program allows the artifacts of the application domain to be captured at various levels of abstraction including: subsystems, procedures, variables, calls, data accesses, and dependencies.

At the onset of the project, it was hoped that Rigi could be used to extract the artifacts mentioned above from the code. Rigi includes parsers to analyze programs written in C, C++, Java and COBOL. The as-is C++ parser supports a specific platform: Windows NT and IBM's VisualAge C++. In order to use the C++ parser, the SORTIE source code would have to be ported to this platform. The use of Borland Builder's proprietary libraries in the code made this a challenging proposition. Another option would have been to build a custom parser where the outputs from the parsing could be loaded into Rigi. This route was abandoned and Imagix 4D was explored.

Imagix 4D

Imagix 4D [24], a proprietary product developed by Imagix Corporation, is a reverse engineering and documentation tool for legacy C and C++ programs. Imagix 4D offers similar functionality to Rigi in that its purpose is to aid program understanding through graphical representations of the program at different levels of detail. This tool was selected because its supporting documentation indicated that the program included configuration files for source code built in Borland C++. After working with the Imagix support department, it was discovered that the configuration files support the Borland Compiler and not the Borland Builder. The Builder provides a proprietary graphical user interface development toolkit that required extensive modifications to the macro definitions in the configuration files that are used by Imagix 4D to parse the source code. This route was also abandoned given that the SORTIE program is not very large and the effort required to parse and load the code for the purpose of visualization did not seem to provide enough benefit.

3.1.2 Manual Strategy

The SORTIE program has been partially reverse engineered following a process used on the Army/STARS/Unisys Demonstration Project [2]. This process includes analyzing legacy system artifacts such as source code and documentation, mining of human expertise, and modeling of operational capabilities. Lower-level system information is used to understand the system at a higher level. The details at each step, or artifacts and assets as they are sometimes called, help to describe the system as-is, however, the ultimate goal is usually to produce a statement of requirements that can be used for reengineering the existing system or forward engineering of a new system. An illustration of this process is provided in Fig. 3.1 below [2].

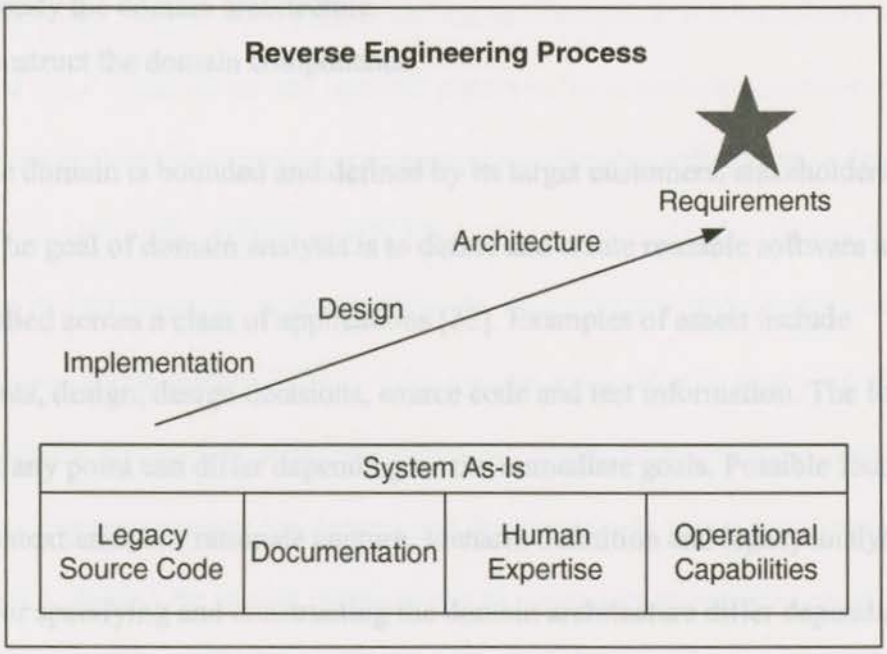


Figure 3.1 Reverse Engineering Process

Most of our understanding of the system as-is came from manually inspecting legacy source code and modeling the application's operational capabilities in terms of user tasks and the system support for those tasks. We verified our knowledge through

communication with application stakeholders since we had very little access to documentation. This kind of process has also been described as analyzing the subject system to identify its current components and their dependencies and to extract and create system abstractions and design information [26].

3.2 Domain Engineering

Domain engineering is the process of modeling a class of subsystems that will support reuse. The process includes four high level tasks [4]:

- Define the domain
- Analyze the domain
- Specify the domain architecture
- Construct the domain components

The domain is bounded and defined by its target customers, stakeholders, and features. The goal of domain analysis is to define and create reusable software assets that can be applied across a class of applications [32]. Examples of assets include requirements, design, design decisions, source code and test information. The focus of the analysis at any point can differ depending on the immediate goals. Possible focus areas include context analysis, rationale capture, scenario definition and legacy analysis. Methods for specifying and constructing the domain architecture differ depending on the domain. A diagram of the domain engineering process is provided in Fig. 3.2 below [4].

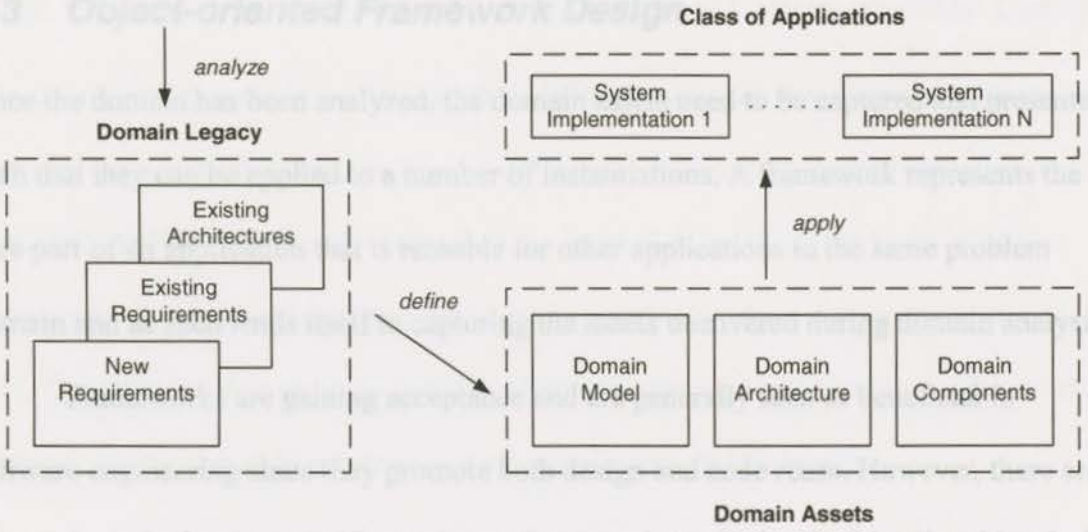


Figure 3.2 Domain Engineering

We used an iterative approach to uncover and analyze the artifacts in this domain. Initially, the broad field of ecosystem modeling and simulation was examined. We investigated other applications and research papers in the domain and documented our understanding of their functionality and architecture. We then refined our understanding of the domain during site visits with SORTIE application stakeholders. It was through this process that we began to understand how the type of model impacts the design of the architecture and refined our definition of the domain to specifically target individual-based population models and began to explore and answer detailed design questions. We captured our understanding of the domain and design ideas in a framework design document and evolved the design through brainstorming and prototyping sessions with the researchers at the Institute of Ecosystem Studies. This process is typically referred to as a spiral model of software development [27].

3.3 Object-oriented Framework Design

Once the domain has been analyzed, the domain assets need to be captured and presented such that they can be applied to a number of instantiations. A framework represents the core part of an application that is reusable for other applications in the same problem domain and as such lends itself to capturing the assets uncovered during domain analysis.

Frameworks are gaining acceptance and are generally seen as beneficial in software engineering since they promote both design and code reuse. However, there are no standards for how to specify or create a framework, so the level of detail and level of code reuse varies widely. In any case, frameworks are not intended to be executable software entities. If a development project can rely on design and code reuse, a number of benefits are obvious [34]:

- Reduction in software development cost
- Shorter software development time
- Better estimation of development time
- Higher quality software

In this study, we benefited from framework development since we were able to gain a better understanding of the general and variable characteristics of the domain and how applications in the domain support those characteristics. During the domain analysis we examined a number of designs. It was important to understand what aspects of those designs might be applicable to the SORTIE application as it is currently. But it was also important to get a sense of variability that occurs within the domain so that new models can be incorporated in the future.

3.3.1 Framework Development vs. Traditional Object-oriented Design

Object-oriented frameworks offer a slightly different approach to traditional software reuse techniques as illustrated in Fig. 3.3 below [31]. Rather than simply providing a library of small software components that can be used as building blocks for a system, frameworks provide an abstraction of the entire architecture for a class of applications [11]. The abstraction is captured in terms of general concepts and structures. This creates a general design that can be applied for each application in the domain.

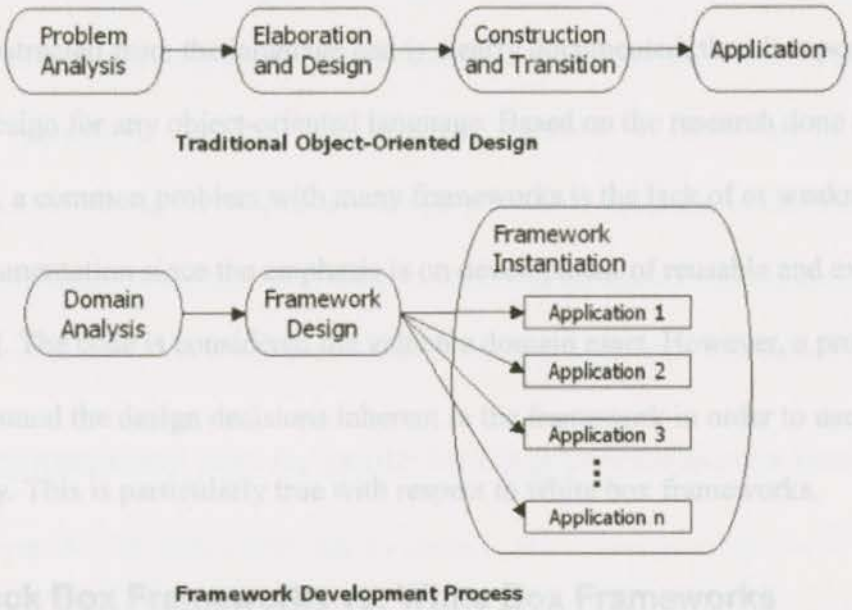


Figure 3.3 Framework Development versus Traditional Object-oriented Design

An object-oriented framework generally consists of a set of collaborating classes. Depending on the level of code reuse provided, the classes could be abstract or concrete or the framework might provide both. In any case, a successful framework follows a few key principles [3]. These principles have been followed in the creation of the framework and are described in this document:

- Separate the interface from the implementation
- Determine what is general and what is variant within an implementation
- Allow the substitution of variant aspects via a common interface
- The general components should be stable, open for extension but closed for modification

The provision of concrete classes can greatly speed up development since those classes can simply be extended. However, it also makes a framework language specific and typically the code becomes the focus rather than the design documentation. If the design is abstracted from the language and is clearly documented, then it is possible to reuse the design for any object-oriented language. Based on the research done as part of this project, a common problem with many frameworks is the lack of or weakness in design documentation since the emphasis is on development of reusable and extendable classes [33]. The code is considered the valuable domain asset. However, a programmer must understand the design decisions inherent in the framework in order to use it successfully. This is particularly true with respect to white box frameworks.

3.3.2 Black Box Frameworks vs. White Box Frameworks

There are two main kinds of frameworks: black box frameworks and white box frameworks [35]. With white box frameworks, it is necessary to create new classes based on the classes provided by the framework. White box frameworks are also sometimes called architecture-driven frameworks and require a good understanding of the framework so that it can be successfully programmed. Black box frameworks typically come with configuration scripts and some kind of automated instantiation tool to instantiate the framework by creating the classes. Therefore, a black box framework does

not require the user to have a detailed understanding of the underlying framework; a person using the framework only needs to know how to supply the tools with data. For this reason, black box frameworks are sometimes referred to as data-driven frameworks [11].

3.3.3 Class Libraries vs. Frameworks

Perhaps the biggest and most important difference between traditional class libraries and frameworks is that frameworks provide an application architecture. If a programmer is simply using a class library, they are responsible for determining the overall control flow through the application, whereas if they are using a framework, the control flow is predefined. Another way of looking at it is that when a programmer writes code for a framework, the intention is that their code fits into the framework and is called by the framework constructs. When using a class library, they write code that calls a library [9].

3.3.4 Polymorphism and Dynamic Binding in Framework Design

The real power and flexibility provided by frameworks comes from the use of polymorphism and dynamic binding. It is achieved by providing a common interface defined in a base class, which can then be extended to accommodate the different implementations. The typical example used to explain polymorphism is the class hierarchy for drawing shapes. The Shape class hierarchy is shown in Fig. 3.4 below.

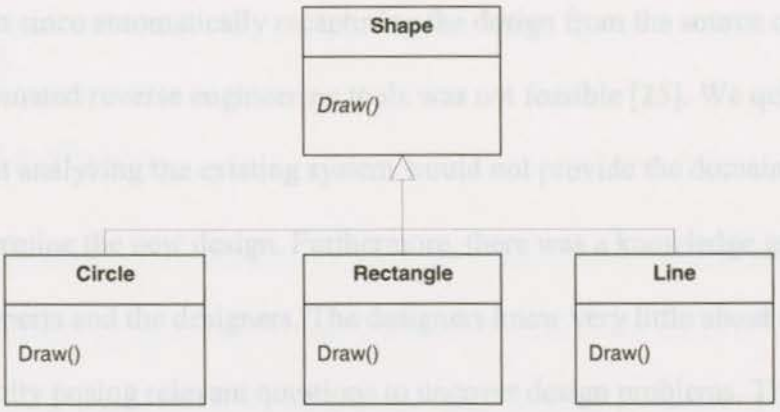


Figure 3.4 The Shape Class Hierarchy

The Shape class is an abstract class that provides a template, a set of interfaces, that all subclasses should implement in order to conform to the common interface. The Shape class provides a *Draw()* method as shown in Fig. 3.4 above. How the Circle implements Draw() will be different than how Rectangle implements since they are completely different shapes. However, the interface is the same so the effect can be triggered in exactly the same way regardless of the type of shape. Dynamic binding is the way object-oriented programming languages implement polymorphism.

Another point that should be considered with the use of frameworks in software engineering is the inherent trade-off between flexibility and performance. Dynamic binding provides the needed flexibility but adds overhead that can negatively affect performance and if not done very carefully can lead to complex software systems that are difficult to maintain.

3.4 Summary

Our reengineering methodology uses a set of well-accepted software engineering techniques and collaboration with stakeholders. In this study, a manual reverse engineering method was appropriate to gain an understanding of the existing system and

its basic design since automatically recapturing the design from the source code through the use of automated reverse engineering tools was not feasible [25]. We quickly discovered that analyzing the existing system would not provide the domain knowledge needed to determine the new design. Furthermore, there was a knowledge gap between our domain experts and the designers. The designers knew very little about the domain and had difficulty posing relevant questions to uncover design problems. The domain experts knew very little modern software engineering techniques and did not know what aspects of their domain might be relevant to creating the new design. We needed a mechanism to bridge this knowledge gap and turned to domain engineering as a communication mechanism. We strengthened our understanding of the domain characteristics through domain-driven investigation and used an iterative participatory framework development process to capture and present our knowledge. Throughout the design of the framework, we were able to explore and evaluate design alternatives by applying the domain-based solutions specifically to the SORTIE application in collaboration with the programmers and stakeholders. The design solutions we explored came from our analysis of the domain and included research papers, legacy research tools, and other frameworks to support ecosystem modeling and simulation.

Chapter 4

Domain Analysis

SORTIE is an application for spatially explicit individual-based population modeling and simulation. A spatially explicit individual-based model tracks the states of each individual within a population and uses a spatial grid structure to model the environment. The model includes state transitioning rules to determine how the individuals interact with each other and respond to their environment. Traditional ecosystem models consider all aspects of the ecosystem as biomass. These models tend to be simpler since the details of the populations within the ecosystem are not modeled in their own right. A spatially explicit individual-based model is a type of ecosystem model where individuals can be tracked within a spatially explicit environment.

Our domain analysis included legacy applications and research papers of this model-type specific domain as well as the relatively broad domain of general ecosystem modeling and simulation. The focus of our analysis in the specific domain was to uncover common characteristics, requirements and constraints as well as alternative solutions to design problems. To gain additional insight as to how we might design for extensibility, we increased the scope of our analysis to the general domain.

There are many legacy applications, toolkits, frameworks and research papers related to ecosystem modeling and simulation. Recently, there has been a fair bit of research and interest in developing frameworks or toolkit type applications for building modeling and simulation environments [6][12][14][18]. These tools tend to focus on providing for extensibility. Specifically, they attempt to make it easy to aggregate

different models of ecosystem dynamics and resource management models to predict their combined effects on the ecosystem.

In this chapter, we present representative domain legacy and summarize the domain characteristics to provide motivation for the solution presented in the next chapter. This domain legacy is divided into two groups: general ecosystem modeling and simulation architectures, and those that specifically address spatially explicit individual-oriented population models. These two groups are detailed in 4.1 and 4.2 respectively. In 4.3 we summarize the domain characteristics.

4.1 General Ecosystem Modeling and Simulation

We investigated a number of systems and documentation for general ecosystem modeling and simulation. A key characteristic of general ecosystem modeling and simulation tools is their provision for multi-model integration and this is reflected in their open architectural structure and design. The example instantiations are typically of watersheds or landscapes where the basic environment constructs are modeled as biomass rather than individually.

The two systems that were examined in detail were ATLSS [18], a project coordinated by the Biological Resources Division, US Geological Survey and OO-IDLAMS [5], developed at Argonne National Laboratory.

4.1.1 ATLSS

ATLSS supports multi-model integration, a key requirement of the reengineering study. However, we did not have access to the ATLSS architecture documentation and instead looked at a technical report of the ATLSS architecture [10]. A multi-model approach is

described as a modeling technique for complex systems where model aggregation at multiple levels is required [7].

The key insights learned from this process of examination were about the design of entity-model interactions to support entity extensibility and multi-model integration. It was discovered that there is a need for a clear separation of entity state and behavior at the individual and landscape levels.

4.1.2 OO-IDLAMS

OO-IDLAMS is based on the Dynamic Information Architecture System (DIAS), an object-oriented framework for modeling and simulating complex, dynamic, multi-scale processes [5]. DIAS [12] was developed at Argonne National Laboratory. It claims to support multi-disciplinary models within a common framework. This framework attempts to provide a reusable code base which is fairly typical in recent research and development in this area.

A key design principle used in the DIAS framework is the decoupling of models within the system. Models communicate only with entity objects, never directly with each other. This makes the framework flexible enough to support interchangeability of models as long as they share the same interface, so different implementations of a similar behavior can be easily hooked in. There is also some flexibility built into the entity objects themselves. Each entity object has variable attribute lists that can be adapted to fit a particular need or implementation.

The control hierarchy among the program objects is what makes this architecture so flexible and it is why it was included in the domain analysis. The Event Simulation Manager cues entity objects to perform certain aspects of their behavior. These behaviors

are addressed using process objects that encapsulate a particular model or provide access to a wrapped external model. This allows for simple model interchangeability. An illustration of the DIAS framework is provided in Fig. 4.1 below.

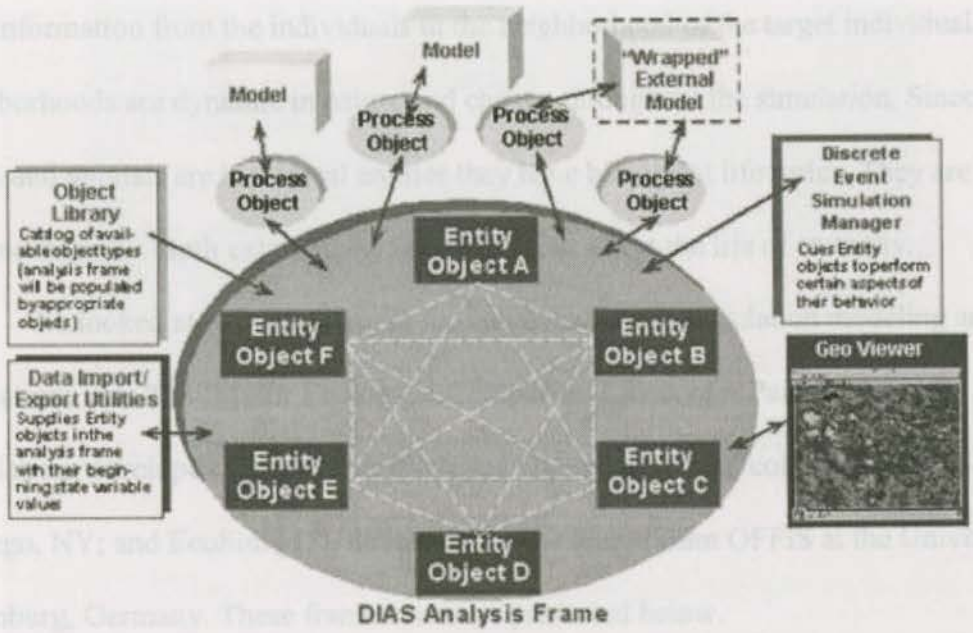


Figure 4.1 DIAS Framework

4.2 *Individual-based Population Modeling and Simulation*

The majority of tools that specifically address spatially explicit individual-based models are designed for individuals that require spatial mobility. These environments are commonly designed to support animal population dynamics where the animals are modeled as individuals. Whereas, SORTIE is a model of succession dynamics of mixed species forests. In this case, the individuals are plants rather than animals. There are some subtle differences but many similarities. These similarities provide the motivation for being able to build a framework that is general enough to be used for individual-based models, regardless of the type of individual.

In both cases, individuals are spatially located at a particular snapshot of time, exist in dynamic populations and compete for resources within the environment. Some processes may require that the next state of the target individual be calculated based on state information from the individuals in the neighborhood of the target individual. These neighborhoods are dynamic in nature and change throughout the simulation. Since both plants and animals are biological entities they have biological lifecycles. They are born, live and then die. Both external and internal events affect the life of an entity.

We looked at two frameworks for individual-based population modeling and simulation: ECLPSS [6], for Ecological Component Library for Parallel Spatial Simulation, developed by the Plant Modeling Group at BTI and collaborators at SUNY in Oswego, NY; and EcoSim [13], developed by the Kuratorium OFFIS at the University of Oldenburg, Germany. These frameworks are presented below.

4.2.1 ECLPSS

The ECLPSS environment includes a modeling framework and a library of reusable components. The goals of this project overlap with the reengineering goals of SORTIE. ECLPSS allows biologists to build new models and experiment with the model structure without tedious reprogramming as the biological concepts are separated from the routine programming tasks.

A large part of the ECLPSS environment is managed via a Graphical Model Building Application. This allows modelers to specify model descriptions as text. These bits of text are translated into code fragments that are eventually linked with the components from the library. This process is depicted in Fig. 4.2 below.

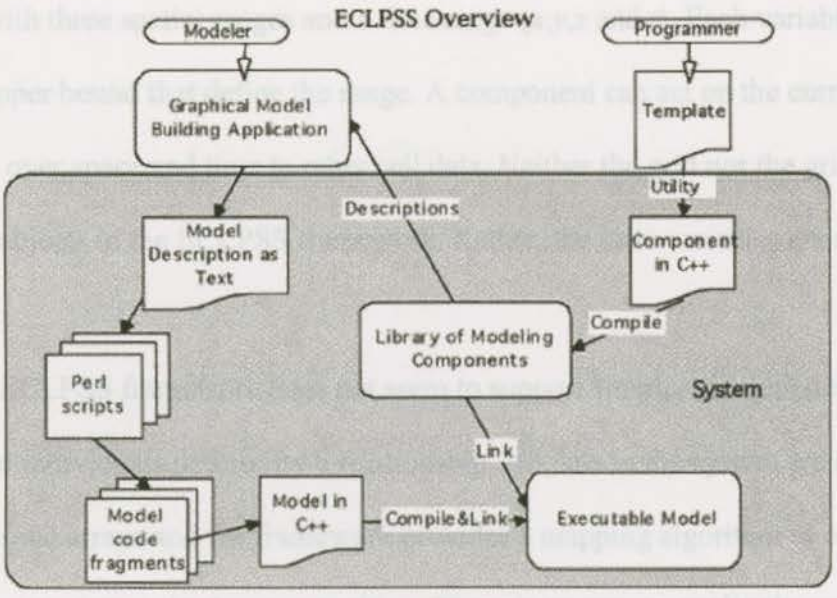


Figure 4.2 Overview of the ECLPSS Environment

The ECLPSS environment is based on a spatial grid representing the area of analysis. Each cell in the grid contains a list of state variables representing the state of the ecosystem at that position in the grid. Every component has access to this state variable list. The system runs through the grid, invoking components to act on each cell in a specified order. If a component requires information outside of the current cell, the component can read data from nearby locations based on physical distance rather than the number of grid cells. The components do not communicate with each other directly, they must communicate via the state variables. If component A requires something from component B, component B must run first and update a common state variable which is later accessed by component A.

Individuals are not modeled as objects; instead, attribute information is captured as a state variable. This attribute information of individuals is not distinguished from other types of ecosystem attribute information. Each state variable in ECLPSS is

associated with three spatial ranges and a time range (x,y,z and t). Each variable has a lower and upper bound that define the range. A component can act on the current cell's data or span over space and time to other cell data. Neither the grid nor the grid cells are modeled as objects in the ECLPSS framework. Rather, the state variables are managed in linked lists.

The ECLPSS framework does not seem to support simple, efficient determination and access to individuals defined by a relationship. All data in the system are stored as one-dimensional arrays and the framework provides a mapping algorithm to convert the time space vector into a single array position. There is no way to explicitly model and directly access data values that are related to the data value being examined.

The main design insight that we gained from the ECLPSS environment was the separation of state and behavior and architectural limitation that components have no direct knowledge of each other and no way to communicate with each other. That way if a component is removed or added, there is minimal disruption to the system. Also, new state variables can easily be added without having to rewrite existing components.

4.2.2 ECOSIM

ECOSIM also provides a class library especially designed to support spatially explicit individual-oriented modeling and simulation of ecological systems [13]. Each cell in the grid is modeled as an object that can change the environment. Each individual in the ecosystem is also modeled as an object that can perform actions at any time during the simulation. All processing, whether at the cell or individual level is managed by a scheduler. Cells and individuals can have actions or tasks that can be scheduled and

triggered for execution at a particular time during the simulation. The main class hierarchy is shown in Fig. 4.3 below.

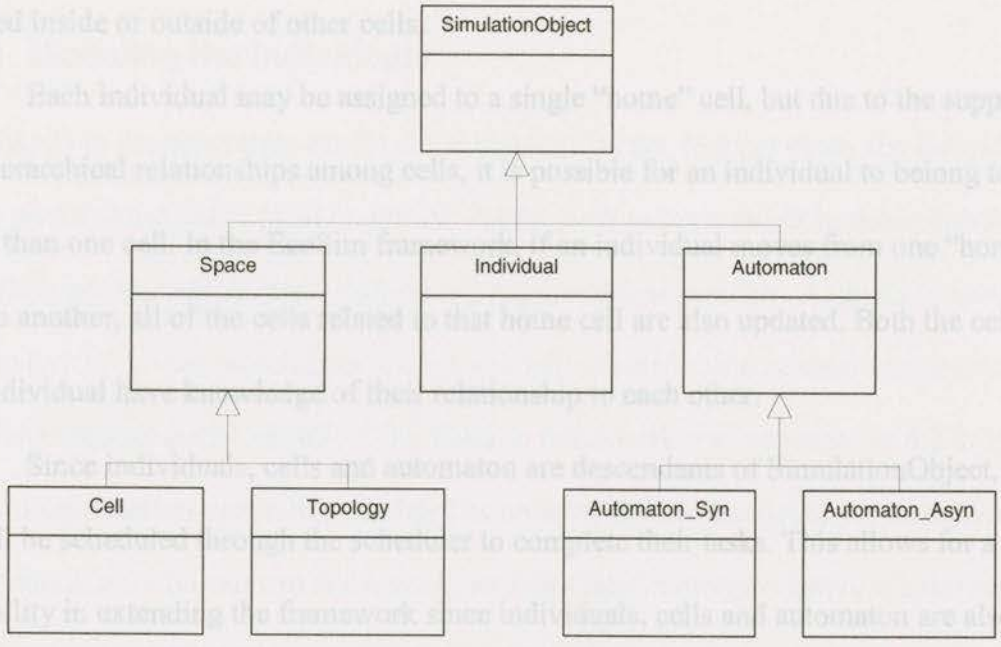


Figure 4.3 EcoSim Class Hierarchy

Both cells and individuals are considered active objects since they can perform actions any time during the simulation. All active objects in the simulation environment are derived from the base SimulationObject. These objects must be scheduled by a scheduler to cue the actions they need to perform. It is therefore a fundamental property of a SimulationObject to be registered with an object that descends from the Scheduler class. A scheduler must know when the active object should be scheduled and what task should be performed.

A cell has two key characteristics: first, individuals may be located within a cell and second ,cells may be related to each other via a topology. The Topology object supports the requirement that individuals often move relative to their current cell and need information about neighboring cells. It captures the relationships between cells. In

fact, the EcoSim class structure goes a step further by not only supporting neighborhood relationships, but also supporting hierarchical relationships between cells. Cells can be located inside or outside of other cells.

Each individual may be assigned to a single "home" cell, but due to the support for hierarchical relationships among cells, it is possible for an individual to belong to more than one cell. In the EcoSim framework, if an individual moves from one "home" cell to another, all of the cells related to that home cell are also updated. Both the cell and the individual have knowledge of their relationship to each other.

Since individuals, cells and automaton are descendants of SimulationObject, they can all be scheduled through the scheduler to complete their tasks. This allows for a lot of flexibility in extending the framework since individuals, cells and automaton are also related to each other. Automata operate on cells; cells contain individuals and individuals have knowledge of the home cell to which they belong. The implication is that a programmer using this framework can create robust automata that are scheduled to process specific cells and the individuals they contain or they can make the cells and individuals themselves robust.

4.3 Domain Characteristics

Based on our analysis of the domain legacy, we were able to determine that there are five core domain characteristics. Applications in this domain should support models that apply to populations of individuals in a spatial environment where there are defined relationships among the individuals and their environment. They should also include support for scheduled processing of models that address multiple levels in the ecosystem

including individual behavior and landscape dynamics. These characteristics are detailed below.

4.3.1 Modeling the Individuals

Individuals in the ecosystem can be any number of things. In most cases, the individuals being modeled are either plant or animal. Plants, such as trees, exists in dense populations and while are spatially located, they do not require spatial mobility. Animals, however, tend to live in sparser populations and are spatially located at a particular moment in time, and require spatial mobility. The EcoSim framework was presented in 4.2.2. In this design individuals are explicitly modeled as objects, objects that occupy a particular cell in the simulated environment and contain attribute information to capture the state of the individual. The Individual base class does not provide attributes or methods relative to a specific point in the cell, since the EcoSim framework has largely been design to support individuals that require spatial mobility. However, exact coordinates could be supported as attributes in a class that descends from the Individual base class.

4.3.2 Modeling the Environment

Information also needs to be captured about the state of the environment. Environment attributes are typically defined at varying scales depending on how the information is collected but also depending on how it is required by the models within the ecosystem. In the EcoSim framework, the environment is modeled explicitly as cells. There is no singular object that captures the entire grid; rather, the cells can be related to each other via a Topology object. As described in 4.2.1, with the ECLPSS model, the environment attributes are captured for a range or a particular point in space and time and then further

organized into a grid area. Our solution blends these ideas and adds a layer of modularization. The environment can be explicitly modeled as cells and organized into a grid structure, but the attributes that capture the state or conditions within that cell are also modeled as objects in their own right to support the concept that conditions that may not necessarily coincide with the segmentation of the cells. Conditions occupy a space in the environment. Rather than having a singular absolute coordinate, they have a coordinate range and intersect with cells.

4.3.3 Modeling the Relationships

In a spatially explicit individual-based population model, data is typically captured and modeled in a discrete manner. That is, data exists discretely at particular points in space. It is necessary to relate this data in some meaningful manner for deterministic processing. As seen in the frameworks presented, the data and relationships are often defined within the context of a grid concept.

Many different kinds of relationships exist in these environments. There are relationships between individuals. For example, individuals may be organized in social groups and may demonstrate group behavior dynamics. Or it could be that individuals close to each other display competitive behavior. These examples embody the primary relationship requirements among individuals. Individuals can either be explicitly related to each other or they can be related to each other based on their proximity. At any point in time and space, an individual may leave or join a relationship. If the individual is explicitly tied to the relationship, then it must explicitly request participation. However, if the relationship is based on proximity, then participation is implicit, and derived from the

individual's proximity. Illustrations of these relationship types are provided in Fig. 4.4 and Fig. 4.5 below.

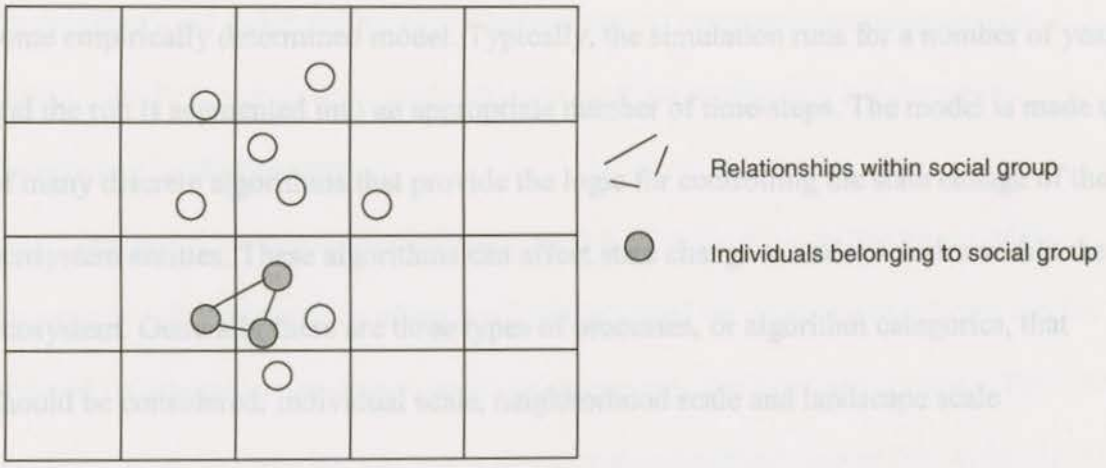


Figure 4.4 Group Relationship

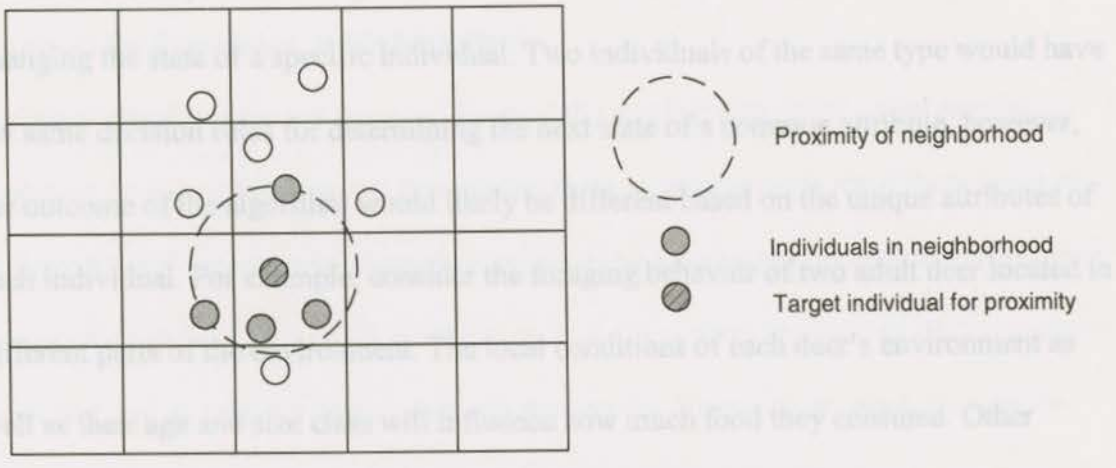


Figure 4.5 Proximity Relationship

The conditions within an ecosystem can also be seen as a type of relationship in the environment. These conditions occupy some space within the environment, but may not necessarily coincide with the segmentation of the cells. The EcoSim framework provides for this by allowing overlapping cells of variable resolution.

4.3.4 Modeling the Processes

Ecosystem modeling and simulation environments simulate ecosystem change based on some empirically determined model. Typically, the simulation runs for a number of years and the run is segmented into an appropriate number of time-steps. The model is made up of many discrete algorithms that provide the logic for controlling the state change of the ecosystem entities. These algorithms can affect state change at various scales within the ecosystem. Generally there are three types of processes, or algorithm categories, that should be considered: individual scale, neighborhood scale and landscape scale processes.

Individual Scale Processes

Individual scale processes target specific individuals and are only concerned with changing the state of a specific individual. Two individuals of the same type would have the same decision rules for determining the next state of a common attribute, however, the outcome of the algorithm would likely be different based on the unique attributes of each individual. For example, consider the foraging behavior of two adult deer located in different parts of the environment. The local conditions of each deer's environment as well as their age and size class will influence how much food they consume. Other examples include individual growth behavior and mortality of individuals.

Neighborhood Scale Processes

Neighborhood scale processes are concerned with changing the state of some attribute with respect to a neighborhood of individuals. These kinds of processes potentially need to examine the conditions and individuals of a particular area within the environment. It might be necessary to determine the area, or neighborhood, relative to a particular

individual, in which case it is possible for each individual to have a unique neighborhood that changes dynamically according to their current location. It could also be the case that the neighborhood is defined by an attribute of the environment. Examples include shading, seedling recruitment, seed predation, nutrient availability and browsing.

Landscape Scale Processes

Landscape scale processes can target any number of individuals or conditions of the environment. In this case, the effect is calculated based on some external factor and the affected individuals or conditions in the target area are identified and changed. There are two steps to landscape scale processes: describe the occurrence across the landscape then translate the effect of the occurrence on relevant individuals. Examples include natural and man-made disturbances such as wind, fire, harvesting and disease.

Support for Processing at Multiple Scales

Architectures that were not designed for spatially explicit individual-based population modeling do not readily support these distinctions in process scale since there is not an underlying grid structure for managing and tracking the ecosystem entities. For instance, as described in 4.2.1, the DIAS architecture does not support spatially explicit population modeling and as such the design does not support the distinction between the various process scales. There are no mechanisms for determining a neighborhood of individuals or to translate a landscape scale event onto target individuals.

The ECLPSS framework is designed to support spatially explicit individual population modeling, however, it does not readily support all of these scales. As has been mentioned, it does not provide mechanisms for dynamically determining a neighborhood around a target individual. It does however support individual scale processes and

landscape scale processes to some degree since individuals can be targeted for processing based on their attributes or spatial location.

The most useful guidance for how to support processes at varying scales comes from the EcoSim architecture since it is also designed for spatially explicit individual population modeling and models the various scales explicitly in the design of the architecture. That is, both the individuals and the cells are modeled in their own right and the architecture also supports the relationships between the scales, individuals are contained within cells, cells are related through topology. This implies that a landscape scale process could easily be translated onto the target individuals and it is relatively efficient to dynamically determine the neighborhood of an individual. And since both individuals and cells can be scheduled to perform tasks, it is reasonably straightforward to schedule the processes at various scales.

4.3.5 Scheduling of Processes

The processes that drive state change within the simulation need to be scheduled for execution. Some processes are required at each timestep. These are typically processes that reflect the natural lifecycles of the ecosystem entities such as birth, growth and death. Others are only required at particular points during the simulation. These are typically processes that reflect rarely occurring external events such as fire and disease. It may also be necessary to schedule a new process dynamically during the simulation run. The frameworks presented in the domain analysis address this requirement by providing some kind of discrete event scheduler. The DIAS architecture supports the scheduling of events made up of an entity object and a behavior. The EcoSim framework supports the

scheduling of a simulation object and a task. However, the underlying mechanism is the same. Both the DIAS behaviors and the EcoSim tasks are methods of the target object.

Processes can be run cell by cell, whereby the system runs every component on each cell at each time step. This approach is followed in the ECLPSS design. It seems that this method could cause unwanted patterns in model behavior. For instance, consider a model that contains algorithms for tree growth, dispersal and death behavior. Each of these algorithms is captured within a component. The system is instructed to run the components in the order mentioned above. That means that the system would start at the first cell in the grid and run all behaviors. So the trees contained in the first cell would experience growth, death and dispersal before the rest of the cells in the grid. It is highly likely that some of the algorithms could take into account state information from neighboring cells. This could cause a ripple effect in the system. The changes on each cell at each time step could directly affect the neighboring cells, something these algorithms may not necessarily take into account. A better approach would be to run each component across every cell in the grid before executing the next component.

4.4 Summary

We examined a number of applications and frameworks as part of the domain analysis. The ones presented in this chapter offer some guidance as to how an application for spatially explicit individual-based population modeled can be constructed. There are two points where the examined applications differ significantly in their design. The first is in the degree of separation between the ecosystem state and the processes that drive the changes to that the state. The other is in the scheduling of these processes.

There are two ways that the design of ecosystem state and behavior is approached. The first approach distinctly separates the state of the ecosystem from the processes within the ecosystem. In this case, the processes act on target individuals or areas within the ecosystem. The benefit of this approach is that processes can easily be changed or added to the simulation without disrupting the rest of the system and the individuals they act on. Processes can be quite easily combined and manipulated without reprogramming. It also makes the design of the individuals in the ecosystem much simpler and more general since the behavior code is external to the individual. However, the biggest challenge with this approach is how to target the appropriate individuals without a high degree of searching and list traversals.

The second approach integrates the state of the ecosystem with the processes that affect the state changes. Typically, with this approach the processes are methods of the classes that encapsulate the state of the ecosystem entity. The benefit of this approach is that the process is directly linked with the individual that it needs to act on, and there is no overhead in finding the individual. The drawback to this approach is that it can be more complicated and potentially more risky to experiment with model algorithms and configuration.

There are also two ways that the scheduling of the ecosystem processes is handled. Either the ecosystem entities can be scheduled directly for processing or the processes themselves can be scheduled to run against all relevant entities. Some architectures support both approaches to scheduling. With the first approach, each biological or ecological entity is scheduled to run a specified internal process at a particular time in the simulation. With the second approach, it is the process that is

scheduled and some method of targeting appropriate entities must be provided. The design of the scheduling is closely tied to the design of state and behavior since the effectiveness of each is clearly interrelated.

Chapter 5

A Framework to Support Spatially Explicit Individual-based Population Modeling and Simulation

The solution we provide in this thesis represents a high level architectural design and is intended to provide guidance on building an extensible environment for spatially explicit individual-based population modeling and simulation. The framework should not only support the initial creation of an application, it should also support the evolution of that application. Changes and enhancements to the application code should be at a minimal risk and cost, particularly in a domain that is well known, understood and predictable. This chapter begins with a summary of common requirements in the domain and motivation is given for how the framework design supports these requirements. Then the framework is presented in detail. The details include a class hierarchy diagram, class descriptions, and explanations of how the framework might be extended depending on the requirements of particular applications. We discuss how to instantiate the framework for SORTIE in Chapter 6.

5.1 Domain Requirements

There is a set of fundamental requirements that map from the domain characteristics. These are functional requirements. They are statements about the characterization of applications in the domain. We also list a set of non-functional requirements that map from the reengineering goals. They are statements that motivate the design choices in the

framework to support the functional requirements. These requirements are summarized in Table 5.1 below.

Number	Description	Type
1	Definition of environment using a spatial grid	Functional
2	Tracking of individual states within a population	Functional
3	Ecosystem modeling at multiple scales	Functional
4	Dynamic relationships among individuals	Functional
5	Dynamic relationships between individuals and the environment	Functional
6	Multi-model integration	Functional
7	Maintainability – design must accommodate predictability in application evolution	Non-functional
8	Performance – design must accommodate computationally intensive algorithms	Non-functional
9	Understandability - design must be modular and not overly complex to extend	Non-functional

Table 5.1 Summary of Domain Requirements

In this domain, there is considerable evidence to support the need for multi-model integration [5][7]. Biologists need to experiment with new ideas about ecosystem dynamics and there is value in observing the combined effects of multiple models. With the basic constructs in place such as grid-based tracking of individuals and scheduled processing, integration of a new model could potentially involve creation of new individuals, specification of relationships between these individuals, other individuals and the environment, and implementation of the processes that enact the model. The

individuals and processes might be completely new or they could be extensions of existing ones.

We have described a system with such requirements as an architecture with a pipe-and-filter architectural style. The models are the filters and the pipes allow them to be integrated into the system. With this in mind, the problem then becomes how to support the creation and inclusion of new filters, modification of existing ones and reconfiguration of the way the filters are hooked together.

New theoretical relationships may be discovered as a result of new research. It is desirable if the framework readily supports the testing and expansion of these theories. A programmer will need some way to knowledgeably connect the new filters into the existing structure so that the resulting configuration makes sense in terms of the overall model. This requires that the programmer have a clear understanding of how the existing filters are connected as well as the implications of the current structure so that they can develop an accurate understanding of the program configuration.

As shown in the previous chapter, there are a number of design solutions that meet the functional requirements, however, the choice of design solution is influenced by the constraints of the non-functional requirements. Through the process of domain analysis we learned that the maintenance hotspots are the models themselves and the processes that drive these models to simulate ecosystem dynamics. These processes can target multiple levels in the ecosystem and may take into account the relationships in the environment. The framework described in the next section clearly separates the state of the ecosystem from the processes to accommodate reliable maintenance of existing models and integration of new ones. The levels in the environment are clearly defined

within the framework so that processes can target ecosystem entities at all levels in the same way.

5.2 Framework Description

This framework is purely an architectural design framework. It is not intended to be full-fledged white box framework since it does not provide classes that can be directly extended by the programmer instantiating a specific application. Rather, the intention is to provide an architectural design template for creating applications in this domain. We present a class hierarchy for managing the communication and control mechanisms in the modeling and simulation environment. The classes described below are abstract and are simply for design guidance.

The framework supports the domain aspects that were summarized in the domain description. It provides for the modeling of individuals, both those that are spatially located as well as spatially mobile. It allows the environment to be segmented into spatial cells and supports the modeling of conditions and relationships within that environment. It does not support the modeling of relationships between individuals using explicit constructs, however, we do provide design guidance for how this could be accomplished. The framework also supports scheduled processing at multiple scales by defining a scheduler that can schedule events at any of the scales using the polymorphic characteristics of the class hierarchy. The framework class diagram is shown in Fig. 5.1 below.

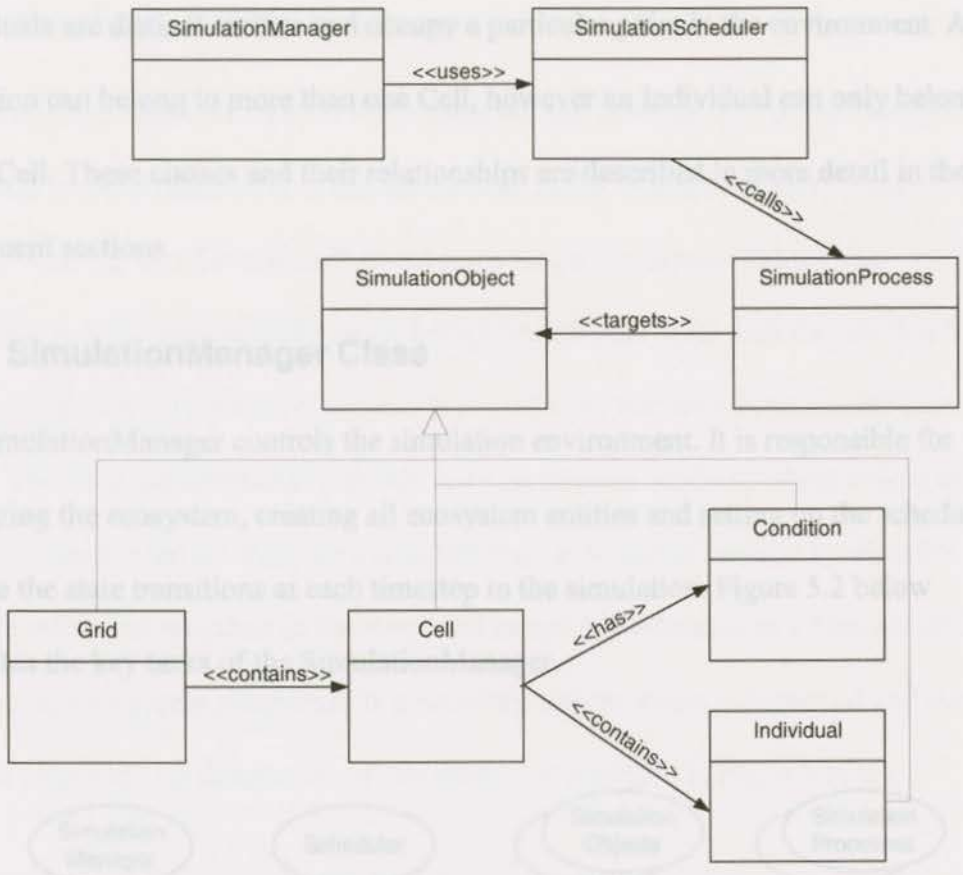


Figure 5.1 Framework Class Diagram

5.2.1 Overview

The SimulationManager controls the simulation environment. It uses the SimulationScheduler to schedule the events to be run during the simulation. The SimulationScheduler contains a queue of events that reference a SimulationProcess and a target SimulationObject. When it is time, the SimulationScheduler invokes the SimulationProcess passing it a pointer to the target SimulationObject. SimulationObjects contain the state information for the ecosystem. They can represent Individuals, Conditions, Cells or an entire Grid. A Grid contains Cells and provides mechanisms for accessing those Cells. A Cell has Conditions and contains Individuals. Conditions are attributes of the environment and occupy an area within the environment whereas

Individuals are distinct entities and occupy a particular point in the environment. A Condition can belong to more than one Cell, however an Individual can only belong to a single Cell. These classes and their relationships are described in more detail in the subsequent sections.

5.2.2 SimulationManager Class

The SimulationManager controls the simulation environment. It is responsible for initializing the ecosystem, creating all ecosystem entities and setting up the scheduler to execute the state transitions at each timestep in the simulation. Figure 5.2 below illustrates the key tasks of the SimulationManager.

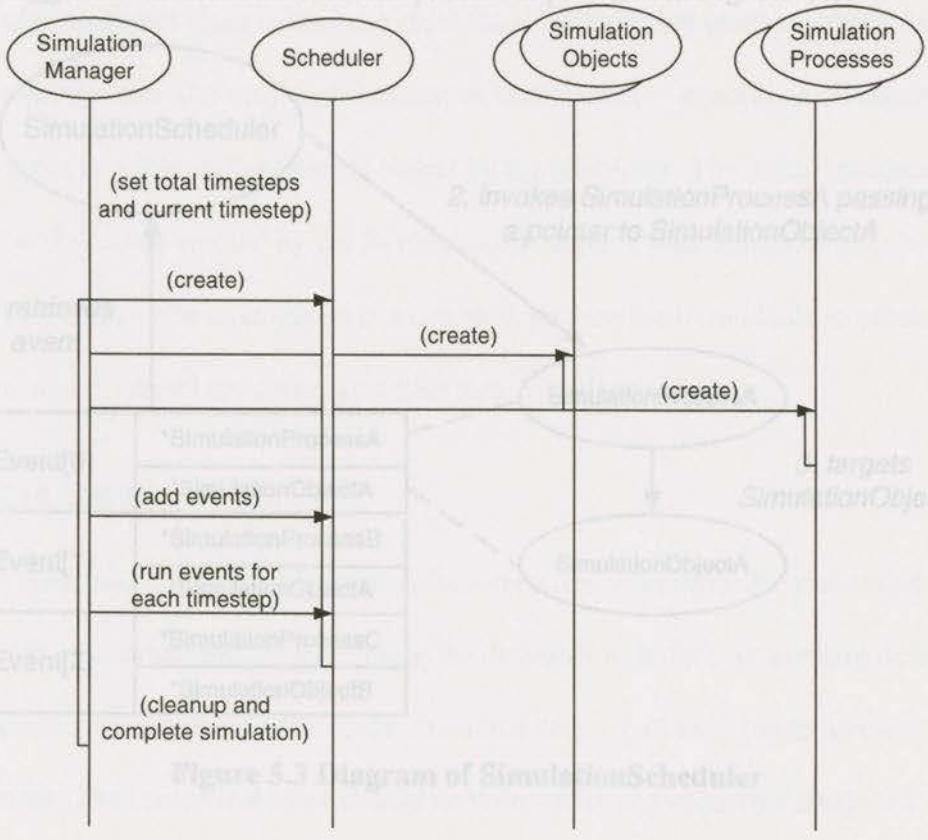


Figure 5.2 SimulationManager Sequence Diagram of Main Tasks

5.2.3 SimulationScheduler Class

The SimulationScheduler handles the scheduling and invocation of the ecosystem processes. Its only function is to handle the scheduling of events at each timestep. An event is made up of a SimulationProcess and a target SimulationObject. The SimulationProcess may actually target SimulationObjects other than the one it is linked to via the event, but only if access to them is provided by the main SimulationObject.

The SimulationScheduler should have an internal queue to which events are added. Events are pulled from the queue and the SimulationScheduler invokes the SimulationProcess specified in the event and passes it a reference to a SimulationObject. The SimulationProcess completes its processing and the event is dequeued and the next event is triggered. An illustration of this process is provided in Fig. 5.3 below.

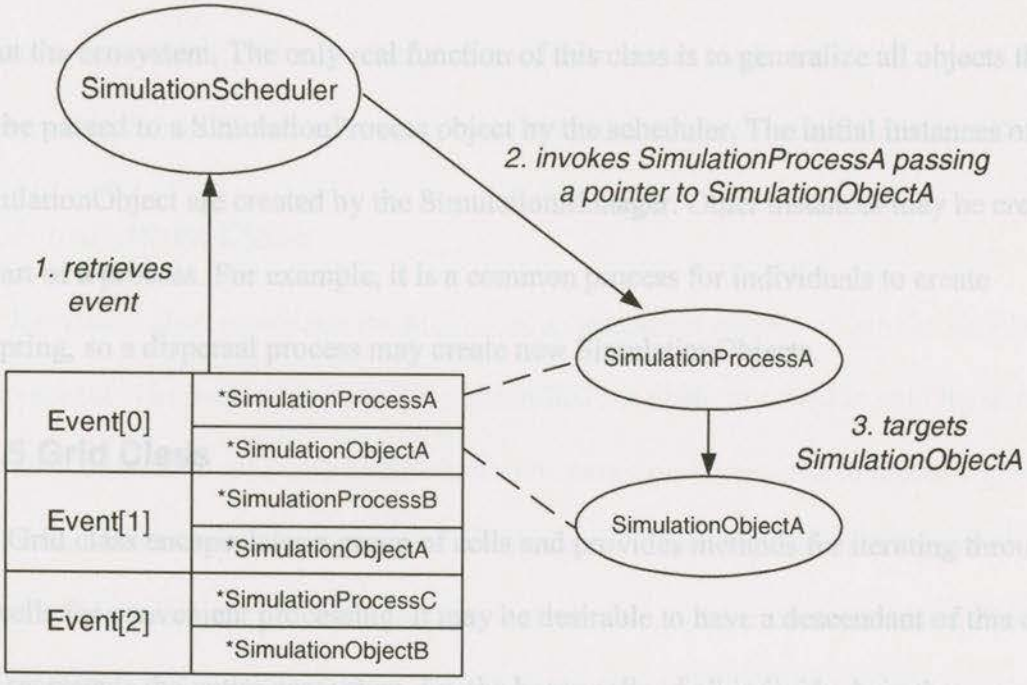


Figure 5.3 Diagram of SimulationScheduler

The SimulationProcess class provides the means for addressing a process within the ecosystem. It is the base class for all objects that perform tasks within the environment. These objects get scheduled by the scheduler to perform their task. SimulationProcess objects should have a common access point, such as an execute() method that accepts a pointer to a target SimulationObject so that they will fit into the rest of the framework. SimulationProcess objects can act on any SimulationObject, so the process may be targeting a specific Individual, a Cell, a Condition or the entire Grid. The code contained within the execute method will be very specific to the scale of the process.

5.2.4 SimulationObject Class

The SimulationObject class is the base class for all objects that contain state information about the ecosystem. The only real function of this class is to generalize all objects that can be passed to a SimulationProcess object by the scheduler. The initial instances of SimulationObject are created by the SimulationManager. Other instances may be created as part of a process. For example, it is a common process for individuals to create offspring, so a dispersal process may create new SimulationObjects.

5.2.5 Grid Class

The Grid class encapsulates a group of cells and provides methods for iterating through the cells for convenient processing. It may be desirable to have a descendant of this class that represents the entire ecosystem, i.e. the home cells of all individuals in the environment. The basic Grid class should contain basic, common area attributes such as plot dimensions. It should also contain variables and iteration mechanisms to provide for

grid traversals. SimulationProcess objects will need to traverse the grid to locate target cells and the conditions and individuals contain therein. Some simulation processes may need to examine all cells contained in the grid, while others may only need to examine cells in a particular subset of the grid. Therefore, positioning mechanisms should be provided in addition to iteration mechanisms.

5.2.6 Cell Class

The Cell class defines a discrete area within the simulation environment. The attributes of a Cell are generalized in such a way so that external objects have a convenient and consistent way to access them. A Cell doesn't necessarily have to be a two-dimensional representation of space, however, this is the most common requirement in spatially explicit modeling environments. The basic Cell class should contain basic common attributes such as cell width and length. It also contains an array of Individuals that are contained within a cell and an array of Conditions that intersect with the Cell coordinates.

5.2.7 Individual Class

The Individual class represents the biological or ecological entities within the simulation environment. The basic processes of an individual are likely to coincide with the natural lifecycle common to all living things: birth, consumption of resource to produce growth, producing offspring and death. Some individuals will require spatial mobility while others will not. It is up to the programmer to determine whether or not the Individual base class should support spatial mobility or not. In any case, an individual is spatially located, both at an absolute coordinate and in a particular cell, at any snapshot in the simulation time. It may be desirable to create a subclass of the Individual base class to support

spatial mobility and to provide methods for handling the individual's movement within the simulation environment.

5.2.8 Condition Class

The Condition class represents a phenomenon within the environment and is included to support conditions that may not necessarily coincide with the segmentation of the Cells. Conditions apply to a space in the environment. Rather than having a singular absolute coordinate, they have a coordinate range. For instance, a condition could start at x=10 and y=10, but it extends to cover the area up to x=25 and y=25. Since the conditions do not line up exactly with the cell division, it is possible for a cell to have two or more values for the same condition type.

5.3 Framework Extension

There are some fairly large design decisions here that are left up to the programmer or designer instantiating their application using the framework. These include issues related to the high-level program control, how to define the simulation objects and processes and how the processes target the appropriate scale in the ecosystem.

5.3.1 Design Considerations in the SimulationManager

The SimulationManager is described as having control over the simulation environment. Thus far we have only described this control in terms of the simulation run, however, there are other issues that should be considered such as how to define or initialize the ecosystem, how to accommodate user interactivity and how to integrate visualizations of the ecosystem dynamics.

Ecosystem Definition

Ecosystem models rely on real world data to make accurate predictions. Typically, ecosystem definition data is stored in some kind of database format whether it be flat data files or tables in a database management system. It might also be the case that the user defines the ecosystem at runtime by filling in a form that embodies the possible characteristics of the ecosystem. The format or composition of the data varies widely from system to system, however, in all systems the requirement to define the ecosystem exists. Standard formats for data such as XML [29] can be useful since they have built in support for checking the validity of the data, but in practicality the research data could be available in any format. In many cases it is quicker to build a custom component to import the data as is.

As mentioned above, the overall state of the ecosystem is constructed from the state of all the entities that make up that ecosystem. The start state of the entities in the ecosystem must be initialized before the simulation environment can effect change on the overall ecosystem. The responsibility of ecosystem initialization is to translate the ecosystem definition and create the objects that represent those entities within the modeling and simulation environment. How exactly each system accomplishes this initialization depends on the structure of the entities in the ecosystem being modeled and the nature of available research data.

User Interactivity

Users of ecosystem modeling and simulation environments make adjustments to the ecosystem between runs to determine relevant impact. For example, the user might want to see what effect increasing the density of a certain population had on the overall

ecosystem. These adjustments can be seen as refinement or redefinition of the original ecosystem definition. With this in mind, the following interactivity options should be considered:

- Give the user re-initialization capabilities, i.e. allow the user to initialize the ecosystem more than once without closing the program
- Allow the user to set up the targets for data export and export timing
- Allow the user to select from a set of simulation visualizations
- Give the user the ability to manipulate the simulation run while it is running (options such as pause, stop, rewind, and fast-forward may be considered)
- Give the user the ability to re-run the simulation and give them the option to override previously exported data or to create a new data set (with some implementations, especially if the models contain a certain degree of randomness, comparison of the model predictions from multiple runs using the same initial data set may be desirable)

Visualizations

Robust simulation systems record the state of the ecosystem at each time-step or set intervals so that the prediction data can be analyzed, graphed or exported. The data recorded at each time-step may be the full ecosystem state or it may be a subset depending on the output requirements. At a minimum, the last state at the completion of the run is generally of interest to the users of these tools. Although it is also common for the required output data formats to differ from the formats used for ecosystem initialization.

Although it is possible to simply save the data to a flat file or database for later use and analysis, users of modeling and simulation systems benefit from the ability to produce graphical representations of the model's predictions at various stages in the simulation. This can be done during the model run or created afterwards using recorded data. This functionality helps the user understand how data is transformed by the system. Different users require different visualizations [29]. Users with an understanding of the employed models might only require a set of simple graphs or charts to see patterns in the model predictions or compare model results. For users with limited understanding of the models a three-dimensional diagrammatic image of the ecosystem state might be more appropriate. Even within the same implementation, the visualization requirements will likely change over time. A good modeling and simulation system should separate the graphical components in such a way that new visualizations can be added without disrupting the rest of the system.

5.3.2 Design Considerations in the SimulationScheduler

The specific instantiation details of the SimulationScheduler will depend on the technology and programming language used for the target implementation. For instance, if standard C++ is used, the SimulationScheduler schedules events that contain pointers to SimulationProcesses and target SimulationObjects. With this approach, the programmer is concerned with dynamic binding issues. However, it may be the case that the implementation requires a distributed computing environment or the integration of a particular technology such as agents.

CORBA [19] can be useful if the target implementation is to be distributed across a network. CORBA applications are composed of individual units of executable software

that combine functionality and data. CORBA uses a standard protocol (IIOP) to control the execution of these software units so that objects in a number of programming languages can communicate in a standard way. The use of agent-based technology is explored in more detail in Chapter 7.

The technology used in the specific instantiation will have an impact on the implementation details of all classes in the framework, however, since the SimulationScheduler is the "glue" that ties the core model components together, it will have the largest impact here. How objects are called and how the methods provided by those objects are invoked will depend on the technology used.

5.3.3 Creating and Extending SimulationProcess Classes

The same process could be implemented at the Grid, Cell or Individual level. It is up to the programmer to decide. For instance, an individual growth process could target a grid whereby the process would need to iterate through the grid, examining the individuals in each cell and calculating their individual growth. This process is illustrated in Fig. 5.4 below.

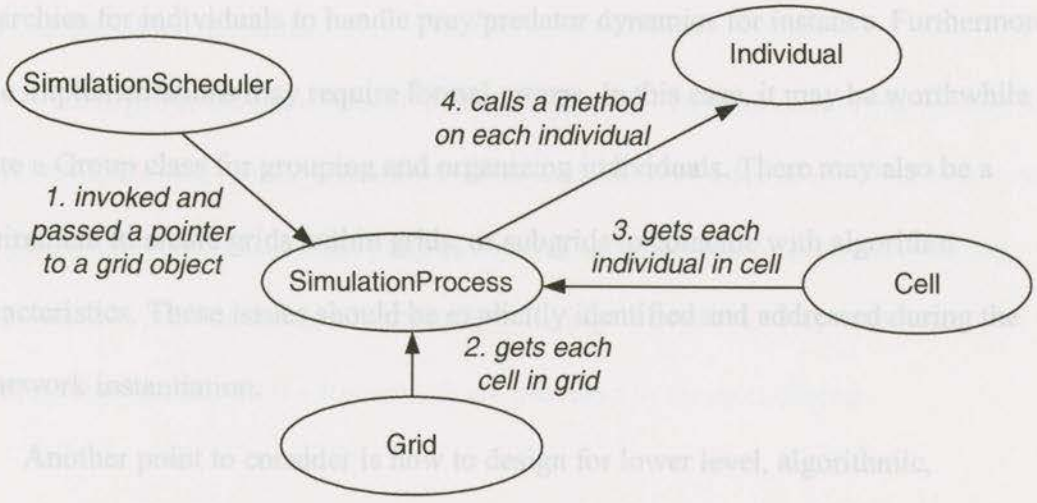


Figure 5.4 SimulationProcess Targeting Individuals via Grid Object

Conversely, an individual growth process could target the individual directly. That is to say that all the individuals would be queued up for individual processing. If there are a number of growth processes, depending on attributes such as species and lifestage, then there are some additional complexities in targeting appropriate individuals with the approach shown in Fig. 5.4. It may be worthwhile for the programmer to carefully think about the various process scales that will be required by their application so that common processing can be isolated in methods that can be reused via an inheritance hierarchy.

5.3.4 Creating and Extending SimulationObject Classes

We learned from our analysis of domain legacy that ecosystem entities are modeled in terms of state and behavior. Our framework provides the SimulationObject super-class for all ecosystem entities so that they can be integrated into the system in a consistent way. We have also described some basic entities that will likely be required by applications in this domain. However, the programmer instantiating the framework will be faced with some important design decisions. Most notably is the determination of how to structure the ecosystem entities. Some models may require complex organizational hierarchies for individuals to handle prey/predator dynamics for instance. Furthermore, some implementations may require formal groups. In this case, it may be worthwhile to create a Group class for grouping and organizing individuals. There may also be a requirement to create grids within grids, or subgrids to coincide with algorithm characteristics. These issues should be explicitly identified and addressed during the framework instantiation.

Another point to consider is how to design for lower level, algorithmic, extensibility. Class attributes capture the state information and the behaviors are captured

in methods. The behaviors and state information may be encapsulated together whereby the entity is represented as a class with attributes and methods for changing those attributes. For greater extensibility and flexibility, sometimes these methods are simply "hooks" to external objects that contains the actual algorithm. We have also seen designs where the state and behavior are completely separated. In this case, the behavior, or the process, simply requests the individual and performs a read/write operation on it. How to specifically model this has been left up to the programmer instantiating the framework.

5.4 Summary

Our solution is presented in terms of a high-level architectural framework and reflects the knowledge gained during the domain analysis. As we have indicated, there are a number of detailed design decisions that are left up to the programmer instantiating the framework. We have presented some of the key issues and design considerations that are likely to come out of the detailed design process. The goal of the framework is to speed up the development of applications in this domain by highlighting common functional requirements, constraints produced by typical non-functional requirements, key design issues and possible solutions.

Throughout the development of the framework, we were able to experiment with design solutions and identify issues specifically related to SORTIE in collaboration with the application stakeholders and programmers. The framework is continually being evaluated as it is being applied to the reengineering of SORTIE. The details of the SORTIE instantiation of the framework are provided in the next chapter.

Chapter 6

Framework Instantiation

This chapter describes the core SORTIE model in terms of the framework and details suggested classes, their relationships, communication and coordination. Our understanding of the SORTIE model and code requirements comes from the manual reverse engineering of the existing code base and discussions with SORTIE stakeholders and programmers.

As the framework was being constructed, different design concepts from the domain analysis were evaluated using the SORTIE model. A design idea was presented and then we attempted to describe a subset of SORTIE in terms of that design. Designs that proved to be too complicated or simply didn't fit adequately with the SORTIE model were rejected. In this way, the design of the framework was iterative and SORTIE was used in the formative evaluation stages. That is, SORTIE was used to guide changes in the framework design.

The details presented in this chapter are meant as a guideline for how the SORTIE program can be reengineered. The model is described at a high level and details on the algorithms are not included, however, conceptual illustrations and descriptions are provided. The ongoing reengineering constitutes a summative evaluation of the framework design and certainly there are still lessons that could be learned from this process.

6.1 Framework Instantiation

This section describes how SORTIE could be reimplemented using the proposed framework. We describe class interactions and how the user tasks in the SORTIE program could be implemented using the framework. The core framework classes remain the same as they have been described in the previous chapter. More details and pseudocode are provided in Appendix A.

As mentioned, the primary reengineering goals are to: increase code maintainability by supporting the modification of existing models and the incorporation of new ones without compromising the reliability of the program; to improve code understandability by increasing modularity and reducing complexity so the code can be maintained in a conceptually straightforward manner; and to improve application usability by increasing programming stability and maintaining current levels of performance. A key constraint on the new design is that the existing development environment cannot be replaced and where possible, small-scale reuse of code is desired.

6.1.1 SimulationObject Classes

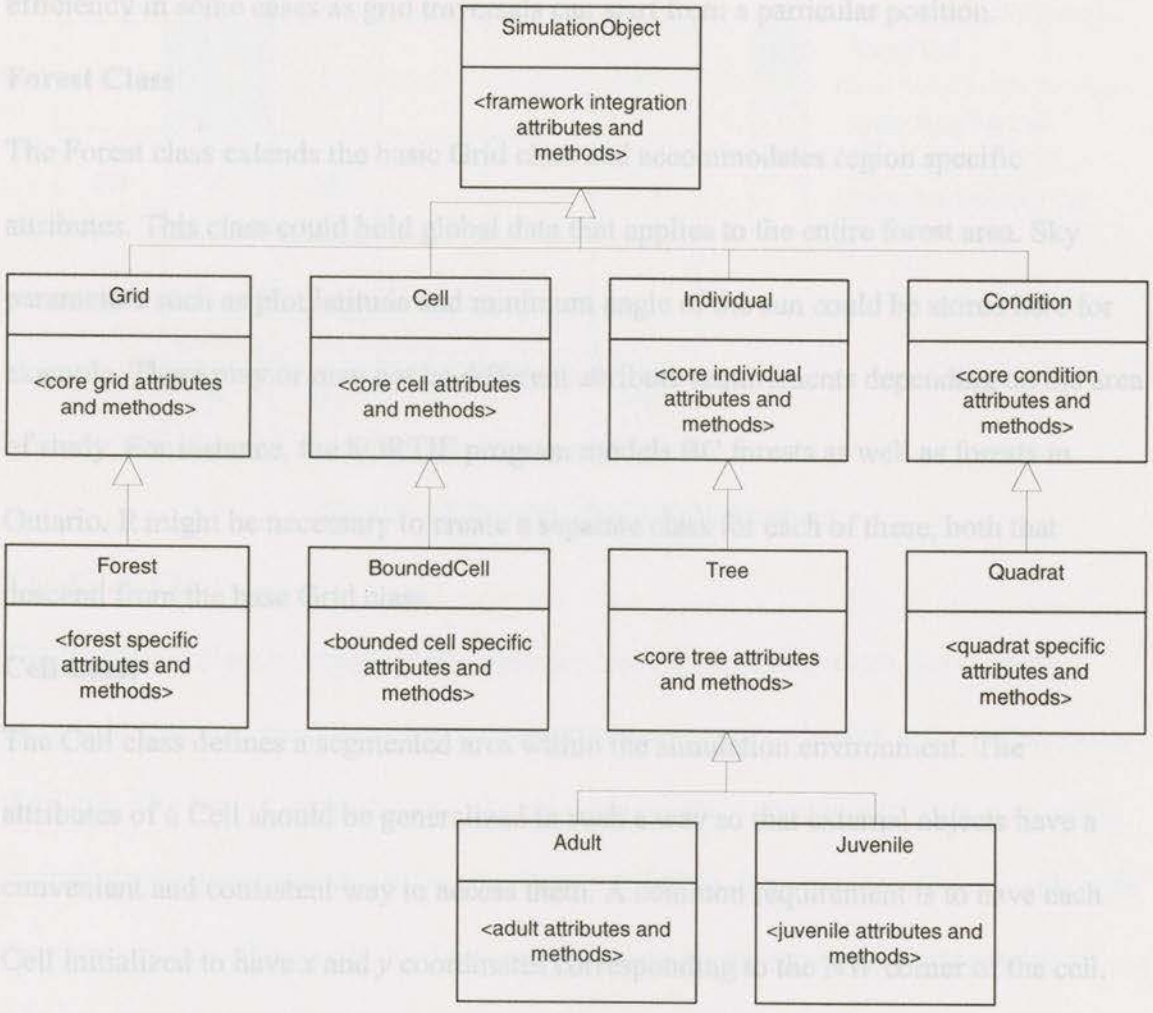


Figure 6.1 SimulationObject Class Hierarchy

Grid Class

The Grid class encapsulates a group of cells and provides methods for iterating through the cells for convenient processing. Some kind of initialization mechanism is required to set the number of cells and their dimensions. Iteration mechanisms would also be useful to allow for generic grid traversal. Row-wise (EW) and column-wise (NS) are commonly required. It might be useful to provide a general positioning method that sets the grid to a

particular cell position based on a coordinate parameter. This would allow for greater efficiency in some cases as grid traversals can start from a particular position.

Forest Class

The Forest class extends the basic Grid class and accommodates region specific attributes. This class could hold global data that applies to the entire forest area. Sky parameters such as plot latitude and minimum angle of the sun could be stored here for example. There may or may not be different attribute requirements depending on the area of study. For instance, the SORTIE program models BC forests as well as forests in Ontario. It might be necessary to create a separate class for each of these, both that descend from the base Grid class.

Cell Class

The Cell class defines a segmented area within the simulation environment. The attributes of a Cell should be generalized in such a way so that external objects have a convenient and consistent way to access them. A common requirement is to have each Cell initialized to have x and y coordinates corresponding to the NW corner of the cell. The absolute coordinate range of the cell may be useful to some processes, particularly ones that affect Conditions.

BorderedCell Class

The BorderedCell class extends the basic Cell class and accommodates commonly needed border relationships among cells. It is often useful to know which cells share a common border. In SORTIE the borders of the boundary cells, the cells at the edges of the grid, are wrapped such that the grid appears continuous. These cell border relationships are illustrated in Fig. 6.2 below.

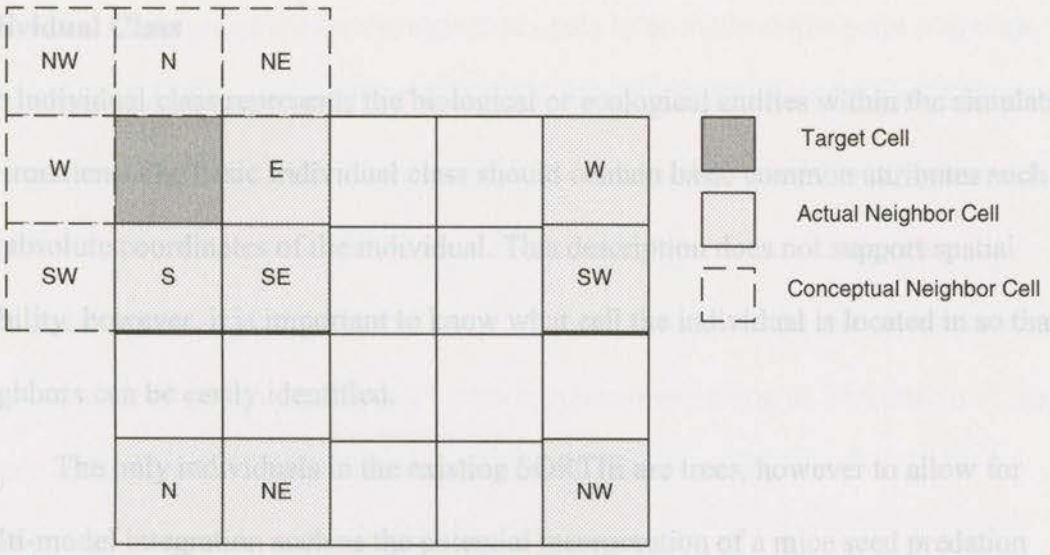


Figure 6.2 BorderedCell Relationships

Condition Class

The Condition class represents a phenomenon within the environment and is really included to support conditions that may not necessarily coincide with the segmentation of the Cells. The basic Condition class should contain basic common attributes such as the coordinate range cell width and length. Since conditions do not necessarily coincide with the segmentation of the cells, the basic condition class does not support knowledge of which cell or cells the condition occupies. A condition can be thought of as a condition or partial condition of a cell and is referenced via the cell.

Quadrat Class

The Quadrat class extends the basic Condition class and accommodates attributes and methods that are needed for quadrat light. Quadrats are user-defined subgrids in the grid, usually two metres square, hence the name "quadrat". These subgrids contain light information used by seedlings to determine the effect of light on their growth. This process is described in more detail below in the QuadratLight section under SimulationProcesses in 6.1.2.

Individual Class

The Individual class represents the biological or ecological entities within the simulation environment. The basic Individual class should contain basic common attributes such as the absolute coordinates of the individual. This description does not support spatial mobility, however, it is important to know what cell the individual is located in so that neighbors can be easily identified.

The only individuals in the existing SORTIE are trees, however to allow for multi-model integration such as the potential incorporation of a mice seed predation model, the individual class should be created. There are four submodels that drive the primary processing of these individuals: Resource, Recruitment, Growth and Mortality. Each submodel consists of a number of species and lifestage specific algorithms. For example, the growth algorithm differs depending on the species of the tree and whether or not it is an adult or juvenile. One of the hardest design decisions to be made is how to model these trees as individuals and associated processes. Since the algorithms are lifestage and species specific, it is advantageous to model each type of individual, defined by lifestage and species, as a separate class with unique attributes and methods for that particular type of tree. Using an inheritance hierarchy, common attributes can be defined in a parent class. The algorithms that drive the individuals behaviors can be directly implemented in the class for simplicity and maintainability. This approach also eliminates the need for redundant attributes since a distinct class is created to hold the attributes of an adult tree of a particular species. A potential downside to this approach is that there needs to be some mechanism to handle lifestage transitioning. When a tree grows from a juvenile into an adult it would need to create an adult version.

The other major design decision that needs to be made at this point is at what level to schedule the processes that relate to individual behaviors. A simple and manageable way to design this is to model each high level submodel as a process that targets the entire forest area. The process iterates through the grid, calling the corresponding method on the individual. Using polymorphism, the same method can be called on all individuals regardless of what subclass they belong to. The design of these processes is discussed in the SimulationProcess Classes section.

Tree Class

The Tree class extends the basic Individual class and accommodates attributes and methods that are common to all tree types. Variables such as tree height, distribution at breast height, canopy radius and growth rate could be stored here for example. Any methods common to all trees should be located here as well so they can be inherited by subclasses.

AdultTree Class

The AdultTree class extends the Tree class and accommodates attributes and methods that are common to adult trees. An individual adult tree uses information that is common to all adult trees such as the percentage of sun, the maximum sun a tree could receive as well as the brightness and foto arrays. Since this information is needed by all trees, but is the same for all trees, it should be passed to the adult tree so that it can calculate the amount of shade it is experiencing. It may be worthwhile to create an object that contains this information and then a reference to that object could be passed instead.

JuvenileTree Class

The JuvenileTree class extends the Tree class and accommodates attributes and methods that are common to juvenile trees. An individual juvenile tree, a seedling, must be able to query a cell to determine which quadrat it is located in so that it can request the amount of light in the quadrat.

6.1.2 SimulationProcess Classes

The basic SORTIE model consists of four main submodels: resource, recruitment, growth and mortality. However, these don't map uniquely to processes within this framework. The resource submodel is split into two processes: QuadratLight and Shading. QuadratLight calculates the light in an area and is used by seedlings to determine growth. Shading calculates the amount of shade an individual experiences from taller individuals in its neighborhood. These processes are described in more detail below.

All of the processes in this instantiation of the framework are grid level processes. That is, all of the processes accept a Grid object as their target simulation object and iterate through the grid processing individuals or conditions as the case may be. This design maps closely to the current design of SORTIE and will make it easier to do the reengineering since the algorithms are defined to iterate in this manner and increases the likelihood of code reuse.

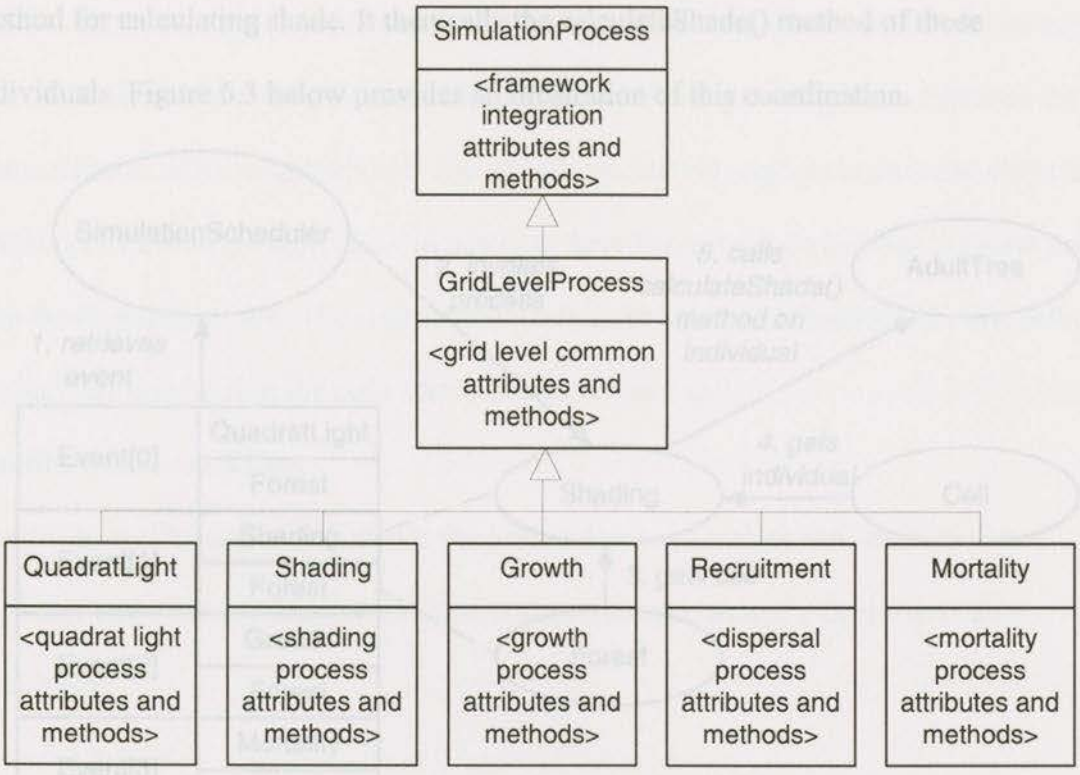


Figure 6.2 SimulationProcess Class Hierarchy

The existing SORTIE program also includes processing for harvesting, planting and natural disturbances such as windstorms. These processes could be designed in a similar manner to the ones described in detail here, but have not been included in the case study due to time constraints.

Each of the simulation processes accepts a grid object. In this case all of the processes deal with the Forest object and iterate through the grid cells calling the appropriate method on the final target object, either an individual or a condition. For example, the shading process targets adult trees, but the actual process object is queued with the forest grid object. Consequently, the shading process object must be able to iterate through the grid, examining cells and requesting the individuals. It must also be able to determine which individuals are adults, because only the adult trees will have a

method for calculating shade. It then calls the calculateShade() method of those individuals. Figure 6.3 below provides an illustration of this coordination.

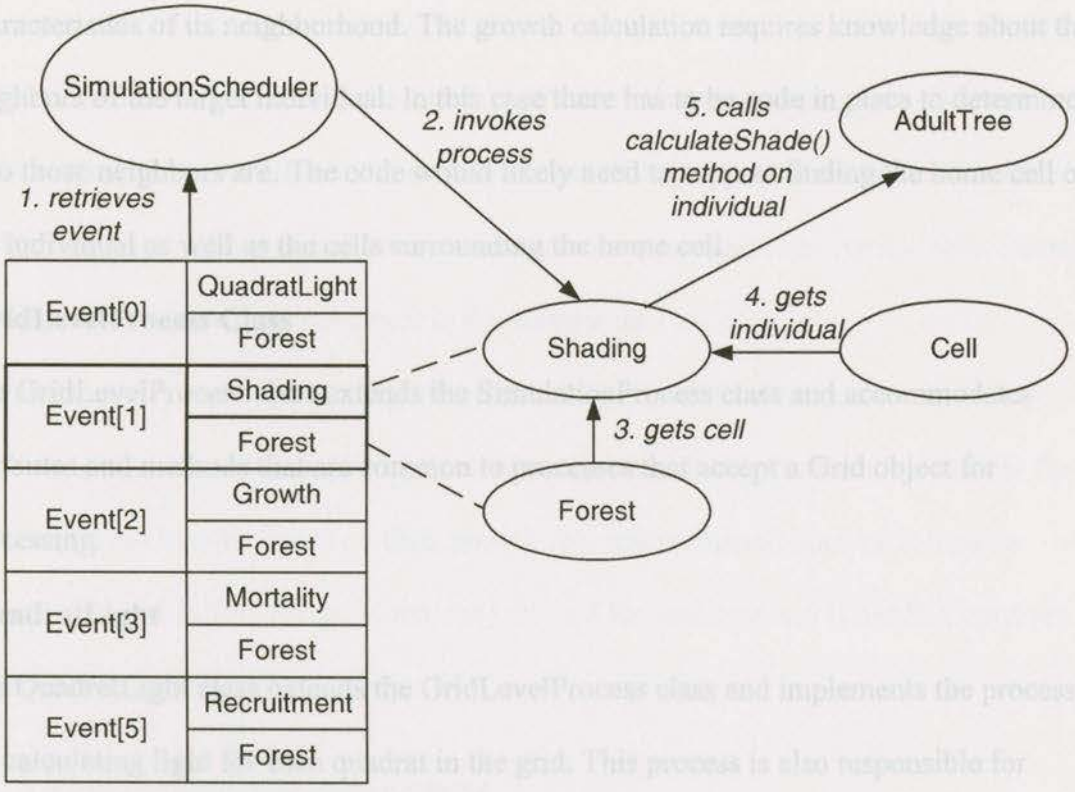


Figure 6.3 Process Coordination Example

SimulationProcess Class

The SimulationProcess class provides the means for addressing a process within the ecosystem. It is the base class for all objects that perform tasks within the environment. These objects get scheduled by the scheduler to perform their task.

There will likely be many descendants from the base SimulationProcess class in an instantiation of the framework. They all must have a public method, such as execute(), so that they will fit into the rest of the framework. SimulationProcess objects can act on any SimulationObject, so the process may be targeting a specific Individual, a Cell, a

Condition or the entire Grid. The code contained within the execute method will be very specific to the scale of the process. Consider an individual growth process based on the characteristics of its neighborhood. The growth calculation requires knowledge about the neighbors of the target individual. In this case there has to be code in place to determine who those neighbors are. The code would likely need to support finding the home cell of the individual as well as the cells surrounding the home cell.

GridLevelProcess Class

The GridLevelProcess class extends the SimulationProcess class and accommodates attributes and methods that are common to processes that accept a Grid object for processing.

QuadratLight

The QuadratLight class extends the GridLevelProcess class and implements the process for calculating light for each quadrat in the grid. This process is also responsible for handling the bath light feature in SORTIE that functions to alter quadrat light. It is an episodic event and uses random numbers to change the quadrat light level.

Shading

The Shading class extends the GridLevelProcess class and implements the process for calculating the amount of shade each individual adult tree experiences as a result of its taller neighbors.

Growth

The Growth class extends the GridLevelProcess class and implements the process for growing each individual tree in the grid. It calls the grow() method on each tree

regardless of lifestage and species. However, the grow() method implementation in each subclass of tree will depend on these characteristics.

Recruitment

The Recruitment class extends the GridLevelProcess class and implements the process for recruiting new seedlings in the grid. It iterates through the grid calling the disperse method for each adult tree and determines which new seedlings survive and then creates new juvenile trees to represent them in the simulation.

Mortality

The Mortality class extends the GridLevelProcess class and implements the process for removing trees that die based on their growth rate and stochiastic factors. It iterates through the grid calling the getMortality() method for each tree and if needed, removes the dead tree from the grid.

6.1.3 SimulationScheduler Class

The SimulationScheduler handles the scheduling and invocation of the ecosystem processes. Its only function is to handle the scheduling of events, SimulationProcesses and their target SimulationObjects, at each timestep. It should contain basic public methods for adding and running events. When the SimulationScheduler is created, it should set the total runLength of the simulation schedule. That way, when events are added, the SimulationScheduler can check to see if the requested runtime fits within the runlength – that is the event is not out of the simulation time frame.

SimulationEvent is an internal class of SimulationScheduler. It is only needed and used within SimulationScheduler. When the addEvent() method is called, an event is created and added to the queue of events. The programmer decides whether or not to

order the queue according to runtime upon event insertion. If the event is successfully added into the queue, the addEvent() method should return a boolean status of True. When the runEvents() method is called, it is passed the current time. The SimulationScheduler should run through the queue of events looking for events where the event time is less than or equal to the current time. When it finds matching events, it should call the execute() method of the SimulationProcess, passing it the pointer to the SimulationObject as a parameter and deleting the event from the queue. It is probably prudent to create some private methods to handle the queuing and dequeuing of events.

6.1.4 SimulationManager Class

The SimulationManager in SORTIE controls the simulation environment. It is responsible for creating all ecosystem entities and setting up the scheduler to execute the state transitions at each timestep in the simulation. Figure 6.4 below illustrates the major states embodied by the SimulationManager.

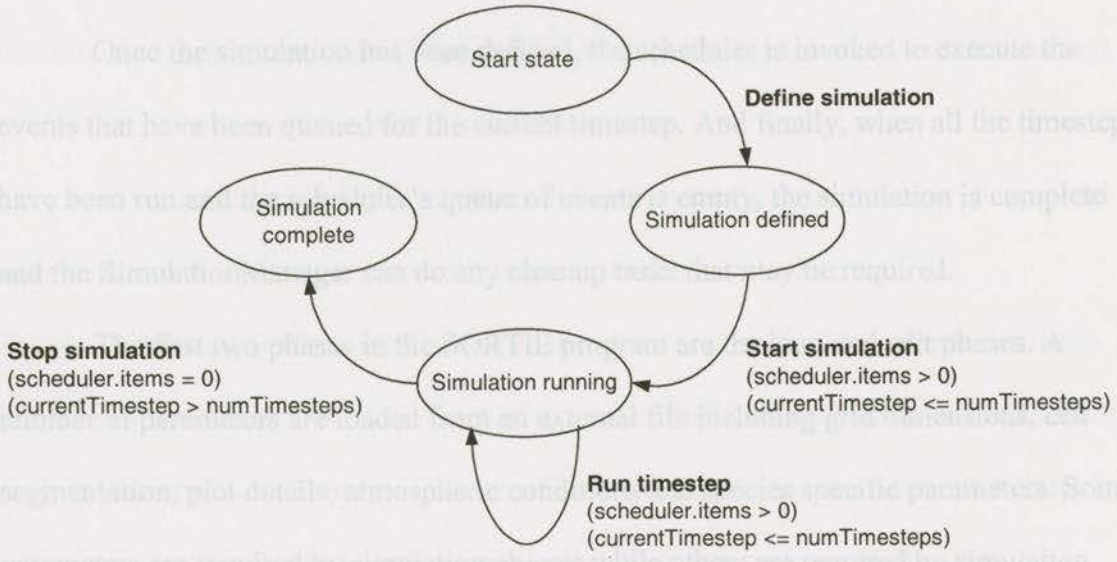


Figure 6.4 Simulation Manager States

The SimulationManager has three main tasks. First, it should define the simulation. This includes initializing the number of timesteps and setting the current timestep. The definition step should also create the entities and processes that make up the ecosystem being simulated and then create their associated events to be run by the scheduler. Although not necessary for correct operation of this framework, we suggest a high level SimulationObject that captures the entire simulation environment and provide a public method so that external objects can request a pointer to it. The SimulationObject could be a Grid object that contains pointers to the home Cells of all the Individuals in the ecosystem. That way an external module such as one that handles visualizations can gain access to the entities and attributes of the simulation environment. Alternatively, the state of the ecosystem could be written into an XML file at various points in the simulation. The visualizations could then access this file to read the necessary ecosystem definition. This approach would allow for greater flexibility since it is a true decoupling of the GUI elements from the core components.

Once the simulation has been defined, the scheduler is invoked to execute the events that have been queued for the current timestep. And finally, when all the timesteps have been run and the scheduler's queue of events is empty, the simulation is complete and the SimulationManager can do any cleanup tasks that may be required.

The first two phases in the SORTIE program are the load and edit phases. A number of parameters are loaded from an external file including grid dimensions, cell segmentation, plot details, atmospheric conditions and species specific parameters. Some parameters are required by simulation objects while others are required by simulation processes. It is recommended that the input/output and parameter file translation

mechanisms be encapsulated within a separate class for ease of maintenance. The current parameter file is a simple fixed position text file and the parameters are read in sequentially. A more robust solution would be to make this parameter file a machine-readable format such as XML.

The program provides numerous graphical user interfaces for editing the parameters that define the simulation environment. Users can edit most of the parameters mentioned above. It is important for extensibility and portability that the user interface components be clearly and distinctly separated from the core application code. It is typical for user interface elements to be platform and compiler specific. The user interface should be seen as a layer that acts on top of the framework. The SimulationManager class should provide hooks to support these program phases. It is recommended that the SimulationManager be initialized using a consistent format, some kind of simulation definition object that packages the user specified parameters. That way, if the parameter file format or user interfaces change, the mechanism for communicating the simulation parameters to the SimulationManager remains fixed and it is only the external modules that need modification.

The third and fourth phases are the run and display phases. The run phase is mostly related to the core framework components and is driven by the SimulationManager calling the SimulationScheduler at each timestep. However, these two phases are closely tied since the display phase handles the visualization requirements such as plot representations, graphs and tables reflecting key parts of the simulation at each timestep. The SimulationManager needs to be able to respond to display requirements at each step. Probably the simplest solution is to provide a public method

for accessing the Forest object so that the visualizations can simply read the required information.

6.2 Summary

The design of the architecture closely matches the conceptual model of SORTIE. The environment is explicitly segmented into cells. These cells contain individual trees and have conditions that represent distinct attributes of the environment such as light availability. We have suggested grid-level processes to mirror the main submodels in SORTIE since the existing algorithms involve similar grid-based processing. Using a similar approach increases the likelihood of code reuse. These high-level processes iterate through the grid calling a common method on the individuals. For instance, the growth process iterates through the grid calling the grow() method of every individual. The implementation of the grow() method differs depending on the type of individual such as whether it is an adult or juvenile tree. The individual type-specific behaviors are tied to the individual, but common attributes and methods are contained within the tree superclass.

In Chapter 2 we described SORTIE as having a pipe-and-filter data flow architectural style. This first level of pipes and filters represents the high level relationships in the SORTIE model: resources are a function of recruitment, growth is a function of resources and mortality is a function of growth. We support these relationships through the scheduler by allowing the submodels to be scheduled sequentially to perform successive transformation of ecosystem data. The second level represents the different functions required in each submodel depending on the target data. We recommend that these differences be encapsulated with the data and suggest

providing public methods on the target simulation objects to initialize data, adjust algorithm parameters and to handle execution of type-specific algorithms.

The framework provides design guidance for the core model functionality in SORTIE. There are many other issues that need to be addressed in the reengineering process such as support for interactivity and visualizations. We summarized some of these issues in the previous chapter from a broad perspective and then more specifically related to SORTIE in this chapter. However, we have not explicitly addressed the needs of the secondary users of SORTIE, the resource managers who use the program in sustainable resource management to predict the effects of planting and harvesting regimes. These management regimes are also simulation processes and can be integrated into the system in the same way the processes that encapsulate natural ecosystem dynamics. In the next chapter we demonstrate how this can be done while showing how agent technology can be integrated into the framework to satisfy typical sustainable resource management requirements.

Chapter 7

An Agent-based Approach

In this chapter we explore work related to the broader context of ecosystem modeling and simulation. To do this, we first need to ask ourselves what research in this domain contributes to society and the greater body of knowledge. Ecosystem modeling and simulation environments allow biologists to explore ideas about natural ecosystem dynamics and this is valuable knowledge on its own. However, this knowledge also allows us to explore ideas about how to manage our ecosystems that contain a wealth of natural resources. With this in mind, it becomes obvious that it is not only important to consider how an application for ecosystem modeling and simulation can be designed for extensibility, but also how to integrate these systems and the models contained therein with models for sustainable resource management. To meet this end, it is important that we look at current research in technologies to support sustainable resource management.

It has been demonstrated that agent-based modeling and simulation can be an effective and powerful tool for exploring the dynamics of ecosystems [14]. Of particular interest is the use of agent-based modeling and simulation for sustainable resource management. If we can build tools that allow resource managers to observe the structure, function, composition and natural changes of ecosystems they can then learn from these and create management practices to mimic them [15]. When applied to ecosystem management, there are many interesting research challenges yet to overcome. Application of agent-based modeling to real problems will help researchers uncover ways to minimize

or overcome these challenges thus building better tools for exploring management practices.

As part of this study, we also explored the use of agent-based technology in the reengineering of SORTIE. The use of agent technology is relatively new and we were only able to find one toolkit for guidance. We looked at a multiagent simulation toolkit called CORMAS [14] to gain insight as to how an agent-based architecture to support spatially explicit individual-based population might be designed and discovered that there is a lot of overlap with how the ecosystem entities in non-agent based architectures are structured. A subset of the SORTIE model was implemented using the Madkit agent platform [20].

In the sections below we provide an overview of agent-based modeling and simulation, summarize the design of the CORMAS framework and present the results of our exploratory research, including the high-level architecture and encountered problems.

7.1 Agent-based Modeling and Simulation

There are many advantages of agent-based modeling and simulation over the more conventional systems dynamics type of modeling [16]. Agent-based models, also called M*ulti-Agent Systems* (MAS), allow us to

- Model heterogeneous actors
- Model links between the micro and the macro levels and study emergent phenomena
- Model the effects of interactions among agents
- Model individual cognition and/or attitudes or beliefs, and observe their effects in the aggregate

Individual-based models are a subset of multi-agent systems that includes any computational system whose design is fundamentally composed of a collection of interacting parts. For example an "expert system" might be composed of many distinct bits of advice that interact to produce a solution. Individual-based models are distinguished by the fact that each "agent" corresponds to autonomous individual in the simulated domain. These agents are autonomous computational entities that communicate with each other via an established communication protocol [8]. There are two key benefits to this autonomy and communication mechanism. First, this allows for a relatively straightforward substitution of system components. New agents with different roles and responsibilities can easily be added to the system as long as they follow the established communication protocol. Second, this architecture naturally lends itself to distributed environments and parallel processing which can be extremely useful with respect to the computational requirements of many models.

The fundamental problem with modeling common resource management is how to simulate the interaction among groups of agents and resource dynamics [8]. Realistic modeling of resource management requires a multidisciplinary approach, relying on expertise in forestry, landscape architecture, computer science, environmental psychology and planning. This requirement indicates a need for collaborative research and an iterative development process.

7.2 CORMAS Framework

CORMAS (Common-pool Resources and Multiagent Systems) is a multiagent simulation toolkit to model natural and social dynamics at multiple scales [14]. It is based on the premise that ecosystem modeling and simulation environments need to be able to

integrate models at different scales. It attempts to provide an environment that makes the process of model integration straightforward through the use of a toolkit implemented in Smalltalk.

The CORMAS framework supports spatially explicit individual population modeling. The framework includes base classes for spatial entities in a grid-based environment. It provides support for spatially located individuals, including the ability to move about and dynamically determine the neighborhood of an individual.

The framework uses agents to represent entities at various scales. Both individuals and social groups are modeled as agents. Since agents intrinsically support the entity requirements, the use of agents in this capacity is a natural fit. Agents have internal data representations that contain state information. They also contain the necessary logic to interact with their environment and modify these internal data representations as a result of the interaction. This constitutes agent, or entity, behavior. As we have seen throughout the ecosystem modeling and simulation literature, the modeling of entity state and behavior is a core design problem. Using agents encapsulates the state information with the behavior since agents are autonomous and require the capacity to reason and make decisions about what to do in response to what is perceived about the environment.

Agents can represent a microscopic level, individuals, or they can represent a macroscopic level, groups or macro agents. In the case of forest ecosystems, an individual at the microscopic level is the tree, while the macroscopic level is the forest. Some phenomena occur at the microscopic level while other phenomena occur at the macroscopic level. When the real world is modeled, the complexity of the dynamics

result in a highly interconnected hierarchy. The CORMAS framework alleges to support this complexity.

A benefit of using agents to represent the ecosystem entities is that they communicate via an established communication protocol. Agents are autonomous software entities. They are programs that run independently of a central control system. This obviously lends itself to distributed computing and allows for parallel processing which can be important for performance particularly in large scale systems with thousands of individuals performing numerous calculations perhaps based on dynamically calculated neighborhoods.

The design of the framework uses design concepts that are familiar in the context of this domain. In fact, the CORMAS framework is very similar in design to the environments investigated as part of the domain analysis presented in Chapter 4. The main difference is that agents are used to represent entities rather than objects. The implication is that the agent responds to a request and makes a decision about whether or not to fulfill that request. When the entities are represented as objects, this kind of autonomy does not exist. The responsibility of the decision making does not lie with the object, rather, it lies with the object that calls a method on the entity. The entity does not have the capacity to reason about whether or not the call was appropriate.

7.2.1 Building an agent-based model with CORMAS

There are three main steps to building an agent-based model with CORMAS:

1. Define specific entities from the CORMAS generic entities
2. Define the control and scheduling of the model
3. Define observer's viewpoints

CORMAS provides a hierarchy of generic classes as shown in Fig. 7.1 below.

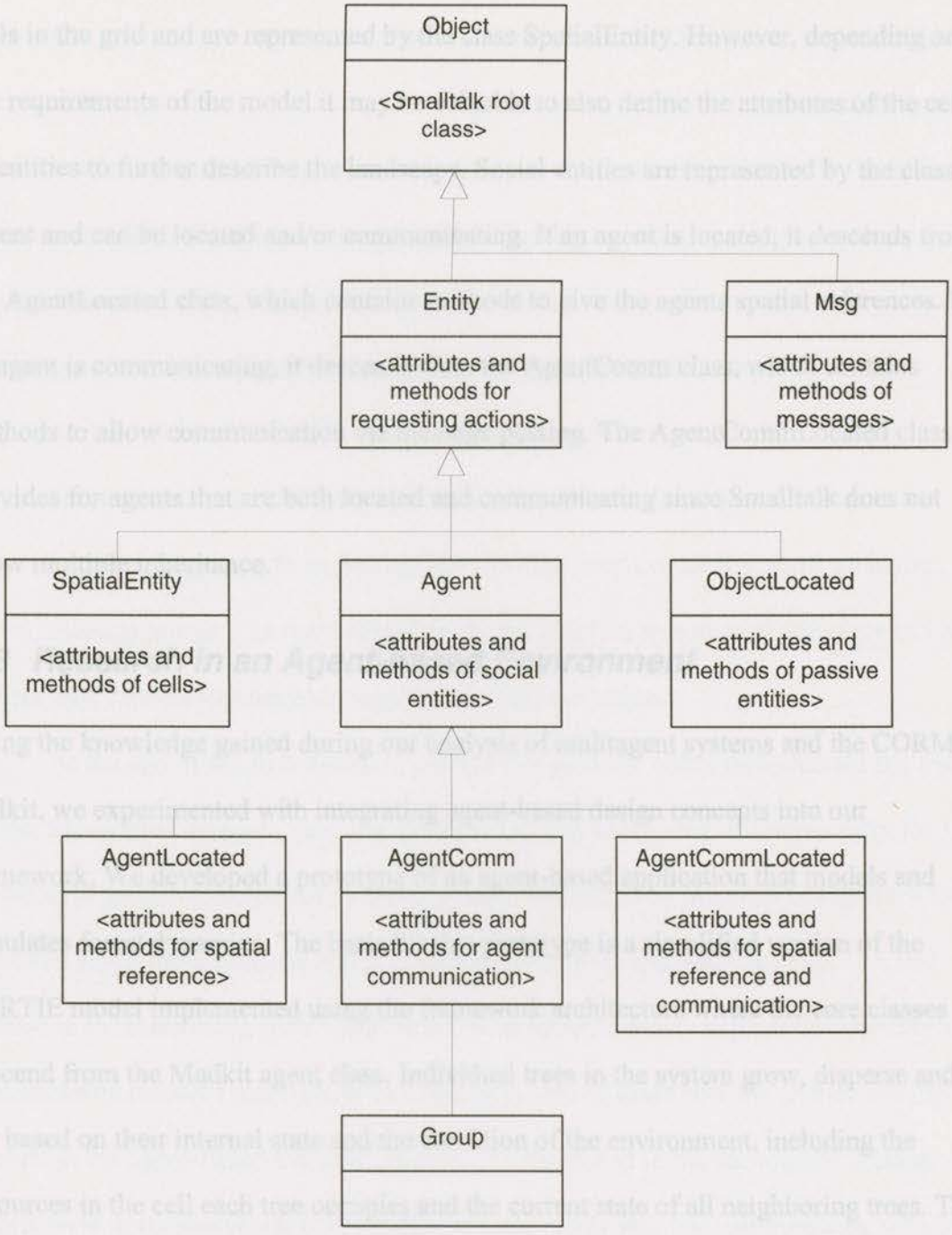


Figure 7.1 CORMAS Class Hierarchy

Each of the classes comes with a set of methods that implement the basic functionality of the class. Entities can be spatial, social or passive. Spatial entities are the cells in the grid and are represented by the class SpatialEntity. However, depending on the requirements of the model it may be valuable to also define the attributes of the cell as entities to further describe the landscape. Social entities are represented by the class Agent and can be located and/or communicating. If an agent is located, it descends from the AgentLocated class, which contains methods to give the agents spatial references. If an agent is communicating, it descends from the AgentComm class, which contains methods to allow communication via message passing. The AgentCommLocated class provides for agents that are both located and communicating since Smalltalk does not allow multiple inheritance.

7.3 Research in an Agent-based Environment

Using the knowledge gained during our analysis of mulitagent systems and the CORMAS toolkit, we experimented with integrating agent-based design concepts into our framework. We developed a prototype of an agent-based application that models and simulates forest dynamics. The basis for this prototype is a simplified version of the SORTIE model implemented using the framework architecture where the core classes descend from the Madkit agent class. Individual trees in the system grow, disperse and die based on their internal state and the condition of the environment, including the resources in the cell each tree occupies and the current state of all neighboring trees. The system also allows various simple land management strategies to be applied.

The development of this prototype underwent two iterations. In the first iteration, we attempted to implement all ecosystem entities as agents including individual trees and

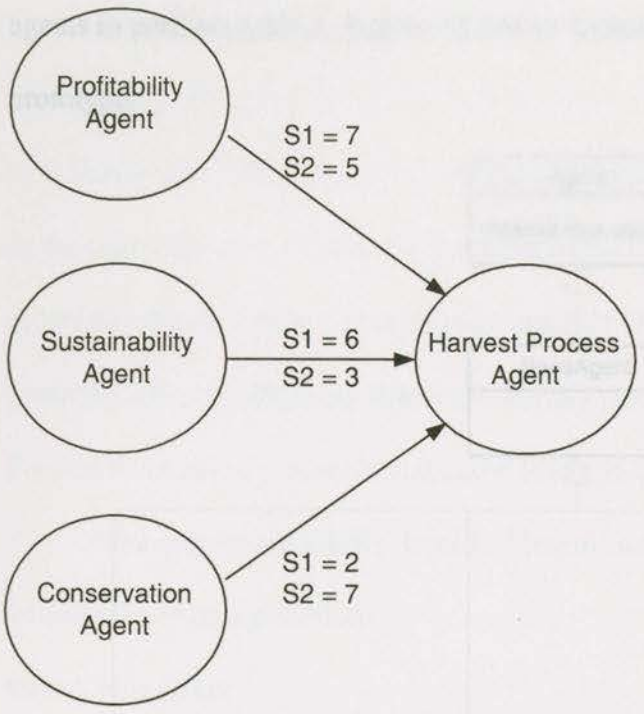
cells. The Tree agents were designed to interact with each other and their environment and had the appropriate internal representations to reason about lifecycle process requests from a process manager type agent. Their reasoning included how much to grow, how many seeds to produce and when to die. Each tree in the simulation sent requests for resource information and neighbor references from the agent representing the spatial environment. The trees then queried their neighboring tree agents, requesting their current state. However, this approach involved a high degree of messaging and since the Madkit platform does not guarantee message delivery there was to be considerable programming overhead to handle data loss. Agents retrieve messages by polling a mailbox type system provided by the Madkit kernal. Through program tracing, it was shown that a message sent from one agent to another does not always show up in the kernal message queue. The main symptom of this problem was missed messages in Tree-to-Tree and Tree-to-ResourceManager agent communication.

At the end of the first iteration, we had two choices: either reimplement the tree agents with robust message handling to create lossless messaging or reimplement the tree agents as objects. We opted for the latter since it seemed to be the most straightforward approach given the timeframe and produced cleaner more manageable code. Our real interest in experimenting with agent-based environments was to explore its applicability to sustainable resource management as well as to demonstrate how the framework can be applied in the agent-based solutions domain. As illustrated in the subsequent sections, our efforts were quite successful and actually fit quite nicely into our reengineering scheme.

7.3.1 Sustainable Resource Management Aspects

A key focus of this prototyping exercise was to test out ideas about the application of agents to the sustainable resource management aspects of ecosystem simulation using the core design concepts in the framework. We implemented a Harvest process agent that targets a Forest grid agent. The Harvest process agent destroys trees based on its *beliefs* about the strategies. It will also be able to *reason* about the effectiveness of the strategies. The Harvest process agent selects harvesting strategies based on current conditions and then requests that these strategies be rated by strategy agents. Upon request, these agents rate the strategies presented to them based on their beliefs about the effectiveness of the strategy in accordance with their goals.

The Harvest process agent uses consensus to determine which strategy to employ. The Harvest process agent uses a team of strategy agents to select an appropriate strategy: profitability, sustainability and conservation. Each of the subordinate agents has its own beliefs about what strategy it thinks should be employed. The Harvest process agent determines what strategies are possible given the current state of the system. It then asks the strategy agents to rate the selected strategies and determines which strategy should be performed based on the consensus calculation. This process is illustrated in Fig. 7.2 below.



- Harvest process agent selects S1 and S2 from available strategies based on current conditions.
- Harvest process agent requests rating of S1 and S2 from each strategy agent
- Strategy agents rate S1 and S2 and send to Harvest process agent
- Harvest process agent orders the ratings returned by the strategy agents
- Harvest process agent applies weights to determine degree of consensus
- Harvest process agent evaluates consensus and selects appropriate strategy

Figure 7.2 Consensus in the Harvest Process Agent

7.3.2 Prototype Architecture

The architecture of the agent-based prototype is similar to the architecture presented in Chapter 5. The main difference between them is in the communication mechanism among some of the components.

The SimulationManager has the same responsibility in the architecture. It controls the simulation environment and triggers components that contain the necessary logic to run a model. A model encapsulates a set of processes, and the ModelManager, similar to the frameworks SimulationScheduler, contains the communication protocols to trigger the ProcessAgents. Ideally, all levels of granularity would be supported using agents. That way, if appropriate, cells and individuals could be directly instructed by process

agents to perform actions. Figure 7.3 below shows the core simulation classes used in the prototype.

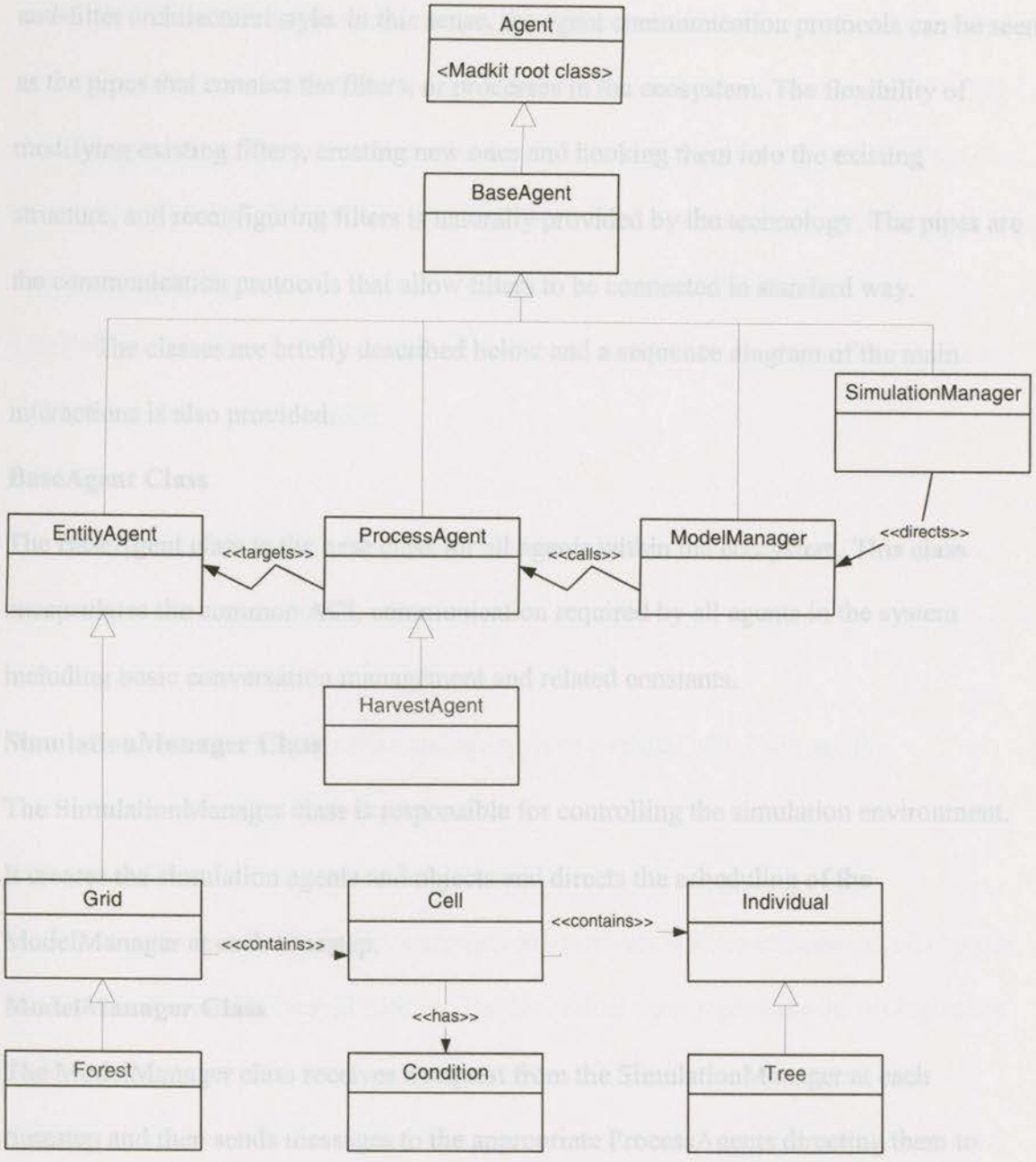


Figure 7.3 Integration of Agents with Framework

The agent-based approach fits nicely with our ideas about framework extensibility. We identified early on in the study that the SORTIE program has a pipe-and-filter architectural style. In this sense, the agent communication protocols can be seen as the pipes that connect the filters, or processes in the ecosystem. The flexibility of modifying existing filters, creating new ones and hooking them into the existing structure, and reconfiguring filters is naturally provided by the technology. The pipes are the communication protocols that allow filters to be connected in standard way.

The classes are briefly described below and a sequence diagram of the main interactions is also provided.

BaseAgent Class

The BaseAgent class is the base class for all agents within the ecosystem. This class encapsulates the common ACL communication required by all agents in the system including basic conversation management and related constants.

SimulationManager Class

The SimulationManager class is responsible for controlling the simulation environment. It creates the simulation agents and objects and directs the scheduling of the ModelManager at each timestep.

ModelManager Class

The ModelManager class receives a request from the SimulationManager at each timestep and then sends messages to the appropriate ProcessAgents directing them to effect change on a particular EntityAgent.

ProcessAgent Class

The ProcessAgent class provides the means for addressing a process within the ecosystem. It is the base class for all agents that perform tasks within the environment. These agents receive instructions from the ModelManager and can act on any EntityAgent within the system. Four grid level processes were implemented in the prototype including individual tree growth, dispersal and death and resource renewal.

EntityAgent Class

The EntityAgent class is the base class for all ecosystem entity agents that contain state information about the ecosystem.

Grid Class

The Grid class holds all the properties and methods that are common to all agents that represent a grided spatial entity.

Cell Class

The Cell class holds the properties and methods of a spatial cell. Cells are the fundamental units of the Grid class.

Individual Class

The Individual class holds all the properties and methods that are common to all objects that represent spatially located entities. The Individual class represents the biological or ecological entities within the simulation environment. The basic processes of an individual coincide with the natural lifecycle common to all living things: birth, consumption of resources to produce growth, reproduction and death.

The ModelManager runs and manages a set of processes. In this prototype, only the harvest process was implemented as a ProcessAgent. The basic lifecycle processes were built into the ModelManager, but they could have been externalized similar to the HarvestAgent. The HarvestAgent first requests a scan of the forest from the ForestAgent to get a sense of the conditions in order to select one or more appropriate strategies. The ForestAgent gets the appropriate information from the cells and trees contained in internal data structures. Once the HarvestAgent has received the necessary information, it uses strategy agents to select an appropriate strategy as described in the previous section and executes that strategy. Additional ProcessAgents and models could easily be fit into the architecture as long as they fit into the group/role structure and followed the necessary communication protocols. These interactions are depicted in Figure 7.4 below.

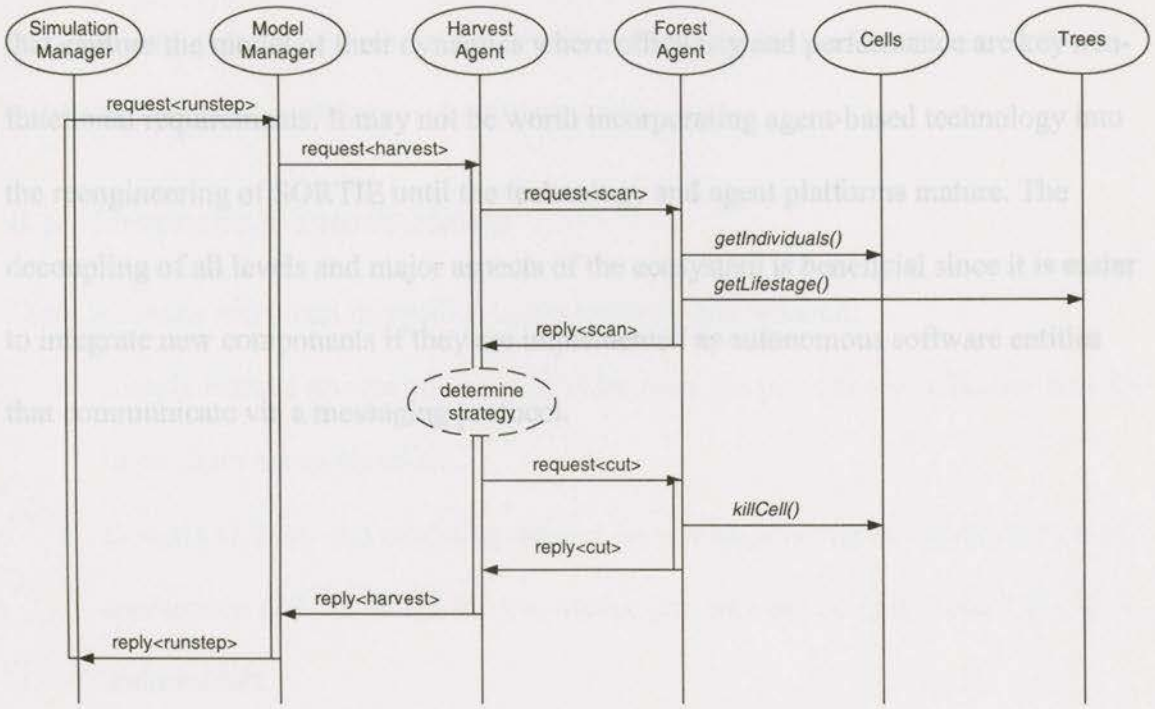


Figure 7.4 Sequence Diagram of Harvest Process

7.4 Summary

Our initial design idea was to implement the Cell, Individual and Condition classes as agents so that all of the core system components would interact the same way. In the original class hierarchy, these classes were descendants from the EntityAgent class. This would also make it so that ProcessAgents could target areas of the environment more specifically.

Using agents for the basis of the architecture provides many advantages, however, unless the core agent platform can support a high quality of service it doesn't seem worth it to implement individuals as agents in this case. Our problem can be characterized as one with a relatively large number of individuals with a high degree of messaging. Forested areas contain a dense population of individuals with deterministic algorithms that capture the model of their dynamics where efficiency and performance are key non-functional requirements. It may not be worth incorporating agent-based technology into the reengineering of SORTIE until the technology and agent platforms mature. The decoupling of all levels and major aspects of the ecosystem is beneficial since it is easier to integrate new components if they are implemented as autonomous software entities that communicate via a messaging protocol.

Chapter 8

Conclusions

The research presented in this thesis represents a study in the application of domain based reengineering methodology. It is a valuable demonstration of a combination of software engineering methods used in a practical application. The thesis also provides important documentation of knowledge and ideas related to the reengineering of SORTIE.

In this chapter we describe the theoretical implications of our research by summarizing the major benefits from each of the techniques applied during the domain-based reengineering. We also discuss the practical implications of this research including the contributions to the SORTIE reengineering project and potential contributions to other projects in the same domain. Finally we describe the known limitations of our results and present possible areas for future work.

8.1 Theoretical Implications

There are many important theoretical implications of this research:

1. Simple manual reverse engineering techniques can provide cost-effective benefits in program comprehension.
2. Domain analysis and modeling support reverse engineering as a communication mechanism and can bridge the knowledge gap between designers and application stakeholders.
3. Domain engineering supports reengineering activities by uncovering alternative solutions to common design problems.

4. Frameworks can be used to capture domain assets defined during domain analysis and are more widely applicable when the design is abstracted from the constraints imposed by implementation choices such as programming language.
5. Agent-based technology is applicable to this domain and can be applied to the framework with respect to the sustainable resource management aspects.

These implications of our research are detailed in the following subsections.

8.1.1 Benefits of Simple Manual Reverse Engineering Techniques

During this study, we had access to the SORTIE source code and development environment thus were able to create an executable and run the program. This allowed us to apply simple program comprehension strategies and reverse engineering techniques to determine what the program does and how it does it. We visually inspected the structure of the code and produced a class hierarchy. Since there was minimal use of inheritance it turned out that our manual efforts were more cost-effective in terms of time than trying to parse the source and load it into a program comprehension tool for the same purpose. We then analyzed the program from the user perspective to get an idea of task support. For each task, we examined the dynamic behavior of the program by stepping through the code as it was running and noted how the classes were involved in supporting the task. Again this technique is simple and effective, yet difficult to reproduce using automated methods.

8.1.2 Domain Analysis to Support Reverse Engineering

We began our study with an analysis of the system as-is, primarily through code inspection and modeling of operational capabilities and gained a better understanding of the system implementation and architectural design. At this point, we understood that a major problem was the lack of object-oriented design principles in the code structure, however, even with mining of human expertise we were unable to get a clear sense of what objects and relationships would be necessary to meet the reengineering goals. A major issue was the differences in backgrounds between the application stakeholders and ourselves. The SORTIE programmers do not have a strong background in object-oriented design and we did not have any background in ecological modeling and simulation. We needed a mechanism to bridge the gap, a platform from which to discuss ideas in a tangible way and consequently turned to domain engineering methods.

Through our investigations of other applications for ecosystem modeling and simulation, we were able to identify and understand the characteristics specific to spatially explicit individual-based population modeling. We did this in collaboration with the application stakeholders through a process of presenting hypotheses about the domain and discussing them in relation to SORTIE. In this way, domain modeling was used effectively to support knowledge acquisition during the reverse engineering process.

8.1.3 Domain Engineering to Support Reengineering Activities

Once we had a better understanding of the domain characteristics, we began exploring domain legacy to uncover alternative solutions for similar design problems. We used collaborative work sessions to present alternative solutions and discuss their potential applicability to the reengineering of SORTIE. We experimented with applying alternative

design solutions to SORTIE through prototyping a subset of the model. This exercise gave us a better understanding of the design problems and the domain characteristics were refined as more was learned about the specific domain. We used our analysis of the domain as a basis for guiding the reengineering activities.

8.1.4 Frameworks to Capture and Discuss Domain Assets

We used an object-oriented framework as a recording medium for our understanding of the domain characteristics and design solutions. The framework served as a basis for discussion and evolved with our understanding of the domain. We abstracted the design from the constraints imposed by implementation choices such as programming language. This makes the framework more widely applicable since it captures an abstracted view of the domain. As we have described, the framework can be adapted to accommodate technologies required by the particular instantiation such as for a distributed environment.

8.1.5 Applicability of Agent-based Technology in this Domain

In the last step of our study, we applied agent-based technology to the framework in support of the sustainable resource management aspects of the domain. Sustainable resource management is about reasoning and decision making to select among alternatives. It can also include communication and collaboration among agents with different belief systems. An agent-based solution is a natural fit.

Modeling and simulation tools allow for experimentation and learning about the effectiveness of land-use strategies. Using agents, support for this learning can be built into the application. As described in the previous chapter, the agent representing the harvesting process could create a set of alternative plans that are then rated by

profitability, sustainability, and conservation agents. The user of the simulation tool could rate the outcome of the plan (based on the effect of the land use strategy on the ecosystem, economy etc) and this feedback could be communicated to the agents. The agents could then update their rating belief system to create better plans.

The integration of agents into the framework also demonstrated how the framework could be adapted to accommodate a specific technology. We implemented the core framework classes as agents to allow for easy integration of new models using established communication protocols.

8.2 Practical Implications

There are two important practical implications of our research. First, the study has been valuable to the SORTIE stakeholders since it was used to guide the reengineering process. The thesis is also of significant value since it captures knowledge related to the reengineering of the program. Second, we have abstracted the domain knowledge from the specific instantiation and documented our design through an object-oriented framework. The domain assets and framework contained in this thesis are potentially useful towards other software engineering projects in the same domain. We detail these implications in the subsections below.

8.2.1 The SORTIE Reengineering Project

We used the study to drive the SORTIE reengineering project and this thesis captures the reengineering assets. The thesis will be of continued use to the SORTIE stakeholders as the reengineering effort progresses and towards maintenance of the application.

We discovered through an iterative prototyping process that with this particular reengineering project, understandability of the design was more important to the success of the reengineering effort than any other non-functional requirement. The application of this framework to a particular instantiation can provide many challenges that require a solid background in object-oriented design principles. If the SORTIE programmers have difficulty mapping their understanding of the existing application onto the design of the framework then it will also be difficult to do the reengineering of SORTIE. We recommend that the SORTIE programmers continue collaboration with an object-oriented designer throughout the reengineering of the application.

A key constraint in this reengineering project is the continued use of the existing development environment. This is mainly to leverage the expertise of the SORTIE programmers and to promote code reuse to speed up the reengineering process. However, we have attempted to abstract the design such that it was independent of development environment so that other languages and technologies can also be considered.

8.2.2 Future Projects in the Same Domain

This thesis captures knowledge that applies to a class of applications. It presents an object-oriented framework that may be useful to other projects in the same domain by providing a structured presentation of common functional requirements, constraints produced by typical non-functional requirements, key design issues and possible solutions.

The design is abstracted from the constraints imposed by choice of specific implementation technologies and thus is more widely applicable to designers and programmers of tools for spatially explicit individual-based population modeling and

simulation. As we have mentioned, these tools can be effective in ecological studies for exploring the dynamics of ecosystems and are also used in sustainable resource management. The domain is large enough that the application specific constraints in future projects will affect design choices in the instantiation of the framework. In any case, this thesis is useful even as a basis for exploring similar design issues and possible solutions.

8.3 Limitations

The reengineering of SORTIE is still being implemented and the suggested design has not been fully verified to meet all the requirements. As well, the original reengineering requirements have changed since the onset of the study, in that the main application programmer is leaving the project before the reengineering is complete. The framework instantiation described in this thesis does not capture all of the design issues and constraints related to the SORTIE model. Many detailed design issues arise during the implementation process and are known only to the programmers. However, the framework can provide the basis for ongoing documentation of these design problems and solutions.

The support provided by the framework for the functional requirements are relatively easy to verify. If the SORTIE model can be implemented using the framework then the stated requirements are being met. Verification of the framework has been an ongoing process in that the requirements evolved through the design of the framework and vice versa. However, the success criteria for non-functional reengineering requirements were not set up at the onset of the project. These success criteria should be

clearly established by the application stakeholders for validating of the framework and can be included as part of the testing of the reengineering application.

8.4 Future Work

There are three main areas where the research presented in this thesis can lead to future work. First, once the application has been reengineered it should also be validated to ensure that the SORTIE model exhibits the same qualities using the new architecture. Second, a comparative study of the legacy application and the new application in terms of model support, performance and maintainability would be of practical value to the SORTIE stakeholders with respect to the original reengineering goals. Third, the use of agents in software engineering is relatively new and is highly applicable in this domain. Continued research in agent-based solutions would also provide interesting future work and has good potential for collaborative research since resource management requires expertise in from multiple disciplines including forestry, computer science and environmental psychology.

Bibliography

[1] Bass L., Clements P., Kazman R. ,"Software Architecture in Practice." ISBN 0-201-19930-0

[2] Lettes, J., Wilson, J. *Army STARS Demonstration Project Experience Report* (STARS-VC-A011/003/02). Manassas, VA: Loral Defense Systems-East, 1996

[3] Fayad, M., Schmidt, D. Guest editorial for the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[4] Foreman, John. *Product Line Based Software Development- Significant Results, Future Challenges*. Software Technology Conference, Salt Lake City, UT, April 23, 1996.

[5] P.J. Sydelko et al. (1999). A Dynamic Object-Oriented Architecture Approach to Ecosystem Modeling and Simulation. Proceedings of 1999 American Society of Photogrammetry and Remote Sensing (ASPRS) Annual Conference, pp. 410-421.

[6] P.B.Woodbury et al (2000). The ECLPSS Environment for Developing Spatial Ecological Models Using Reusable Components. http://cycas.cornell.edu/ECLPSS/

[7] Cubert, R. M., T. Goktekin, P. A. Fishwick. MOOSE: architecture of an object-oriented multimodeling simulation system. Proceedings of Enabling Technology for Simulation Science, Part of SPIE AeroSense '97 Conference, Orlando, Florida, April 22-24.

[8] Doran J. E. "Agent-Based Modelling of Ecosystems for Sustainable Resource Management". In "Multi-Agent Systems and Applications". (*ACAI'01 Summer School, Prague, July 2001,* Proceedings), eds. M Luck, V Marik, O Stepankova, R Trappl. Springer: LNAI 2086. June 2001. Pp. 383-403

[9] Dagermo, P; Knutsson, J. Development of an Object-Oriented Framework for Vessel Control Systems. Dover Consortium. Technical Report ESPRIT III/ESSI/DOVER Project #10496, 1996

[10] ATLSS: Across-Trophic-Level System Simulation: An Approach to Analysis of South Florida Ecosystems. Biological Resources Division, United States Geological Survey Technical Report

[11] Ralph Johnson and Brian Foote. ``Designing Reusable Classes." Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35

[12] P. A. Campbell et al. The Dynamic Information Architecture System: An Advanced Simulation Framework for Military and Civilian Applications. http://www.dis.anl.gov/DIAS/papers/SCS/SCS.html September 2001

[13] EcoSim http://www.offis.uni-oldenburg.de/projekte/ecotools/project_ecotools4.htm March 2002

[14] Bousquet, F., Bakam, I., Proton, H. and Le Page, C., 1998. Cormas: common-pool resources and multi-agent Systems. Lecture Notes in Artificial Intelligence 1416: 826-838

[15] Ecoforestry Institute http://ecoforestry.ca/ February 2002

[16] Gilbert GN, Terna P. How to Build and Use Agent-based Models in Social Science. *Mind and Society*, 2000, 1(1), pp57-72

[17] Clements P, Kazman R., Klein M. *Evaluating Software Architectures: Methods and Case Studies*, published by Addison-Wesley in Oct 2001.

[18] ATLSS, Biological Resources Division, U.S. Geological Survey. http://www.atlss.org/, March 2002.

[19] CORBA Basics, Object Management Group. http://www.omg.org/gettingstarted/corbafaq.htm, June 2002

[20] MadKit Development Guide, Version 3.1. http://www.madkit.org/, March 2002.

[21] SORTIE, C. Canham, Institute of Ecosystems Studies. http://www.ecostudies.org/people/cvs/canham.html, December 2000.

[22] Borland C++ Builder, Version 5. http://www.borland.com/cbuilder/cbuilder_5/index.html, December 2000.

[23] Rigi http://www.rigi.csc.uvic.ca/ September 2000

[24] Imagix 4D http://www.imagix.com/products/imagix4d.html September 2000

[25] T.A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294-306, 1989.

[26] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13-17, January 1990.

[27] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61-72, May 1988.

[28] XML
http://www.w3.org/XML/ July 2002

[29] S.K. Card, Mackinglay J. D., and B. Shneiderman. *Readings in Information Visualization – Using Vision to Think.* Morgan Kaufmann, San Francisco, 1999.

[30] David Garlan and Mary Shaw, *Advances in Software Engineering and Knowledge Engineering,* Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, Singapore, pp. 1-39, 1993.

[31] Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Developement Process.* Addison-Wesley, 1999

[32] Prieto-Diaz, R. "Domain Analysis: An Introduction." *Software Engineering Notes 15,* 2 (April 1990): 47-54

[33] Douglas C. Schmidt, ``Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," Handbook of Programming Languages}, Volume I, edited by Peter Salus, MacMillan Computer Publishing, 1997

[34] David S. Hamu and Mohamed E. Fayad, "Achieve Bottom-Line Improvements with Enterprise Frameworks," The Communications of ACM, October 1997

[35] Fayad, M. E., Schmidt, D. C., and Johnson, R. E. *Building Application Frameworks.* Addison-Wesley Pub Co, 1st edition, 1999

Appendix A

SORTIE Instantiation Supplemental

This appendix provides some implementation ideas and suggestions for the SORTIE instantiation classes.

The Grid class

```
Class Grid : public SimulationObject
{
protected:
      Cell[][] myCells;
      int gridHeight;
      int gridWidth;
      int numEWCells;
      int numNSCells;
      int lenEWCells;
      int lenNSCells;
      boolean iterateNS;

public:
      Grid();
      Grid(int numEWCells, int numNSCells, int lenEWCells, int
lenNSCells);
      iterateNS();
      iterateEW();
      setGridPosition(int x, int y);
      addNextCell(Cell*);
      Cell *getNextCell();

}
```

The constructor initializes the grid by setting the number of cells and their length. However, this could easily be changed, or additional constructor added to support some alternative initialization mechanism. The two methods iterateNS() and iterateEW() set the iteration mechanism to either traverse the grid cells row-wise (EW) or column-wise (NS). The traversal scheme could default to EW unless otherwise specified. The current

position for traversing the grid can be set using the setGridPosition() method. This method will determine which cell corresponds to the specified coordinate. The addNextCell() method inserts a pointer to the specified cell into the myCells array according to the current position in the array as defined by the iterate and position methods. The getNextCell() method returns a pointer to the next cell in the myCells array as defined by the iterate and position methods.

The Cell Class

```
Class Cell : public SimulationObject
{

protected:
      int NWcorner_x;
      int NWcorner_y;
      int lengthNS;
      int lengthEW;
      Individual[] myIndividuals;
      Condition[] myConditions;
      int numIndividuals;
      int numConditions;

public:
      Cell(int nwc_x, int nwc_y, int lenNS, int lenEW);
      addIndividual(Individual *i);
      removeIndividual(Individual *i);
      addCondition(Condition *c);
      removeCondition(Condition *c);
      int numIndividuals();
      int numConditions();
      List<Individual> *getIndividuals();
      List<Conditions> *getConditions();
      Individual *getIndividual(int x, int y, str individualId);
      Condition *getCondition(int x, int y, str conditionId);
}
```

The addIndividual() and addCondition() methods add the specified pointer to the myIndividuals and myConditions arrays respectively and increment the variable corresponding to the total number of each. The removeIndividual() and

removeCondition() methods remove the specified pointer to the myIndividuals and myConditions arrays respectively and decrement the variable corresponding to the total number of each. The numIndividuals() and numConditions() methods return the number of individuals and number of conditions respectively. The getIndividuals() and getConditions() methods return arrays containing the individual and condition pointers respectively. These methods would be used by SimulationProcess objects to retrieve target individuals and conditions. The getCondition() and getIndividual() methods return a single condition and individual respectively using the specified absolute coordinate and a unique id for the type of individual or condition.

The Individual Class

```
Class Individual : public SimulationObject
{
protected:
     Cell* inCell;
     int x;
     int y;

public:
     Individual(int x, int y, Cell *myCell);
     Cell* getCell();
     Individual *getNeighbors();

}
```

The Individual is initialized to a pair of absolute coordinates via the constructor. The getCell() method returns a pointer to the cell where the Individual is located. The getNeighbors() method is provided since determining the neighbors in the cells immediately neighboring the home cell will likely be a common operation. This method returns an array of pointers to individuals in the cells that border the home cell. If there

are no neighboring individuals, this method returns null. Sample code for the getNeighbors() method is provided below.

```
<List>Individual *getNeighbors()
{
  Cell[] neighboringCells = inCell->getBoundaryCells();
  int numCells = neighboringCells->length();

  Individual[] neighbors;

  for (j=0; j<numCells; j++) {
    if (neighboringCells[j]->numIndividuals > 0) {
       appendArray(neighbors, neighboringCells[j]-
     >getIndividuals);
     }
   }
   return neighbors;
}
```

The SimulationProcess Class

```
class SimulationProcess : public Object
{

protected:
     int sequence;

public:
     SimulationProcess(int sequence);
     int execute(SimulationObject *simObj);
}
```

When a SimulationProcess is created, it should set a sequence value to help in determining when it should be requeued if need be. There will likely be many of descendants from the base SimulationProcess class in an instantiation of the framework. They all must have an execute method so that they will fit into the rest of the framework. SimulationProcess objects can act on any SimulationObject, so the process may be

targeting a specific Individual, a Cell, a Condition or the entire Grid. The code contained within the execute method will be very specific to the scale of the process.

Consider the following processes:

- Individual mortality process
- Individual growth process based on shading in neighborhood

The code for the individual mortality process might be as simple as:

```
int execute(Individual *i)
{
   int reQueue = 0;

   if (i->growthRate < 0){
      i->die();
   } else {
      reQueue = sequence;
   }
}
```

Whereas in the the neighborhood influenced individual growth process, the calculation requires knowledge about the neighbors of the target individual. In this case there has to be code in place to determine who those neighbors are. The code would likely need to support finding the home cell of the individual as well as the cells surrounding the home cell. Obviously this code is slightly more complex and might look something like:

```
int execute(Individual *i)
{
   int reQueue = 0;
   double shade = calculateShade(i);
   double growthRate = baseGrowthRate/shade;
```

```
    i->height = i->height*growthRate;
}
```

Where calculateShade is a private method. Example code for this is shown below.

```
double calculateShade(Individual *i)
{
   Individual[] neighbors = I->determineNeighbors();

   // some code in here that calculates the shade on
   // iterating through the individuals
}
```

Where determineNeighbors() is a public method of the Individual class. Potential code for this process is shown below under the Individual class description.

The SimulationScheduler Class

```
class SimulationScheduler : public Object
{

      class SimulationEvent : public Object
      {

      public:
            SimulationObject *eventObject;
            SimulationProcess *eventProcess;
            int eventTime;

            SimulationEvent(SimulationObject*, SimulationProcess*,
      int runtime)

      }

protected:
      SimulationEvent[] events;
      int simulationTime;
      int runLength;

public:
      SimulationScheduler(int runLength);
          boolean addEvent(SimulationObject*, SimulationProcess*, int runTime);
                                runEvents(int currentTime);

}
```

When the SimulationScheduler is created, it should set the total runLength of the simulation schedule. That way, when events are added, the SimulationScheduler can check to see if the requested runTime fits within the runLength – that the event is not out of the simulation time frame.

SimulationEvent is an internal class of SimulationScheduler. It is only needed and used within SimulationScheduler. When the addEvent() method is called, an event is created and added to the queue of events. It is up to the programmer whether or not to order the queue according to runTime upon event insertion. Although, if not properly implemented, this may add more overhead than it is worth. If the event is successfully added into the queue, the addEvent method should return a boolean status of True. When the runEvents() method is called, it is passed the currentTime. The SimulationScheduler should run through the queue of events looking for events where the eventTime is less than or equal to the current time. When it finds matching events, it should call the execute() method of the SimulationProcess, passing it the pointer to the SimulationObject as a parameter and deleting the event from the queue. It is probably a good idea to create some private methods to handle the queuing and dequeuing. As shown in the SimulationProcess class description, if the execute() method returns an integer greater than zero, then the event should be requeued at the current time plus the returned integer.

The code for the runEvents() method might look something like:

```
void runEvents(int curTime)
{
  SimulationEvent curEvent;
  int reQueue;

  resetEventQueue();

  while( isMoreEvents() )
  {
```

```
    curEvent = getNextEvent();
    if (curEvent->eventTime == curTime)
    {
       reQueue = curEvent->eventProcess->execute(eventObject);

       if (reQueue > 0) {
         reQueueEvent(curTime+reQueue);
       } else {
          deQueueEvent();
       }

    }
  }
}
```

VITA

Surname: Macdonald Given Names: Sachen Mary Margaret

Place of Birth: Campbell River, British Columbia, Canada

Educational Institutions Attended:

University of Victoria 1993 to 2002

Degrees Awarded:

B.Sc. University of Victoria 1999

Honors and Awards:

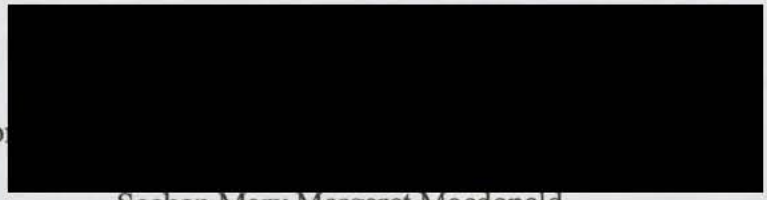
Publications:

UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria of Victoria Library, and to make copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain by the University of Victoria shall not be allowed without my written permission.

Title of Thesis:

Domain-based Reengineering of a Spatially Explicit Individual-based Population Modeling and Simulation Research Tool

Autho 

Sachen Mary Margaret Macdonald

September 3, 2002