2006

A Reflection Based Framework for Automation of Parameter Variations in Simulation

George H. Gilman Jr.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A REFLECTION BASED FRAMEWORK FOR AUTOMATION OF PARAMETER VARIATIONS IN SIMULATION

By

GEORGE H. GILMAN, JR.

A Thesis submitted to the
DEPARTMENT OF COMPUTER SCIENCE
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2006

The members of the Committee approve the Thesis of George H. Gilman, Jr. defended on November 21, 2005

R C Lacher
Professor Directing Thesis

Geoffrey Brooks
Outside Committee Member

Dan Schwartz
Committee Member

Sara Stoecklin
Committee Member

David B Whalley
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The purpose of computer simulation is to utilize a mathematical model to recreate a real world situation such that the behavior and interactions of the entities involved can more easily be understood. By varying initial conditions and external stimuli in a controlled simulated environment, simulation often provides much better insight to an entity's behavior than would be readily observable in the real world.

Traditional simulation design relies upon dynamically varying a set of input parameters and comparing the simulation outputs to determine parameter sensitivities. Each parameter which will be dynamically altered requires a separate piece of code, generally a looping structure, to control the changing parameter values. Though many input parameters exist within the scope of a simulation, traditional design facilitates the measurement of only a small subset of these parameters as it is not feasible to write the code to alter all input parameters. Varying the number of dynamic parameters is costly due to required software changes and it is not always known during the development phase which parameters are of interest.

One approach for altering nearly all parameters in a simulation, rather than a small subset, is through the use of reflective architectures. Rather than statically defining and altering a small subset of input parameters, reflective architectures provide facilities whereby any simulation object which exists in memory can be examined and altered at run time. The premise for the work in this paper is that a framework built upon reflective architecture can be built which provides a simple, flexible mechanism for manipulation of computer simulation input parameters. This thesis outlines a solution based upon reflective architectures and describes a framework which was created to facilitate this purpose.

CHAPTER 1

INTRODUCTION

The purpose of computer simulation is to determine how a system, either naturally occurring or man-made, behaves as a reaction to external stimuli. Initial conditions, internal processing, environmental parameters, and the magnitude of each stimulus are used as inputs to a series of equations and processes which approximate how a system under test will behave. A variety of uses exist for simulation including training, scheduling, resource management, system development, and studying natural phenomena.

Utilizing a simulation in order to understand how a system or phenomenon behaves under a variety of conditions, an analyst will choose a series of input parameters over which the system under test will be measured. Such a series of input parameters is called a *run matrix* and is generated based upon probable inputs which will yield a desired series of outputs.

Traditionally simulation design has been centered on being able to execute a finite series of pre-defined run matrices. The simulation will allow a small, finite subset of input parameters to be varied across a series of executions or runs. This grouping of runs is known as a *run set*. The issue with the traditional methodology of simulation design is that a simulation often has far more inputs than is feasible to automate execution of run matrices. Rather than having simulations designed to dynamically vary all input parameters, a smaller finite set of chosen parameters is allowed to be altered. This leaves the other parameters to be altered manually via text files or database entries.

A solution to the problem of finite run matrices for computer simulation is introduced in this paper. Rather than developing a simulation around varying a finite number of input parameters, it will be shown through using reflection a framework can be built which allows any memory resident simulation parameter to be part of a run matrix with minimal additional coding in the simulation.

The rest of the paper is laid out as follows: Chapter 2 covers additional preliminary background, covering simulation methodologies and reflective architectures. Chapter 3 describes the different components which are required to develop a framework to solve the issues of dynamic parameter alterations. Finally, Chapter 4 discusses an implementation of the described framework, the utilization of which is described in Chapter 5. Chapter 6 describes the extensibility of the framework while Chapter 7 describes some of the shortcomings and future work. Chapter 8 draws the final conclusions of the work performed.

CHAPTER 2

BACKGROUND

2.1 PARAMETERIZATION

According to Thesen [1], “Simulation is simply the use of a computer model to “mimic” the behavior of a complicated system and thereby gain insight into the performance of a system under a variety of circumstances.” With appropriate modeling simulation can be used to achieve a number of benefits over the traditional cycle of build-and-fix. Though some metrics are difficult to quantify, benefits include:

- Lower lifecycle cost
- Ability to control parameters that would be out of the user’s control in the real world [2]
- A simulation can generally be executed a large number of times for less than the cost of a single field trial
- Risk reduction

For a number of years the author of this paper has been directly or indirectly involved in the investigation and development of a number of computer simulations utilized by the U.S. Navy. Regardless of simulation design and purpose, each simulation in effect executes in a similar way. As seen in Figure 1, as a simulation begins execution input parameters are read in from databases and text files. Following, the simulation executes and results are generated. One of the initial input parameters is then modified and the simulation is re-executed. This cycle repeats until all combinations of the *n* inputs under test are tested.

The processes by which each simulation defines and alters input parameters, called *parameterization,* were varied. However, none of the simulations utilize a method that is flexible in defining the input parameters and the subset of these parameters which is altered between scenarios. Most simulations have a small, finite set of parameters which can be automatically parameterized while others provide for no automated facilities; parameterization is accomplished by manually creating different setup files. For complex simulations where often a single run set may take a week or more to execute, this is unacceptable.

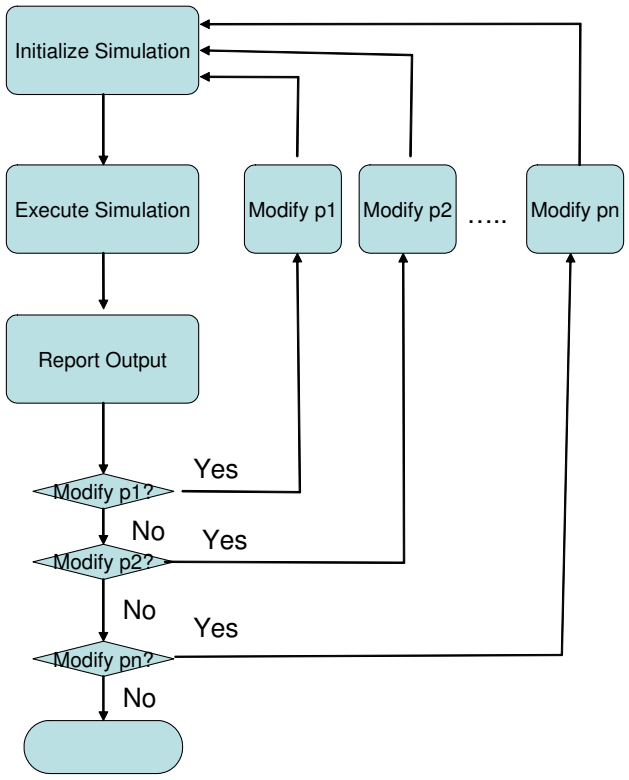

Figure 1: Simulation Execution

2.2 PARAMETER SPACE

Referring to Figure 1, the parameter search space for a run matrix can become extremely large. Assuming the innermost parameter, p_1 is iterated m_1 times, the innermost loop in the nest structure executes on the order of $O(m_1)$. In modifying parameters the number of execution cycles for each parameter is independent of other iterations. Thus, the next closest outer loop, which modifies p_2, in turn is executed m_2 times. Hence, like the inner loop the secondary loop executes on the order of $O(m_2)$, with an overall execution between the two loops of $O(m_1) * O(m_2) = O(m_1 * m_2)$. Fully expanded, assuming n as the number of parameters under study, the run matrix complexity is on the order of $O(\prod_{i=1}^{n} m_i)$. Assuming the worst case, if all loops within the nest execute the same number of iterations, this simplifies to $O(m^n)$. From the complexity determination we can readily observe that the number of parameters in the run matrix is more important

in determining the overall run matrix size than is the number of iterations per parameter. However, increasing either provides a rapidly growing matrix.

The first step in determining the size of the run matrix is to use knowledge and intuition to determine input parameters of interest and bounding regions for these parameters. An analyst knowledgeable of the system under test will determine based upon system specifications and desired responses a preliminary list of relevant parameters and values.

The next step in determining the run matrix is to perform sample executions within the prescribed run matrix. Rather than initially executing all combinations within the matrix, a few selected examples are executed. Based upon these results parameters are often excluded and the size of the solution space for each parameter altered. Though generally the solution space for individual parameters is reduced by this method it is possible to actually increase the space if unexpected results occur.

For practical purposes, the number of parameters observed in the solution space is kept small. The size, however, is determined by the design and implementation of the simulation as well as the observations being made. From past experience, fewer than 6 parameters are normally observed and the number of iterations per parameter is generally no larger than 10. Though these numbers appear modest, such a scenario yields a maximum run matrix size of 10^6.

As an example of run matrix size, consider an underwater acoustic scenario including the following parameters:

- 6 sets of environmental conditions
- 5 water depths per environment
- 3 wind speeds per environment
- 3 different sensors
- 3 acoustic signatures
- 10 ranges between vehicle and sensor

For this example, the overall size of the run matrix is 8100 entries. Run matrices with a few thousand entries, such as this, are most common. However, larger and smaller matrices occur as well.

2.3 EXAMPLE SIMULATIONS

Following is a list of simulations which the author has been involved with along with a description of how parameter optimization is performed in each case.

2.3.1 COUNTERMEASURE EVALUATOR (CME)

The CME was originally developed in 1968 in support of the MK 37/MK 46 systems [3][4]. The simulation is a real-time hardware-in-the-loop (HWIL)

simulation and stimulation facility and included modeling for sonar systems, torpedoes, mines, mine hunting systems, countermeasures, ships, and submarines.

The parameterization facilities for the CME consisted of generating two separate input files. The first file defines the overall scenario such as which ships and torpedoes were present and initial parameters for the first run. The second file contained a series of parameter alterations which override input values in the original data file. The number of entries in the second file is somewhat minimized since only alterations from the initial plan were entered and the simulation contains a small number of parameters which are able to be altered. Adding a new parameter combination for dynamic alteration requires additional coding in the simulation. Furthermore, no process is in place to automate the generation of input files; rather they are manually edited.

2.3.2 NAVAL MINE WARFARE SIMULATION (NMWS)

NMWS was originally developed at Naval Surface Warfare Center – White Oak (NSWC-White Oak) for applications with anti-air warfare (AAW), theater missile defense (TMD), and strike warfare [3][4]. In later years it was adapted include mine warfare and amphibious warfare capabilities.

The simulation methods employed in NMWS are probabilistic in nature and entire battle fleets can be simulated in a single simulation as an unlimited number of entities can be evaluated. The main measurements of the simulation include time to complete mission, system effectiveness, and logistics.

The scenario initialization for NMWS is almost entirely defined within database structures. In order to allow parameter values to change between runs as part of a run set, NMWS has a scripting tool which can be used to generate new scenarios. However, each time a different parameter set is defined, a different script must be coded to modify the parameters between run executions. This process is time consuming and cumbersome.

2.3.3 AUTONOMOUS LITTORAL WARFARE SYSTEMS EVALUATOR-MONTE CARLO (ALWSE-MC)

ALWSE-MC was developed under funding from the Office of Naval Research (ONR) to provide an analysis tool for underwater unmanned vehicles (UUVs) [3]. The general focus of the tool is in the applicability of mine hunting. The simulation relies predominantly on probability tables and as such is a Monte Carlo style simulation.

Unlike the CME, ALWSE-MC does provide some automated parameterization facilities. Within the simulation most parameters are altered by the user interaction with the Graphical User Interface (GUI). The parameters which are altered between run executions in an automated manner each exist in

an editable fashion as a text box component in the user interface. When the application is launched, it dynamically queries each text box and provides facilities by which the user can easily automate the parameterization of any of these parameters. This method is flexible in that if a new text box component is added to the GUI, the user can automatically automate the processing of the contained parameter without the developer adding any additional code.

Though the method outlined here is somewhat flexible, it is not without flaws. First, though most parameters which would normally be optimized reside in the GUI, not all do. Some reside in databases, text files, or optional screens; all areas beyond the control of this method. Secondly, the data is tied to a specific control. The method works with text boxes only. If the data resides in another format of GUI control, this solution fails. Finally, even though parameter inspection is dynamically handled, the application itself contains all code related to the inspection and parameterization. This makes the reuse of the method difficult.

2.3.4 OTHER SIMULATIONS

Other simulations which the author of this paper has direct familiarity with either provide no automated methods for parameter optimization, or methods less advanced than the previously discussed simulations. Simulations such as the Personal Computer Shallow Water Acoustic Tool-set (PC-SWAT) [5] [6], Autonomous Littoral Warfare Systems Evaluator – Engineering Simulation (ALWSE-ES) [3] [7], as well as others, have little or no support for flexible parameterization. An exhaustive survey of simulation articles has failed to provide any additional methods. Many articles exist with reference in determining the number of parameters required and which parameters are important [8] [9]. However, none appear to describe any automated processes for parameter generation.

2.4 REFLECTIVE ARCHITECTURE

The concept of reflection was originally proposed by Smith [10]. Reflection allows the programmer access to data representing aspects of the system itself [11]. The key to reflection is for the programming architecture to offer the software developer a meta-data interface supporting the inspection and adaptation of the underlying virtual machine. It has been provided by a series of languages such as JAVA, Smalltalk, ObjVlisp, CLOS, Python, PHP, and the .NET languages. Reflection characteristics are broken into two broad categories; behavioral and structural reflection.

2.4.1 STRUCTURAL REFLECTION

The fundamental principles of reflection rely on allowing the programmer to have access to the internal representation of a system and allowing the programmer to then effect changes on this internal representation. The term for exposing the internal mechanisms is *reification* and for allowing the programmer access to effect changes is *absorption*.

Structural reflection is the ability of a language to provide a complete reification of the program currently executing. This includes the data within the program as well as underlying abstract data types. Hence, the software developer is allowed access to the structure of the objects, classes, and data types of an object-oriented design at run time. Some of the capabilities provided by reification include:

- Retrieving a class name
- Discovering class modifiers
- Discovering super classes
- Identifying interfaces implemented by a class
- Identifying class constructors
- Obtaining information on method calls

Through structural reflection's absorption method calls for objects can be determined and thus invoked at run-time rather than compile time. Through structural reflection the user is even allowed to programmatically alter data structures such as the definition of a class, function, or a record on demand.

The support for structural reflection varies between languages. For example, Java provides facilities for introspection of classes and invocation of methods but little in the way of allowing these structures to be altered [12]. However, structural reflection is considered easier to support than behavioral reflection [13].

2.4.2 BEHAVIORAL REFLECTION

While structural reflection relies upon programmatically determining and utilizing a program's structure at run-time, behavioral reflection is the ability to alter the behavior of operations in a program at run-time. Behavioral reflection was originally proposed by Smith [10] in 1982 and other researchers [14][15] have investigated its usefulness. Behavioral reflection allows the description and modification of underlying code within the program. For the purposes of this paper, behavioral reflection is not relevant and thus the topic is left to be discussed by other sources.

CHAPTER 3

REFLECTIVE SOLUTION THEORY

One method of providing a flexible, robust framework for automated parameterization is through reflection. Through reflective means, the user can choose an object at run-time for which parameters should be altered for optimization purposes. The program can display a list of properties to the user, allow each property chosen for optimization, and systematically execute all combinations of parameters.

3.1 PROPERTY INTROSPECTION

The first step in allowing access to parameters is through reification, or introspection, of a class. It is assumed that all simulation objects exist within the memory space of the application. Given this and a method by which an individual object can be chosen in the GUI, the application can inspect the object and determine its properties at run-time. The parameters for the object are then displayed within the GUI, including parameter names and values. However, some distinction must be made between processing functions and property accessors.

To determine properties of an object, a common naming sequence or other method by which properties can be extracted must be utilized. After all, not all methods of a class are created to access properties. The simplest and most common way to designate accessors is through the terms ‘get’ and ‘set’. It is common practice in most languages today to have all accessor names begin with the terms ‘get’ and ‘set’. So, while introspection of a class will return all functions, accessors or otherwise, a common naming scheme such as prefixing accessor methods with ‘get’ and ‘set’ allows differentiation between accessors and general purpose functions. As well, accessors generally require no parameters for retrieving values and a single parameter for modifying values in accessors. This can be used as a secondary method for determining properties versus more generalized methods.

Another important issue of property introspection with relation to accessors is data type. Given basic data types such as integer and floating point, later setting a value through an accessor will be simple. However, if the underlying data type of a property is a complex type such as another object or structure, such access becomes more difficult. Decisions must be made for the framework that determines how to handle complex data types. The options begin with handling only native numerical data types, but may be able to be expanded to enumerations, strings, and objects.

Once the properties of an object have been determined, a GUI component should be created to display the properties. Without such a component, the end

user will have no method by which to choose parameters to be altered in the simulation. A common display method can be seen in Integrated Development Environments (IDEs), such as in Figure 2 which was taken from Microsoft Visual Studio .NET. However, Figure 2 is only one example of such a design and many other implementations are possible.

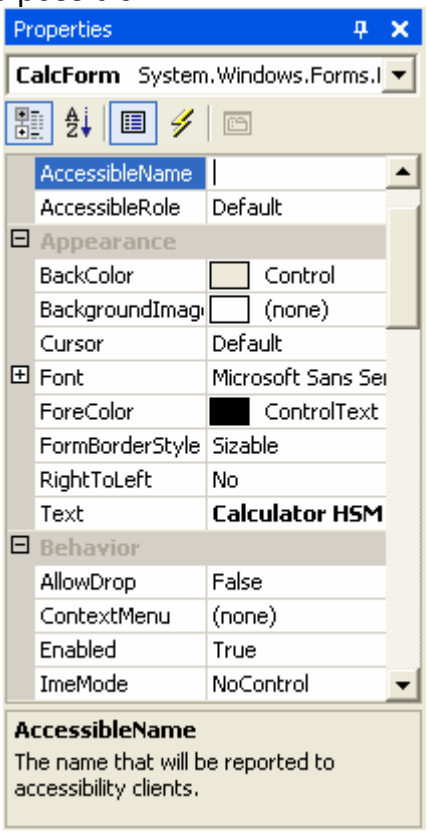

Figure 2: Property Display Example

It is noteable that the simplest solution for determining which properties are accessible to the end user of the simulation is to expose all properties which contain public methods to both retrieve and modify the values of the property. However, dependent upon application, more restricted access may be required. Greater restrictions allow the developers and end users to have a different view of the data. The issue of data access is left for future development.

3.2 PARAMETER ABSORPTION

When the user chooses a parameter to alter dynamically, he must also determine the range of values for the alterations. The control values are the initial value, a step size, and number of iterations. Along with a reference to the original object, a new Iterator class can be created which knows how to alter the

underlying property of the original object. UML for such an Iterator class can be seen in Figure 3.

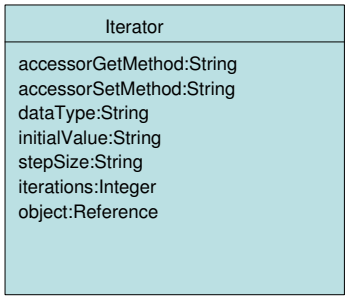

Figure 3: Iterator Class

As an example, assume we have a vehicle class for which we are going to be altering the speed between individual runs of the run set. The speed needs to be altered for simulation purposes from twenty miles per hour to thirty and then forty. The speed of a vehicle is accessed via “getSpeed” and “setSpeed” methods of the vehicle. In Figure 3, the accessorGetMethod and accessorSetMethod will be “getSpeed” and “setSpeed”, respectfully, as strings. These will be used to retrieve and set the values of the property. The datatype would most likely be a floating point number, possibly “float”. However, data type, initial value, and step size are all left as strings. They will be parsed at run time as to eliminate data type dependencies in setting the values. Thus, stepSize would be “10.0”, initialValue “20.0”, and iterations would be set to 3. The object reference would be set to the original object which is having the parameters altered.

The Iterator class, as described above, generates a linear search path in the parameter space. Figure 4 displays the linear search path within the search space defined by a single parameter, p_1. In this figure, the slope of the search path is determined by stepSize and the number of iterations determines the size of the length of the path within the search space. This search path is determined by the run matrix, as discussed earlier. Looking at two parameters, as in Figure 5, the search path is defined as a series of lines which form a plane within the search space.

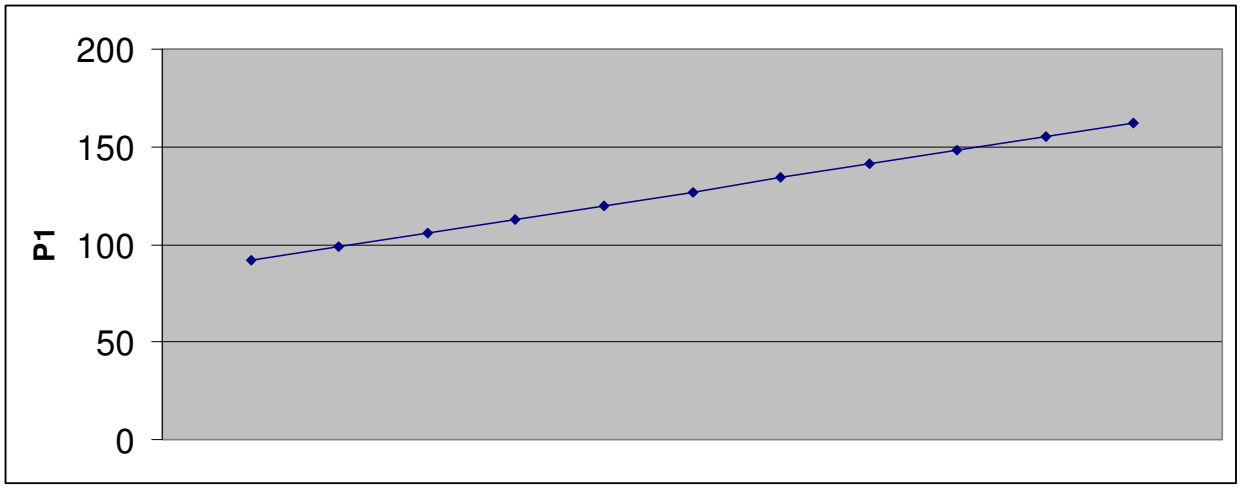

Figure 4: Linear Search Path for Single Parameter

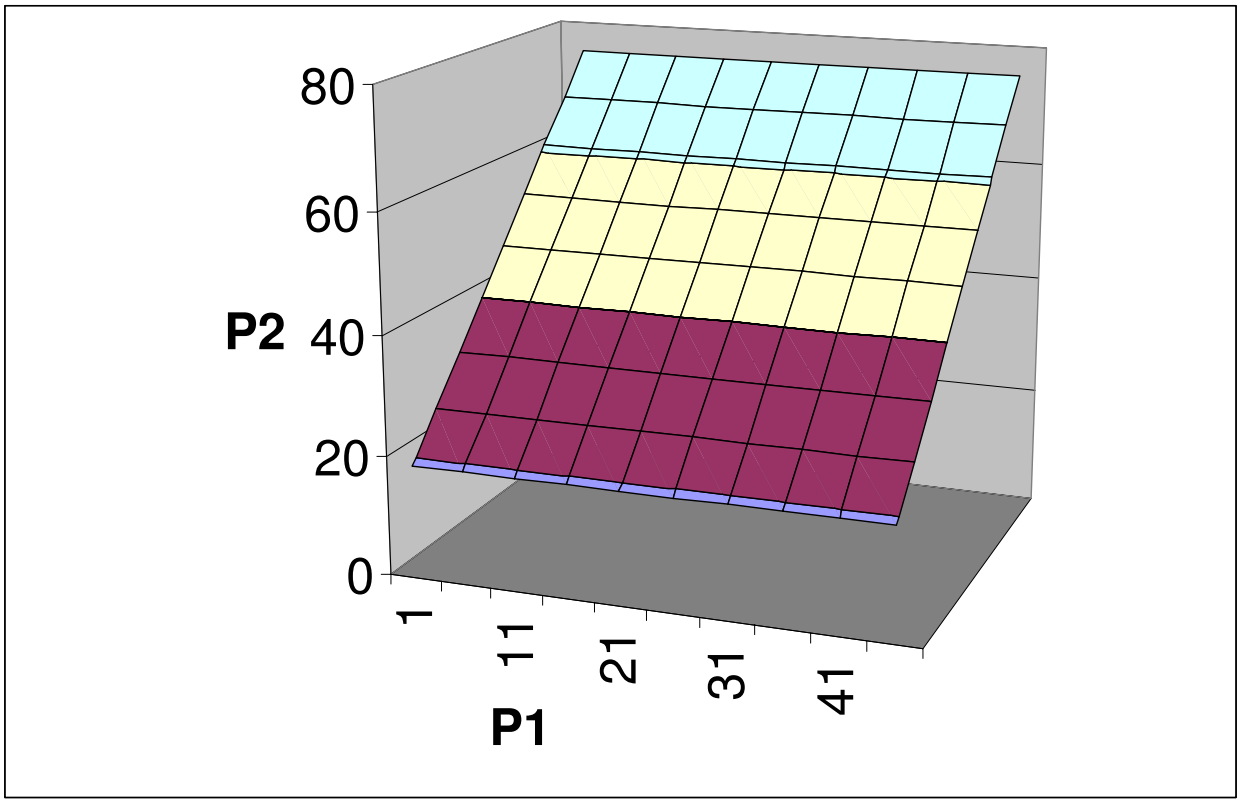

Figure 5: Linear Search Space for Two Parameters

Though a single iteration type, a linear iteration, is discussed here other iterations can be realized as well. For example, a discrete list of values for the property to take and exponential growth are other viable search paths. An

example of an exponential search path for a single parameter can be seen in Figure 6.

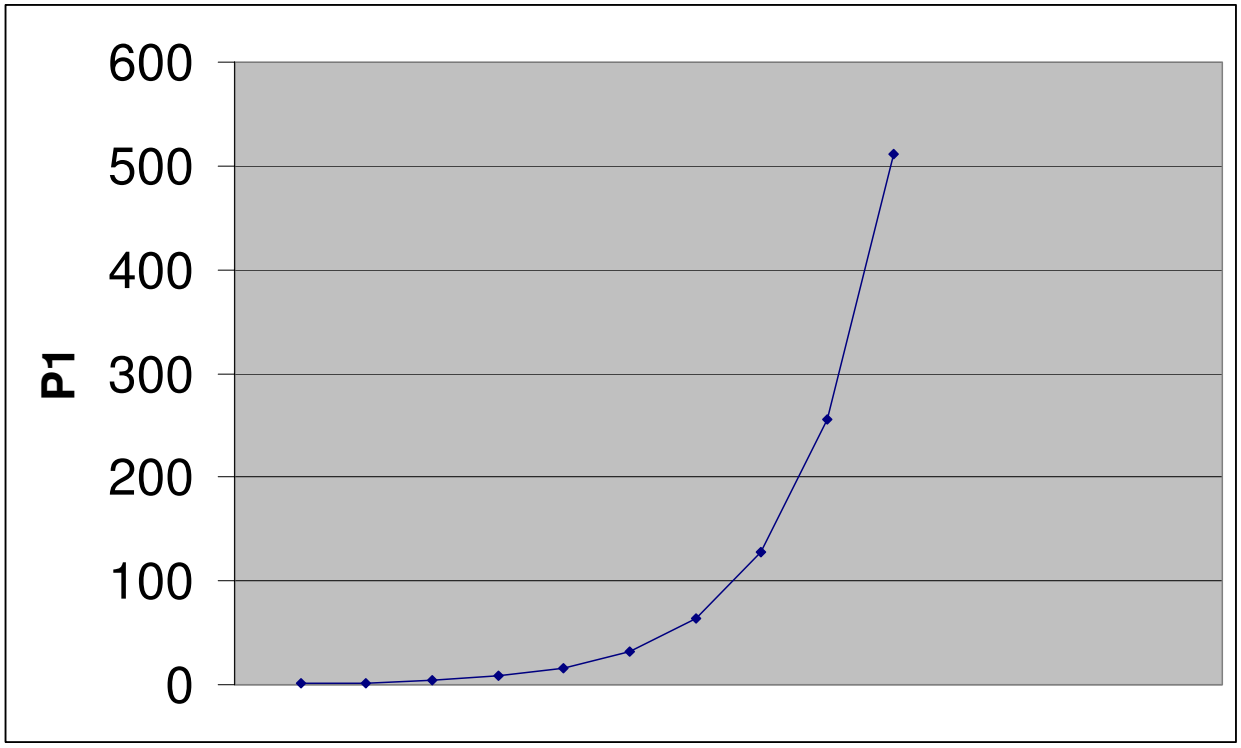

Figure 6: Exponential Search Path

3.3 NESTING PARAMETERIZATIONS

The user has the ability to iterate over more than one parameter at a time. As such, after a single parameter iteration has been determined, other parameter iterations may be added to the list. At run-time between run executions, each parameter can be iterated for all possible parameter combinations.

3.4 SETTING PARAMETER VALUES

Through the introspection process outlined above, a class is created as in Figure 3. Through the stored “accessorSetMethod” string and reference to the underlying object, reflection can be used to alter the value of the underlying parameters. One benefit of using structural reflection in this fashion is that we are accessing the underlying object data via the accessors, as the original developer intended. Thus, the internal representation of the property can remain hidden from view of the end user.

3.5 SIMULATION ENGINE DESIGN

Assuming a framework designed as above, the simulation engine will require little interaction with the framework. However, some interaction is necessary. First, the simulation GUI which is knowledgeable about the objects of the simulation must be able to pass objects to the framework via either the GUI control mentioned above or another mechanism. All object introspection, containment, and iteration, however, will be transparent to the GUI; hidden within the framework. Next, the simulation engine and developed framework must be synchronized such that the simulation engine is notified when the underlying simulation data has been modified by the framework for the iteration. Possible mechanisms for this control are a single method call or possibly an Observer design pattern [16].

3.6 RESEARCH CONTRIBUTION

Though reflective architecture has been discussed in a number of venues [10][14][15], extensive research has not been able to produce any evidence of work related to the utilization of reflective architecture to provide a flexible mechanism for dynamically altering input parameters of any system. Hence, it is the author's belief that the concept as outlined in this thesis is unique to this document.

CHAPTER 4

IMPLEMENTATION

A solution for the prescribed framework was developed and implemented under Visual Basic .NET 2005 Beta 2. The implementation language is relatively unimportant and the solution is easily adaptable to other languages which support reflection.

4.1 INTROSPECTION

The first step in providing a dynamic parameterization solution is to determine the properties of the inspected object at run time. Though the solution was intentionally developed as a generic framework, the choice of implementation language brought some interesting nuances to introspection. Primarily, unlike most languages Visual Basic .NET has the concept of a “property”. While most languages such as Java and C++ have notional concepts of accessors, in reality the mechanisms which exist to create accessor methods cannot be distinguish between accessors and processing functions. A “property” in Visual Basic .NET is a specialized set of functions that are used only for providing accessor methods. These can be easily distinguished from processing functions. This provides a convenient way to determine attributes which can be parameterized. Alternately in other implementations, inspection of functions that are prefixed with “get” and “set” is a solution, as discussed earlier.

Another interesting aspect of language choice is the existence in the Visual Basic IDE of the PropertyGrid control. This is a user interface control which, given the reference to the object, performs introspection upon the object and provides the end user with a detailed list of all object properties and allows for modification. Internally this reification is based upon reflection.

After much work and experimentation, it was determined that writing the code for such a control would be of little use since the provided one suited all requirements. The only item utilized by the framework which is taken directly from this user control is the method property name, given as a string value. Since the properties of the object are determined at runtime, even if additional properties are added later none of the end user application code needs to be modified.

4.2 PROPERTY ABSORPTION

To facilitate the data absorption for properties to be parameterized, a series of classes were developed. These classes hide the data type dependent implementation details of how to retrieve and set property values. The classes provide not only retrieval and alteration capabilities but simple iteration

mechanisms as well, such as adding and subtracting. The class diagram for this implementation can be seen in Figure 7.

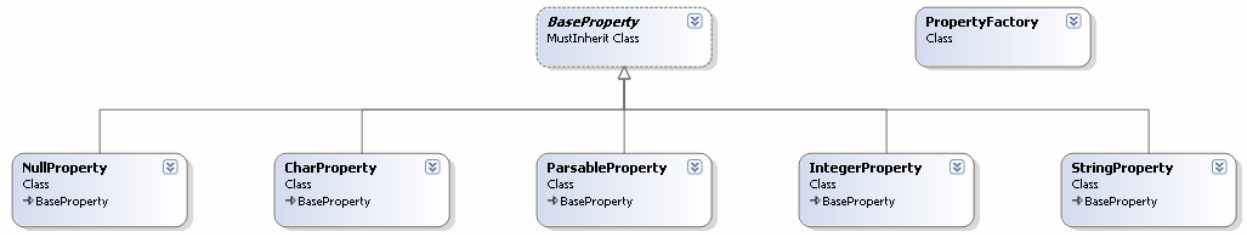

Figure 7: Absorption Layer Classes

The end user does not directly access any of the automation data. Rather, he communicates via the parameterization layer which in turn accesses the absorption layer. The parameterization layer is further discussed later within this document.

In the absorption layer property accessors are manipulated via textual strings. Since the access to BaseProperty interfaces are consistent regardless of underlying data type, a string representing a floating point number “1.23” can be handled as easily as a textual string value “ABC”. Each property data type, when passed a string representing the value to be set to the property, determines how to internally store the value. This aspect is obviously data type dependent; hence the need for the BaseProperty inheritance structure.

In addition to the ability of storing and retrieving property values, each BaseProperty subtype must be able to add and subtract numeric values. This again is a data type dependent process. For numeric types the process is obvious 1+1=2, 1.2 + 2.3 = 3.5, etc… However, for the character and string data types the choices were less obvious. For a character data type, the resultant value is the integer offset from the ASCII character table. Hence, ‘a’ + 1 results in the value of ‘b’ being set. For string based properties, a post-fixed numeric is added to the string based upon the input number. For example, “Hello” + 1 results in “Hello1” and “Hello123” + 7 results in “Hello130”. Though adding numeric values to strings may appear to have little value, it well serves a purpose in simulation for naming output files. If the scenario is named “ScenarioA” and is executed three times, a convenient method of auto naming the files is “ScenarioA1”, “ScenarioA2”, and “ScenarioA3”.

One property data type not easily recognizable from the class diagram is the ParsableProperty data type. This data type is capable of handling all numeric data types under the implementation language. Each numeric type inherently provides a *parse()* method. Via the parse method all required operations are made. Thus, there is no requirement to specifically develop a subtype to handle 16 bit integers differently than 64 bit floating point numbers.

The one class shown in the diagram which is not part of the BaseProperty inheritance is the PropertyFactory class. This class simply provides factory

pattern creation methods for generating *BaseProperty* subtypes. However, this is accomplished through introspection. It is determined via reflection at run time the data type of the property being sent to the factory. From this determination the proper subtype is created. As implemented, the code to make this determination in the factory follows the format:

```
Dim info As System.Reflection.PropertyInfo =
      aObject.GetType.GetProperty(aName)
If info.PropertyType Is GetType(System.Char) Then
       Return New CharProperty(aObject, aName)
```

Another utilization of reflection within the factory can be seen in the final property type, ParsableProperty. This property is somewhat different in that it handles a large number of underlying data types. As such, the previous method of determining data type is not sufficient. Rather, using reflection it is determined if the object supports the "*parse (string)*" method. As extracted from the implementation code, this is determined by the GetMethod function:

```
Return Not aObject.GetMethod("Parse", New Type()
      {GetType(String)}) Is Nothing
```

A brief discussion of the automation process implementation was discussed here. A full code listing for the absorption task can be found in Appendix A.

4.3 PARAMETERIZATION

Parameterization was created as a separate subsystem from the absorption tasks. While the automation tasks handle the data type dependent aspects of setting and retrieving data values, the job of the parameterization subsystem is to provide a flexible looping structure for system variables. The class diagram representing this subsystem can be seen in Figure 8.

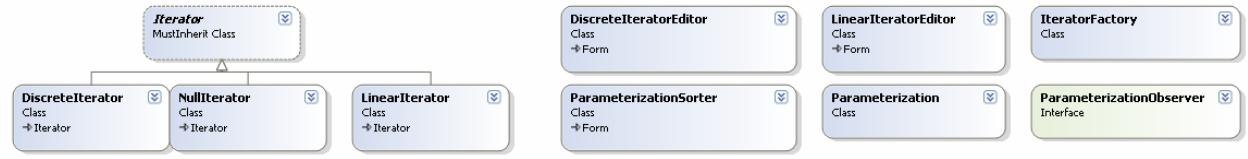

Figure 8: Parameterization Layer Classes

The primary concern of this subsystem is to provide different looping mechanisms. Two are currently provided in this implementation: discrete and linear iterations. Discrete iterations occur when a finite length list of items is to be processed but without a linear structure. For example, the set {1,5,7,30,29} is a discrete set. Linear is a set where the increment between consecutive values is fixed. This type occurs more often in simulation than discrete and is described as a starting value, the number of iterations, and the amount to increment between iterations, such as the set {10, 20, 30, 40, 50} which starts at 10 and increments

by 10 for a total of 5 values. The various iterators are represented as part of the Iterator hierarchy in the above diagram.

To facilitate the nesting of iterations, the Parameterization class provides mechanisms for maintenance of an iteration list, complete with sorting capabilities. Form classes in the diagram above provide GUI support for editing the different iterator properties. The entire interface to the framework is provided through this Parameterization class. The interface method names and descriptions are provided in Table1.

Table 1: End User Framework Interface

Method Name	Function
`AddTail`	Creates a new parameterization. The return value is the iterator created for the parameterization
`increment`	Perform one incrementation of the parameterization nest
`incrementAll`	Execute all incrementations of the parameterization nest
`reset`	Reset data values to those prior to any increment
`edit`	Open a dialog box used for editing and sorting parameterizations
`initialIteration`	Begin initial incrementation of nests. Should be called prior to an initial call to increment()
`registerObserver`	Register an observer for the parameterization
`unregisterObserver`	Unregister an observer for the parameterization

For the end user's code to be notified when a full loop cycle has completed, the user code will implement the *ParameterizationObserver* interface and call the *registerObserver* method of the Parameterization instance to be observed.

A full code listing for the Paramterization subsystem can be seen in Appendix B.

CHAPTER 5

APPLICATION

A sample application was created as part of this effort to demonstrate and better describe the utilization and capabilities of the developed framework. A full source listing for the application can be seen in Appendix C. Below some highlights from the application are discussed.

5.1 APPLICATION DESCRIPTION

The application written very loosely represents a scenario previously executed in the ALWSE-ES simulation. In the ALWSE-ES simulation a study was performed to determine the susceptibility of a single craft to a single mine. A variety of parameters were explored such as a various mine logic/sensing capabilities, environmental conditions, range to target, and vehicle-mine depth differential. The application written to demonstrate this framework is not nearly to the level of detail of the ALWSE-ES simulation. Nor is the processing involved in the demonstration code meant to represent real world physics. However, the similarities to the runs executed on the ALWSE-ES simulation demonstrate the applicability of this framework in real world applications.

5.2 APPLICATION DESIGN

The class diagram describing the simulation design can be seen in Figure 9. In the diagram two different mine types can be seen, called AlphaMine and BetaMine. Each implements the *Mine* interface, supporting a single method *update*(), which determines, based upon range to target, if each mine logic will acoustically be satisfied. If so, the mine will ‘fire’.

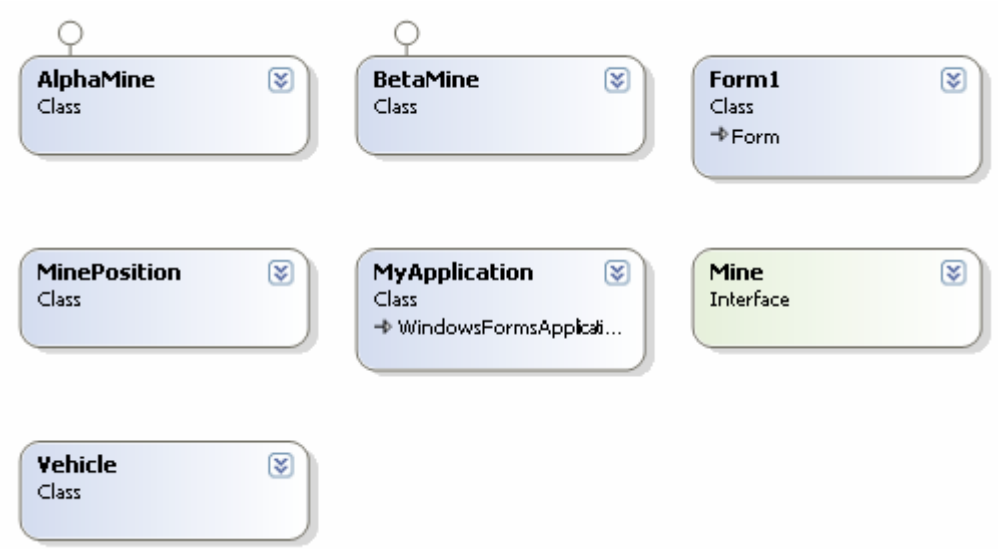

Figure 9: Application Design

The *Vehicle* class represents the vehicle under test and contains positional data along with an acoustic signature level. The *MinePosition* contains the positional information of the mine along with a reference to the underlying mine logic.

5.3 FRAMEWORK INVOCATION

A simple set of instructions must be sent from the parent application to the framework. These list of instructions are described in Table 2. This table represents a subset of the methods described in Table 1 as not all are required to be utilized for this scenario.

Table 2: Framework Utilization for Demonstration Application

Method Name	**Function**
`AddTail`	Creates a new parameterization. The return value is the iterator created for the parameterization
`incrementAll`	Execute all incrementations of the parameterization nest
`edit`	Open a dialog box used for editing and sorting parameterizations
`registerObserver`	Register an observer for the parameterization
`unregisterObserver`	Unregister an observer for the parameterization

To facilitate the framework invocation, a PropertyGrid was used in the application. This provides a mechanism for determining the properties of the application objects. In the GUI, shown in Figure 10 when the user selects an

object from the TreeView, it is displayed in the property viewer. After this, the user sets the properties directly and creates parameterizations as well. To create a parameterization, the user simply right-clicks the property in the PropertyGrid and determines the parameterization type. At this point, a call is made to the Parameterization subsystem of the framework, causing an additional iteration layer to be added to the iteration list upon editing the iteration properties. At any point, the user may click the “Edit Param” button which allows for sorting of the iteration nest.

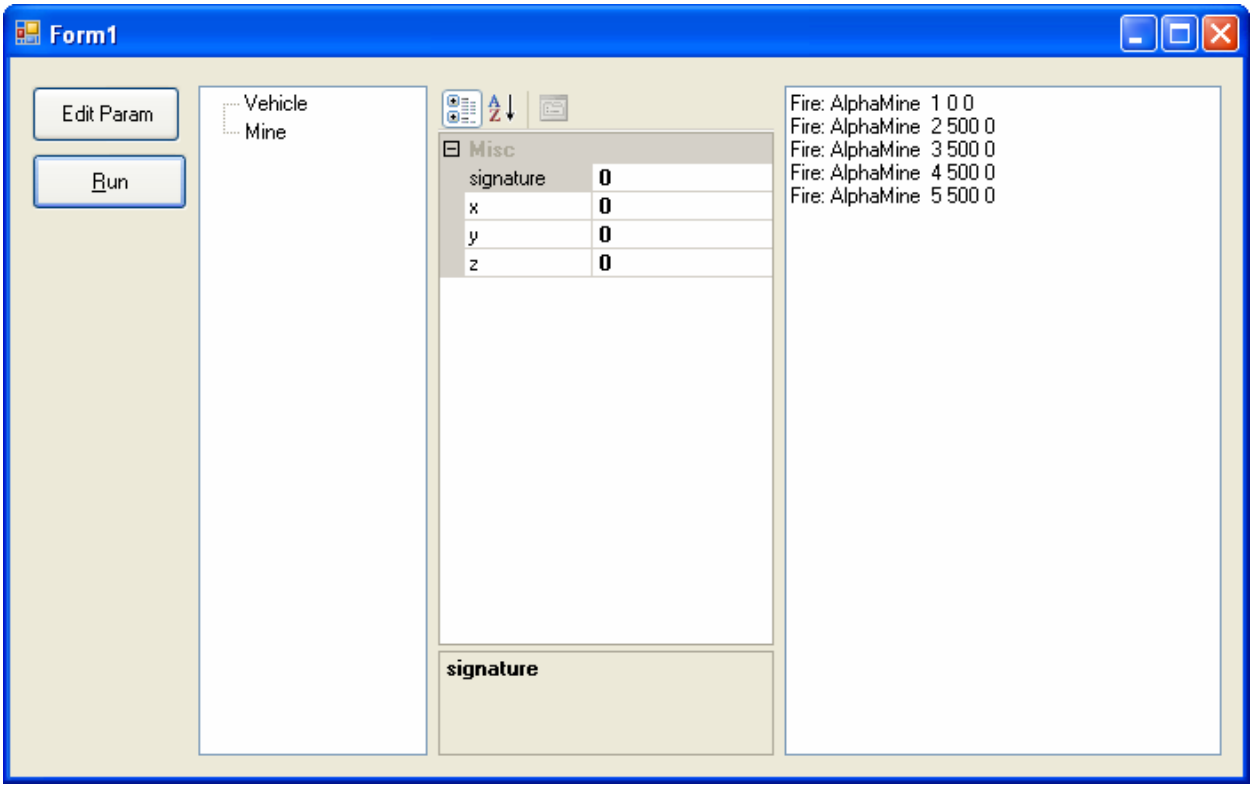

Figure 10: Demonstration Application

Upon completion of setup for all properties and parameterizations, the user clicks the “Run” button. This makes a call to the *incrementAll* method of the Parameterization layer.

5.4 APPLICATION CONCLUSIONS

Utilizing the application the end user has the choice of dynamically altering any of the relevant parameters of the simulation. In representing the original ALWSE-ES runs, the vehicle-mine offset can be altered by parameterizing over the vehicle’s y value and acoustic signature of the vehicle can easily be altered as well. Though those examples are numeric, string values are also optionally

altered. In the example application the creation of a discrete list of strings consisting of “AlphaMine” and “BetaMine” for the name of the mine logic results in the different mines being executed for the same scenarios.

CHAPTER 6

EXTENSIBILITY

Though the framework was developed to handle a wide array of common uses, many were beyond the scope of this research. This section describes the most likely required extensions to the framework.

6.1 DATA TYPES

A general requirement of the framework is to handle each data type separately. It is not possible to provide a single class which has the ability to store and retrieve data from all data types, as well as perform basic mathematical operations. As implemented, the framework handles all numeric data types, regardless of integer vs. floating point or number of storage bits utilized. Likewise, it also handles strings and characters. If it is determined that other data types should be handled as well, such as user defined classes, a new subtype of *BaseProperty* must be implemented and the *PropertyFactory* updated. Changes as such are relatively unlikely as most parameterizations in simulation are numeric. However, a case can be made where a class which represents a numeric type such as a complex number should be parameterized.

6.2 ITERATIONS

Though the different data types are somewhat unlikely to be added to the framework, the number of possible iteration methods is much larger. Logarithmic, geometric, or other series can be readily identified. To extend the framework to handle other iterations, a new *Iterator* subtype must be created which represents the logic of determining initial iteration step and how to determine the intermediate and concluding values. Likewise, an editor must be provided for editing the *Iterator*'s properties and the *IteratorFactory* updated.

CHAPTER 7
FUTURE WORK

The framework provides a flexible, convenient method for dynamically altering simulation parameters. However, as with any software it can be improved. First, though linear alteration is most common, there are a large number of other iteration methods which could be used to extend the framework. Data types are somewhat better covered. However, user defined classes and simple enumerations are not.

The largest area of work though is in determining a method of archiving the parameterizations. By design, each Iterator contains a reference to the object which it is manipulating. This provides a quick, easy access method via reflection. However, rebuilding references after reading a parameterization from a data file is not a simple task. One possible method of facilitating archival is by requiring each object which can be parameterized to implement a common interface which contains a numeric unique identifier. This unique ID can be utilized at file load time to rebuild the parameterization structure. However, this requires a bit more work on the end user side to maintain the IDs and thus takes away a degree of flexibility.

Another interesting extension to the framework which was brought to my attention as a result of this work is the applicability of this framework to determining the relevance of simulation parameters. Throughout this text the assumption has been made that the relevance of input parameters is known by the end user of a simulation. As such, the end user determines the parameters to be altered. However, with slight modifications this framework can be utilized to not only measure the relevance of parameters but also determine which parameters are relevant to the scenario. Rather than simply nesting loops of predetermined parameters, the framework can be altered to randomly choose a handful of parameters from the entire set of parameters. Provided a weight function to measure the performance and executing over a large number of runs in a Monte Carlo fashion, a determination of relevant parameters would be made. This determination would aid in defining the run matrix. Most of the mechanisms required to facilitate the determination of parameter relevance are included in the framework outlined in this thesis. However, some work would be required to complete this task.

One final area of growth, expanding upon the random parameter premise above, is for parameter optimization. The framework, through some adaptations, could be extended for optimal path searching. With the addition of a weight functions and optimal search pathing such as a gradient search mechanism, rather than simple iteration, the framework can be utilized in an adaptive format. The framework itself could determine parameter values which optimally fit a solution.

CHAPTER 8
CONCLUSIONS

In this document the requirement for dynamically altering simulation parameters between executions has been discussed. Though an important concept, parameterization cannot be shown to exist in a reusable format under current simulation technology.

Beyond examining the requirement, a solution for dynamically determining parameters at run time using framework built upon reflection was described, along with an implementation of this framework. The framework is reusable, flexible, and requires minimal impact upon the encompassing simulation.

APPENDIX A
SOURCE LISTING FOR ABSORPTION SUBSYSTEM

```
'*****************************************************************
'*
'* Module Name: BaseProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide common base class for maintaining different property
'*    types. These property types are dependent upon underlying data type
'*    being manipulated.
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  CharProperty, IntegerProperty, NullProperty,
'*    ParseableProperty, StringProperty
'*
'* Implementation Notes: Abstract class
'*
'*****************************************************************
Public MustInherit Class BaseProperty
    ' Member attributes
    Private mPropertyOf As Object   'Reference to object whose property we are
                                    'manipulating
    Private mPropertyInfo As System.Reflection.PropertyInfo 'Information on
                                    'property being manipulated

    ' Property: Name
    ' Purpose: Provide accessor to name of property being manipulated.
    '       For example, if watching
    '       vehicle.x, this string would be set to 'x'
    Public ReadOnly Property Name() As String
        Get
            Return Me.PropertyInfo.Name
        End Get
    End Property
    ' Property: PropertyOf
    ' Purpose: Provide accessor to reference of instance being manipulated
    Public Property PropertyOf() As Object
        Get
            Return mPropertyOf
        End Get
        Set(ByVal value As Object)
            mPropertyOf = value
        End Set
    End Property
    ' Method: Parse
    ' Purpose: Provide common method by which a string value can be stored
    '       into the property being
    '       manipulated
    ' Design Notes: Abstract
    Public MustOverride Function Parse(ByVal aText As String) As Object
    ' Method: Add
    ' Purpose: Provide method by which an operend, as a string, can be used to
    '       manipulate underlying
    '       property data
    Public Overridable Sub Add(ByVal aOperend As String)
```

```
        Me.value = Me.value + Me.Parse(aOperend)
    End Sub
    ' Method: Subtract
    ' Purpose: Provide method by which an operend, as a string, can be used to
    '       manipulate underlying property data
    Public Overridable Sub Subtract(ByVal aOperend As String)
        Me.value = Me.value - Me.Parse(aOperend)
    End Sub
    ' Method: value
    ' Purpose: Return the value, properly typed, of the underlying data
    Public Property value() As Object
        Get
            Return Parse(valueText)
        End Get
        Set(ByVal value As Object)
            valueText = value.ToString
        End Set
    End Property
    ' Method: valueText
    ' Purpose: Provide access to the underlying data value, as a string
    Public Property valueText() As String
        Get
            Return PropertyInfo.GetValue(Me.PropertyOf, Nothing).ToString()
        End Get
        Set(ByVal value As String)
            PropertyInfo.SetValue(Me.PropertyOf, CObj(Me.Parse(value)), _
                  Nothing)
        End Set
    End Property
    ' Method: New
    ' Purpose: Constructor
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        Me.PropertyOf = aPropertyOf
        PropertyInfo = aPropertyOf.GetType.GetProperty(aPropertyName)
    End Sub
    ' Property: PropertyInfo
    ' Purpose: Provide accessor to information about the property being
    '       manipulated
    Protected Property PropertyInfo() As System.Reflection.PropertyInfo
        Get
            Return Me.mPropertyInfo
        End Get
        Set(ByVal value As System.Reflection.PropertyInfo)
            Me.mPropertyInfo = value
        End Set
    End Property
End Class
```

```
'*****************************************************************
'*
'* Module Name: CharProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide facilities for manipulating a property that is of char
'*      data type. Property values are manipulated by name.
'*
'* Supertypes/Interfaces: BaseProperty
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class CharProperty
    Inherits BaseProperty
    Public Overrides Function Parse(ByVal aText As String) As Object
        Return aText.Chars(0)
    End Function
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        MyBase.New(aPropertyOf, aPropertyName)
    End Sub
    Public Overrides Sub Add(ByVal aOperend As String)
        'Allow 'a' + 1 to be 'b', 'b'+2 to be 'd', etc...
        If IsNumeric(aOperend) Then
            Me.value = Chr(Asc(Me.value) + CInt(aOperend))
        End If
    End Sub
    Public Overrides Sub Subtract(ByVal aOperend As String)
        'Allow 'b' - 1 to be 'a', 'd'+2 to be 'b', etc...
        If IsNumeric(aOperend) Then
            Me.value = Chr(Asc(Me.value) - CInt(aOperend))
        End If
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: IntegerProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide facilities for manipulating a property that is of Integer
'*      data type. Property values are manipulated by name.
'*
'* Supertypes/Interfaces: BaseProperty
'* Known subtypes:  None
'*
'* Implementation Notes: Class is not required due to capabilities of
'*    ParsableProperty.
'*    This class was left to show how a numeric class should be written were
'*    it not to fall under the ParseableProperty realm.
'*
'*****************************************************************
Public Class IntegerProperty
    Inherits BaseProperty
    Public Overrides Function Parse(ByVal aText As String) As Object
        Try
            Return Integer.Parse(aText)
        Catch
            Throw New System.Exception("Integer Property Parse Error")
            Return 0
        End Try
    End Function
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        MyBase.New(aPropertyOf, aPropertyName)
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: NullProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide safe property type when one cannot be defined properly.
'*    If all code is working
'*    properly, a NullProperty instance should never exist
'*
'* Supertypes/Interfaces: BaseProperty
'* Known subtypes:  None
'*
'* Implementation Notes: Reference Null object design pattern
'*
'*****************************************************************
Public Class NullProperty
    Inherits BaseProperty
    Public Overrides Function Parse(ByVal aText As String) As Object
        MsgBox("Cannot parse NullProperty")
        Return 0
    End Function
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        MyBase.New(aPropertyOf, aPropertyName)
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: ParsableProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide facilities for manipulating any property which supports
'*    the method Parse(string)
'*    Hence, this class can represent any numeric data type in VB.Net.
'*    Manipulations to underlying
'*    data will take place based upon property name.
'*
'* Supertypes/Interfaces: BaseProperty
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class ParsableProperty
    Inherits BaseProperty
    Public Overrides Function Parse(ByVal aText As String) As Object
        Try
            Return PropertyInfo.GetValue(MyBase.PropertyOf, _
                  Nothing).Parse(aText)
        Catch
            MsgBox("Value must be an parsable.")
            Return 0
        End Try
    End Function
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        MyBase.New(aPropertyOf, aPropertyName)
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: PropertyFactory.vb
'* Module Type: Class
'*
'* Purpose: Provide factory creation facilities for creating Property
'*    instances
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes: Factory design pattern
'*
'*****************************************************************
Public Class PropertyFactory
    ' Method: isObjectParseable
    ' Purpose: Determine if an object implements the "Parse(string)" method
    Private Shared Function isObjectParseable(ByVal aObject As Object) As _
            Boolean
        Try
            Return Not aObject.GetMethod("Parse", New Type() _
                  {GetType(String)}) Is Nothing
        Catch
            Return False
        End Try
    End Function
    ' Method: createProperty
    ' Purpose: Provide creation facilities for BaseProperty instances
    Public Shared Function createProperty(ByRef aObject As Object, _
            ByVal aName As String) As BaseProperty

        'Look up the information about this property type
        Dim info As System.Reflection.PropertyInfo =  _
            aObject.GetType.GetProperty(aName)

        'Ensure info did actually find and determine the property in question.
        'If not, abort the factory and return nothing.
        If info Is Nothing Then
            Return Nothing
        End If

        'Based upon the data type of the property being manipulated, create
        '   different BaseProperty subtypes. Each subtype represents a
        '   different datatype being manipulated.
        If info.PropertyType Is GetType(System.Char) Then
            Return New CharProperty(aObject, aName)
        ElseIf info.PropertyType Is GetType(System.String) Then
            Return New StringProperty(aObject, aName)
        ElseIf info.PropertyType Is GetType(System.Int16) Then
            Return New IntegerProperty(aObject, aName)
        ElseIf isObjectParseable(info.PropertyType) Then
      'Parseable is left last such that any specific types would override
            Return New ParsableProperty(aObject, aName)
```

```
        Else
            MsgBox("Error!! Unhandled property type")
            Return New NullProperty(aObject, aName)
        End If
    End Function
End Class
```

```
'*****************************************************************
'*
'* Module Name: StringProperty.vb
'* Module Type: Class
'*
'* Purpose: Provide facilities for manipulating a property that is of
'*    string data type. Property
'*    values are manipulated by name.
'*
'* Supertypes/Interfaces: BaseProperty
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class StringProperty
    Inherits BaseProperty
    ' Method: Parse
    ' Overrides: BaseProperty.Parse
    Public Overrides Function Parse(ByVal aText As String) As Object
        Return aText
    End Function
    Public Sub New(ByRef aPropertyOf As Object, ByVal aPropertyName As String)
        MyBase.New(aPropertyOf, aPropertyName)
    End Sub
    ' Method: Add
    ' Overrides: BaseProperty.Add
    ' Implementation Notes: "Hello" + 1 yields "Hello1",
    ' "Hello1" + 1 yeilds "Hello2", etc...
    Public Overrides Sub Add(ByVal aOperend As String)
        Dim numericValue As Int32
        Dim currentPosition As Int32 = Me.valueText.Length + 1

        'Seprate out the numeric post-fix from the textual prefix
        Do
            currentPosition = currentPosition - 1
        Loop While IsNumeric(Mid(Me.valueText, currentPosition))

        If currentPosition >= Me.valueText.Length Then
            numericValue = 0
        Else
            numericValue = CInt(Mid(Me.valueText, currentPosition + 1))
            Me.valueText = Mid(Me.valueText, 1, currentPosition)
        End If

        'The resultant string is the textual prefix post-fixed with the sum
        ' of the initial
        ' numeric post-fix and the integer value passed in as the operend
        Me.valueText = Me.valueText & CStr(numericValue + CInt(aOperend))
    End Sub
    Public Overrides Sub Subtract(ByVal aOperend As String)
        Dim numericValue As Int32
        Dim currentPosition As Int32 = Me.valueText.Length + 1
```

```
        'Seprate out the numeric post-fix from the textual prefix
        Do
            currentPosition = currentPosition - 1
        Loop While IsNumeric(Mid(Me.valueText, currentPosition))

        If currentPosition >= Me.valueText.Length Then
            numericValue = 0
        Else
            numericValue = CInt(Mid(Me.valueText, currentPosition + 1))
            Me.valueText = Mid(Me.valueText, 1, currentPosition)
        End If

        'The resultant string is the textual prefix post-fixed with the sum
        '   of the initial
        '   numeric post-fix and the integer value passed in as the operend
        Me.valueText = Me.valueText & CStr(numericValue - CInt(aOperend))
    End Sub
End Class
```

APPENDIX B:
SOURCE LISTING FOR PARAMETERIZATION SUBSYSTEM

```
'*****************************************************************
'*
'* Module Name: Iterator.vb
'* Module Type: Class
'*
'* Purpose: This is an abstract class which allows Iterators to behave in a
'*    common manner.
'*    An Iterator is a class which can alter a property through a series of
'*    values.
'*    The value will change between subsequent calls to the increment()
'*    method.
'*
'* Supertypes/Interfaces: Class
'* Known subtypes: DiscreteIterator, LinearIterator
'*
'* Implementation Notes: Abstract class
'*
'*****************************************************************
Public MustInherit Class Iterator
    'Member attributes
    Private mInitialValue As String 'Starting value for the iterator
    Private mProperty As Properties.BaseProperty    'Property/attribute being
altered via the iterator
    Private mTag As String  'A string which can be used to describe
property/attribute being altered

    'Abstract methods
    Public MustOverride Function increment() As Boolean 'Go to the next
sequential value
    Public MustOverride Sub edit()  'Edit the properties for this iterator
    Public MustOverride Sub initialize()    'Reset the iteration to initial
values
    ' Method: tag
    ' Purpose: Proerty allows for descriptive text to describe
attribute/property being
    '       manipulated
    Public Property tag() As String
        Get
            Return mTag
        End Get
        Set(ByVal value As String)
            mTag = value
        End Set
    End Property
    ' Method: iteratedProperty
    ' Purpose: Provide access to attributge/property being manipulated
    Protected Property iteratedProperty() As Properties.BaseProperty
        Get
            Return mProperty
        End Get
        Set(ByVal value As Properties.BaseProperty)
            mProperty = value
        End Set
```

```
    End Property
    ' Method: operator =
    ' Purpose: Determines if two iterators are equivalent.
    ' Design notes: To be equivalent, two
    '       iterators are not necessarily the same reference. Rather, if two
    '       iterators are attempting to iterate over the same
property/attribute then
    '       they are considered equivalent.
    Public Shared Operator =(ByVal aIterator1 As Iterator, ByVal aIterator2 As
Iterator) As Boolean
        Return aIterator1.iteratedProperty.PropertyOf Is
aIterator2.iteratedProperty.PropertyOf And aIterator1.iteratedProperty.Name =
aIterator2.iteratedProperty.Name
    End Operator
    ' Method: operator <>
    ' Purpose: Determines if two iterators not equivalent.
    ' Design notes: Simpple opposite of = operator
    Public Shared Operator <>(ByVal aIterator1 As Iterator, ByVal aIterator2
As Iterator) As Boolean
        Return Not aIterator1 = aIterator2
    End Operator
    ' Method: valueText
    ' Purpose: String value of the property being referenced, at the current
iteration.
    Public ReadOnly Property valueText() As String
        Get
            Return iteratedProperty.valueText
        End Get
    End Property
    ' Method: initialValue
    ' Purpose: Provide access to value of property prior to any iterations.
    Protected Property initialValue() As String
        Get
            Return mInitialValue
        End Get
        Set(ByVal value As String)
            mInitialValue = value
        End Set
    End Property
    ' Method: reset
    ' Purpose: Return iterator to initial state, piror to any iterations being
performed.
    Public Sub reset()
        iteratedProperty.valueText = initialValue
    End Sub
    ' Method: New
    ' Purpose: Initialize Iterator upon creation.
    ' Overrides: Constructor
    Public Sub New(ByVal aProperty As Properties.BaseProperty)
        iteratedProperty = aProperty
        initialValue = aProperty.valueText
        mTag = "Unnamed"
    End Sub
```

```
    ' Method: toString
    ' Purpose: Provide a textual representation of the data contained in an
iterator
    Public Overrides Function toString() As String
        Return "Tag: " & Me.tag & " Property: " & Me.iteratedProperty.Name
    End Function
EndClass
```

```
'*****************************************************************
'*
'* Module Name: DiscreteIterator.vb
'* Module Type: Class
'*
'* Purpose: This class represents a discrete iterator. A discrete iterator
'*      is one that comprises of a discrete list of values to be iterated
'*      over. These values are intended to be unrelated.
'*
'* Supertypes/Interfaces: Iterator
'* Known subtypes: None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class DiscreteIterator
    'Supertypes/interfaces suported
    Inherits Iterator

    'Member attributes
    Private mCurrentIndex As Int32  'Used to keep track of current iteration
    'element
    Private mList As New System.Collections.ArrayList   'List of discrete
values attribute can take

    ' Method: currentIndex
    ' Purpose: Store/retrieve the current value of the positional index
    '           which keeps track of the current discrete element of
    '           the iteration
    Private Property currentIndex() As Int32
        Get
            Return mCurrentIndex
        End Get
        Set(ByVal value As Int32)
            mCurrentIndex = value
        End Set
    End Property
    ' Method: items
    ' Purpose: Store/retrieve the list of values that the property can take
    ' Design notes: ReadOnly property such that reference to collection
    '       cannot be destroyed.
    Public ReadOnly Property items() As System.Collections.ArrayList
        Get
            Return mList
        End Get
    End Property
    ' Method: edit
    ' Overrides: Iterator.edit
    ' Purpose: This method provides access to an editor capable of editing
    '   a discrete list of values
    Public Overrides Sub edit()
        Dim form As New DiscreteIteratorEditor(Me)
```

```
        form.ShowDialog()
    End Sub
    ' Method: initialize
    ' Overrides: Iterator.initialize
    ' Purpose: Set the state for starting a new set of iterations
    ' #pre: none
    ' #post: currentIndex = 0, iteratedProperty.valueText = items.Item(0)
    Public Overrides Sub initialize()
        currentIndex = 0
        Me.iteratedProperty.valueText = items.Item(currentIndex)
    End Sub
    ' Method: increment
    ' Overrides: Iterator.increment
    ' Purpose: Set the parameterized value to the next entry in the collection
    ' #pre: currentIndex >= 0
    ' Design notes: Iterative function. If this element in the collection
    '   can be incremented safely, then do so. Otherwise, attempt to iterate
    '   the next element in the collection. Return value will be true if
    '   anything in the collection can be incremented. Otherwise the
    '   return value will be false.
    Public Overrides Function increment() As Boolean
        Debug.Assert(currentIndex >= 0)
        currentIndex = currentIndex + 1
        If currentIndex >= items.Count Then
            Return False
        Else
            Me.iteratedProperty.valueText = mList.Item(currentIndex)
            Return True
        End If
    End Function
    ' Method: New
    ' Overrides: Constructor
    Public Sub New(ByVal aProperty As Properties.BaseProperty)
        MyBase.New(aProperty)   'Must have reference to underlying attribute
        mCurrentIndex = 0   'Initialize to point to the first item in the list
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: DiscreteIteratorEditor.vb
'* Module Type: Form
'*
'* Purpose: This class is the internal editor for a DiscreteEditor.
'*      It allows for the editing of a simple lest of discrete elements.
'*
'* Supertypes/Interfaces: Form
'* Known subtypes: None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class DiscreteIteratorEditor
    Private mData As DiscreteIterator   'Reference to DicreteIterator being
edited
    ' Method: data
    ' Purpose: Store/retrieve the DiscreteEditor being edited
    Private Property data() As DiscreteIterator
        Get
            Return mData
        End Get
        Set(ByVal value As DiscreteIterator)
            mData = value
        End Set
    End Property
    ' Method: New
    ' Purpose: Initialize new instance of DiscreteIteratorEditor
    ' Overrides: Constructor
    Public Sub New(ByVal aData As DiscreteIterator)

        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        mData = aData   'Maintain reference to data being edited

        Dim lString As String

        'Iterate through the list of discrete entries already contained within
the DiscreteIterator
        'and populate the screen with these for editing purposes.
        For Each lString In data.items
            tbItems.Text = tbItems.Text & lString & vbNewLine
        Next
    End Sub
    ' Method: btnOK_Click
    ' Purpose: When the user hits the ok button on the screen, store the
edited
    '       list of strings back into the DiscreteIterator. Then hide the
form.
    ' Overrides: button click event handler
```

```
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnOK.Click
        Dim lIndex As Int32
        Dim lTrimmedString As String

        data.items.Clear()  'Destroy the outdated list of entries in the
DiscreteIterator
        For lIndex = 0 To tbItems.Lines.Length - 1  'Loop over new entries,
adding each to DiscreteIterator
            lTrimmedString = Trim(tbItems.Lines(lIndex))
            If lTrimmedString.Length <> 0 Then
                data.items.Add(lTrimmedString)
            End If
        Next
        Me.Hide()
    End Sub
    ' Method: btnCancel_Click
    ' Purpose: When the user hits the 'Cancel' button, hide this form. Do not
save any
    '       changes to the edited data.
    ' Overrides: button click event handler
    Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCancel.Click
        Me.Hide()
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: IteratorFactory.vb
'* Module Type: Class
'*
'* Purpose: Create instances of Iterators
'*
'* Supertypes/Interfaces: Class
'* Known subtypes:  None
'*
'* Implementation Notes: Factory creational design pattern for creating
instances of Iterator
'*
'*****************************************************************
Public Class IteratorFactory
    ' Data type: etIteratorType
    ' Purpose: Defines the different types of iterators which can be created
    Public Enum etIteratorType
        eLinear     'Sequental with even spacing such as 5, 10, 15, 20...
        eDiscrete   'Discrete list with no dependencies between entries
    End Enum
    ' Method: createIterator
    ' Purpose: Create a new iterator whose subtype is defined by the input
parameters
    Public Shared Function createIterator(ByVal aType As etIteratorType, _
        ByVal aProperty As Properties.BaseProperty) As Iterator

        If TypeOf aProperty Is Properties.NullProperty Then
            Return New NullIterator(aProperty)
        End If

        If aType = etIteratorType.eLinear Then
            Return New LinearIterator(aProperty)
        ElseIf aType = etIteratorType.eDiscrete Then
            Return New DiscreteIterator(aProperty)
        Else
            Return New NullIterator(aProperty)
        End If
    End Function
    ' Method: createIterator
    ' Purpose: Create a new iterator whose subtype is defined by the input
parameters
    Public Shared Function createIterator(ByVal aType As etIteratorType, _
        ByVal aObject As Object, ByVal aPropertyName As String) As Iterator
        Dim lProperty As Properties.BaseProperty

        lProperty = Properties.PropertyFactory.createProperty(aObject,
aPropertyName)
        Return createIterator(aType, lProperty)
    End Function
End Class
```

```
'*****************************************************************
'*
'* Module Name: LinearIterator.vb
'* Module Type: Class
'*
'* Purpose: Represents a list of sequential entries
'*
'* Supertypes/Interfaces: Iterator
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class LinearIterator
    'Interfaces/supertypes
    Inherits Iterator

    'Member attributes
    Private mStart As String        'Starting value for sequence
    Private mStep As String         'Spacing between entry(n) and entry(n+1)
    Private mMaxSteps As Int32      'Number of elements in sequence
    Private mCurrentStep As Int32   'Current placement within the sequence

    ' Property: start
    ' Purpose: Accessor for mStart
    Public Property start() As String
        Get
            Return mStart
        End Get
        Set(ByVal value As String)
            mStart = value
        End Set
    End Property

    ' Property: maxSteps
    ' Purpose: Accessor for mMaxSteps
    Public Property maxSteps() As Int32
        Get
            Return mMaxSteps
        End Get
        Set(ByVal value As Int32)
            If value < 1 Then   'Cannot have less than 1 steps in the setf
                MsgBox("Steps must be set to >= 1. Setting to 1")
                mMaxSteps = 1
            Else
                mMaxSteps = value
            End If
        End Set
    End Property
    ' Property: currentStep
    ' Purpose: Accessor for mCurrentStep
    Public Property currentStep()
        Get
```

```
            Return mCurrentStep
        End Get
        Private Set(ByVal value)
            mCurrentStep = value
        End Set
    End Property
    ' Property: stepSize
    ' Purpose: Accessor for mStep
    Public Property stepSize() As String
        Get
            Return mStep
        End Get
        Set(ByVal value As String)
            Try
                Me.iteratedProperty.Parse(value)  'Ensure this is the right
data type
                mStep = value
            Catch ex As Exception
                Throw New System.Exception("Invalid step size")
                mStep = "1"
            End Try
        End Set
    End Property
    ' Method: initialize
    ' Purpose: Initialize the iterator. To do this, set currentStep to 0 and
value to initial alue
    Public Overrides Sub initialize()
        currentStep = 0
        Me.iteratedProperty.valueText = Me.start
    End Sub
    'Method: increment
    'Purpose: Increment to the next value for the iterator
    'Overrides: Iterator.increment
    Public Overrides Function increment() As Boolean
        currentStep = currentStep + 1
        If currentStep >= maxSteps Then
            Return False
        Else
            Me.iteratedProperty.Add(stepSize)
            Return True
        End If
    End Function
    'Method: edit
    'Purpose: create GUI editor to edit this instance with
    Public Overrides Sub edit()
        Dim form As New LinearIteratorEditor(Me)

        form.ShowDialog()
    End Sub
    'Method: New
    'Purpose: Property initialize LinearIterator
    Public Sub New(ByVal aProperty As Properties.BaseProperty)
        MyBase.New(aProperty)
```

```
        start = aProperty.valueText
        mStep = "1"
        mMaxSteps = 1
        mCurrentStep = 0
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: LinearIteratorEditor.vb
'* Module Type: Form
'*
'* Purpose: Provide GUI access for editing LinearIterators.
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class LinearIteratorEditor
    'Member attributes
    Private mData As LinearIterator 'Reference to LinearIterator being edited

    ' Property: data
    ' Purpose: Provide access to LinearIterator being edited
    Private Property data() As LinearIterator
        Get
            Return mData
        End Get
        Set(ByVal value As LinearIterator)
            mData = value
        End Set
    End Property
    ' Method: New
    ' Purpose: Initialize LinearIteratorEditor when created
    Public Sub New(ByVal aData As LinearIterator)
        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        data = aData
        txtStep.Text = data.stepSize
        txtStart.Text = data.start
        txtCount.Text = data.maxSteps.ToString
    End Sub

    ' Method: btnOK_Click
    ' Purpose: When user clicks ok button, submit changes and close the editor
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnOK.Click
        Try
            data.stepSize = txtStep.Text
            data.maxSteps = Int32.Parse(txtCount.Text)
            data.start = txtStart.Text
            Me.Hide()
        Catch ex As Exception
            MsgBox("Invalid data")
        End Try
    End Sub
```

```
    ' Method: btnCancel_Click
    ' Purpose: When user cancels editing process, close the form but do not
save data
    Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCancel.Click
        Me.Hide()
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: LinearIteratorEditor.vb
'* Module Type: Form
'*
'* Purpose: Provide GUI access for editing LinearIterators.
'*
'* Supertypes/Interfaces: Iterator
'* Known subtypes:  None
'*
'* Implementation Notes: Null object pattern. Class performs no actions.
'*      It is simply a safe placeholder for valid Iterators
'*
'*****************************************************************
Public Class NullIterator
    'Define inheritence
    Inherits Iterator

    ' Method: edit
    ' Purpose: Provide access to editing capabilities by popping up a form
    Public Overrides Sub edit()
        MsgBox("Invalid iterator. Cannot edit")
    End Sub

    ' Method: increment
    ' Purpose: Allow data to be incremented
    Public Overrides Function increment() As Boolean
        MsgBox("Invalid iterator. Cannot increment.")
    End Function
    ' Method: New
    ' Purpose: Constructor
    Public Sub New(ByVal aProperty As Properties.BaseProperty)
        MyBase.New(aProperty)
    End Sub
    ' Method: initialize
    ' Purpose: Initialize iterator to first value in set
    Public Overrides Sub initialize()
        MsgBox("Invalid iterator. Cannot initialize.")
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: Parameterization.vb
'* Module Type: Class
'*
'* Purpose: Maintain list of individual iterators and provide access methods
to this underlying data
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes: Provides the 'Observed' portion of Observer pattern
'*
'*****************************************************************
Public Class Parameterization
    'Member attributes
    Private mIterations As System.Collections.Generic.LinkedList(Of Iterator)
'List of iterations
    Private mObservers As System.Collections.Generic.LinkedList(Of
ParameterizationObserver)    'List of observers for observer pattern
    ' Method: registerObserver
    ' Purpose: Part of observer pattern. Allow observers to register with this
class for notification
    Public Sub registerObserver(ByVal aObserver As ParameterizationObserver)
        'Ensure observer not listed twice.
        observers.Remove(aObserver)

        'Add the observer to the list
        observers.AddLast(aObserver)
    End Sub
    ' Method: unregisterObserver
    ' Purpose: Allow observers to unregister for notifications
    ' Design notes: Part of observer design pattern
    Public Sub unregisterObserver(ByVal aObserver As ParameterizationObserver)
        observers.Remove(aObserver)
    End Sub
    ' Method: notifyIterationComplete
    ' Purpose: Notify all observers that a single iteration has been completed
    Private Sub notifyIterationComplete()
        Dim lObserver As ParameterizationObserver

        For Each lObserver In observers
            lObserver.iterationComplete()
        Next
    End Sub
    ' Method: notifyParameterizationComplete
    ' Purpose: Notify all observers that the entire parameterization set has
been completed
    Private Sub notifyParameterizationComplete()
        Dim lObserver As ParameterizationObserver

        For Each lObserver In observers
            lObserver.parameterizationComplete()
```

```
        Next
    End Sub
    ' Property: observers
    ' Purpose: Provide accessors to list of observers
    Private Property observers() As System.Collections.Generic.LinkedList(Of
ParameterizationObserver)
        Get
            Return mObservers
        End Get
        Set(ByVal value As System.Collections.Generic.LinkedList(Of
ParameterizationObserver))
            mObservers = value
        End Set
    End Property
    ' Property: iterations
    ' Purpose: Provide accessors to list of iterations
    Public Property iterations() As System.Collections.Generic.LinkedList(Of
Iterator)
        Get
            Return mIterations
        End Get
        Private Set(ByVal value As System.Collections.Generic.LinkedList(Of
Iterator))
            mIterations = value
        End Set
    End Property
    ' Method: New
    ' Purpose: Constructor
    Public Sub New()
        mIterations = New System.Collections.Generic.LinkedList(Of Iterator)
        observers = New System.Collections.Generic.LinkedList(Of
ParameterizationObserver)
    End Sub
    ' Method: moveUp
    ' Purpose: Provide sorting capabilities for iterators being mainted in the
iterator list
    Public Sub moveUp(ByVal aIterator As Iterator)
        Dim lNode As System.Collections.Generic.LinkedListNode(Of Iterator)
        Dim lPreviousNode As System.Collections.Generic.LinkedListNode(Of
Iterator)

        'Find the item in the collection
        lNode = iterations.Find(aIterator)

        'If item being reordered isn't found or is already in the first
position, abort the sorting
        If lNode Is Nothing Or lNode Is iterations.First Then
            Exit Sub
        End If
        lPreviousNode = lNode.Previous

        'Remove iterator from the current position and replace at the newly
defined position
```

```
        iterations.Remove(aIterator)
        iterations.AddBefore(lPreviousNode, aIterator)
    End Sub
    ' Method: moveDown
    ' Purpose: Provide sorting capabilities for iterators being matined in the
iterator list
    Public Sub moveDown(ByVal aIterator As Iterator)
        Dim lNode As System.Collections.Generic.LinkedListNode(Of Iterator)
        Dim lNextNode As System.Collections.Generic.LinkedListNode(Of
Iterator)

        'Find the item in the collection
        lNode = iterations.Find(aIterator)

        'If we can't find the referenced item in the list or already at the
end of the list, abort the sort
        If lNode Is Nothing Or lNode Is iterations.Last Then
            Exit Sub
        End If
        lNextNode = lNode.Next

        'Remove from current position and place in desired position
        iterations.Remove(aIterator)
        iterations.AddAfter(lNextNode, aIterator)
    End Sub
    ' Method: remove
    ' Purpose: Provide facility by which an iterator can be removed from the
collection
    Private Sub remove(ByVal aIterator As Iterator)
        iterations.Remove(aIterator)
    End Sub
    ' Method: findIteratorInList
    ' Purpose: Provide facilities to search for a particular iterator.
    ' Design notes: Linear search
    '       Written separately from collection's find() method because find()
will search
    '       for items referencing the same instance. The find written here,
however, will
    '       search for items via the comparison (=) operator such that though
they may not
    '       be the same instance, they represent the same data
    Private Function findIteratorInList(ByVal aIterator As Iterator) As
Iterator
        Dim lIterator As Iterator

        For Each lIterator In iterations
            If lIterator = aIterator Then Return lIterator
        Next
        Return Nothing
    End Function
    ' Method: AddTail
    ' Purpose: Provide facility for adding an iterator to the collection
    Public Function AddTail(ByVal aIterator As Iterator) As Iterator
```

```
        'First ensure this paramterization does not already exist. If it does,
don't add it
        'Simply return the currently existing one
        Dim lIterator As Iterator = findIteratorInList(aIterator)

        If Not lIterator Is Nothing Then
            iterations.Remove(lIterator)
        End If

        iterations.AddLast(aIterator)
        Return aIterator
    End Function
    ' Method: AddTail
    ' Purpose: Provide facility for adding an iterator to the collection
    Public Function AddTail(ByVal aType As IteratorFactory.etIteratorType, _
        ByVal aObject As Object, ByVal aPropertyName As String) As Iterator

        Dim lIterator As Iterator = IteratorFactory.createIterator(aType,
aObject, aPropertyName)

        AddTail(lIterator)
        Return lIterator
    End Function
    ' Method: incrementNode
    ' Purpose: Attempt to increment a single loop in the nested loop
structure.
    '   If successful, return true. Otherwise, attempt to increment for the
next outer loop.
    ' Design notes: Recursive function
    Private Function incrementNode(ByVal aNode As
System.Collections.Generic.LinkedListNode(Of Iterator)) As Boolean
        Dim lData As Iterator

        lData = aNode.Value

        'Attempt to increment the currently chosen loop
        If lData.increment Then
            'If successful, return true
            Return True
        ElseIf aNode Is iterations.Last Then
            'If not successful at incrementing current loop and this is the
shallowest loop
            'in the nest, return false since we cannot loop any more
            Return False
        Else
            'Couldn't increment the current loop but possibly more iterations
can be performed.
            'Reset this node to initial value and attempt to increment next
outer loop
            If incrementNode(aNode.Next) Then
                lData.initialize()
                Return True
            Else
```

```
                Return False
            End If
        End If
    End Function
    ' Method: increment
    ' Purpose: Perform a single incrementation of the loop structure. This is
attempted by first
    '   looping once for the innermost loop. If successful, task is complete.
Otherwise, increment
    '   the next outer loop until either a) a successful incrementation was
completed b) All loops
    '   have been exhausted and therefore we cannot loop further
    Public Function increment() As Boolean
        If Not iterations.First Is Nothing Then
            If incrementNode(iterations.First) Then
                notifyIterationComplete()
                Return True
            Else
                notifyParameterizationComplete()
                Return False
            End If
        Else
            Return False
        End If
    End Function
    ' Method: incrementAll
    ' Purpose: Increment through all values present in the iterations.
    Public Sub incrementAll()
        initialIteration()

        Do While (increment())
            'Nothing to do here. Just wait for all increments to complete
        Loop
        reset()
    End Sub
    ' Method: reset
    ' Purpose: Reset all iterations to pre-execution values. This is normally
used after all
    '   iterations are complete to set data to pre-run conditions.
    Public Sub reset()
        Dim lIteration As Iterator

        For Each lIteration In iterations
            lIteration.reset()
        Next
    End Sub
    ' Method: initialIteration
    ' Purpose: Initialize all attributes which are part of the overall
iteration process
    '   to the first position.
    Public Sub initialIteration()
        Dim lIteration As Iterator
```

```
        For Each lIteration In iterations
            lIteration.initialize()
        Next

        'Initial iteration counts as an iteraton, so notify accordingly
        notifyIterationComplete()
    End Sub
    ' Method: edit
    ' Purpose: Display GUI form for editing the entire collection of
parameterizations
    Public Sub edit()
        Dim form As New ParameterizationSorter(Me)

        form.ShowDialog()
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: ParameterizationObserver.vb
'* Module Type: Interface
'*
'* Purpose: Provide common interface for notification of parameterization
events
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes: Provides interface for the 'Observer' portion of
Observer pattern
'*
'*****************************************************************
Public Interface ParameterizationObserver
    Sub iterationComplete()         'A single update to the nested loop
structure has been completed
    Sub parameterizationComplete()  'All loops in the nested loop structure
have been fully exhausted
End Interface
```

```
'*****************************************************************
'*
'* Module Name: ParameterizationSorter.vb
'* Module Type: Form
'*
'* Purpose: Provide GUI form for sorting list of iterators in a
parameterization
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class ParameterizationSorter
    'Member attributes
    Private mParameterization As Parameterization   'Reference to
parameterizaton being edited
    ' Property: param
    ' Purpose: Provide accessor to underlying data which is currently being
edited
    Private Property param() As Parameterization
        Get
            Return mParameterization
        End Get
        Set(ByVal value As Parameterization)
            mParameterization = value
        End Set
    End Property
    ' Method: repopulate
    ' Purpose: Take all data from the Parameterization being edited and push
it out to the form
    '   for display purposes
    Private Sub repopulate()
        Dim lItem As Iterator

        lstData.Items.Clear()
        For Each lItem In param.iterations
            lstData.Items.Add(lItem)
        Next
    End Sub
    ' Method: New
    ' Purpose: Constructor
    Public Sub New(ByVal aParam As Parameterization)
        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        param = aParam
        repopulate()
    End Sub
    ' Method: btnOK_Click
```

```
    ' Purpose: Event handler for clicking OK button on GUI. Simply hide the
form.
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnOK.Click
        Me.Hide()
    End Sub
    ' Method: btnUp_Click
    ' Purpose: Event handler for clicking the 'Up' button on the form. Move
the currently
    '   slected nest of the structure up by one level
    Private Sub btnUp_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnUp.Click
        If Not lstData.SelectedItem Is Nothing Then
            param.moveUp(lstData.SelectedItem)
            repopulate()
        End If
    End Sub
    ' Method: btnDown_Click
    ' Purpose: Event handler for clicking the 'Down' button on the form. Move
the currently
    '   slected nest of the structure down by one level
    Private Sub btnDown_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnDown.Click
        If Not lstData.SelectedItem Is Nothing Then
            param.moveDown(lstData.SelectedItem)
            repopulate()
        End If
    End Sub
    ' Method: btnEdit_Click
    ' Purpose: Event handler for clicking the 'Edit' button on the form. Open
the editor for the
    '   currently selected iteration
    Private Sub btnEdit_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEdit.Click
        If Not lstData.SelectedItem Is Nothing Then
            Dim lItem As Iterator
            lItem = lstData.SelectedItem
            lItem.edit()
        End If
    End Sub
End                                                                    Class
```

APPENDIX C
SOURCE LISTING FOR APPLICATION

```
'*****************************************************************
'*
'* Module Name: AcousticCalculator.vb
'* Module Type: Class
'*
'* Purpose: Provide utility methods for acoustic calculations
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes: Utility class, all shared methods
'*
'*****************************************************************
Public Class AcousticCalculator
    ' Method: calculateTransmissionLoss
    ' Purpose: Calculate transmission loss of acoustic signal
    ' Implmentation: Assuming 20*log(R) transmission loss. This assums
    '   1) spherical spreading (deep water)
    '   2) Not frequency dependent
    Public Shared Function transmissionLoss(ByVal aRange As Double) As Double
        Return 20.0 * Math.Log10(aRange)
    End Function
    ' Method: convertDbToPower
    ' Purpose: Convert acoustic decibel level to a power level
    Public Shared Function convertDbToPower(ByVal aDb As Double) As Double
        Return Math.Pow(10.0, aDb / 10.0)
    End Function
    ' Method: convertPowerToDB
    ' Purpose: Convert acoustic power level to decibels
    Public Shared Function convertPowerToDB(ByVal aPower As Double) As Double
        Return 10.0 * Math.Log10(aPower)
    End Function
    ' Method: range
    ' Purpose: Calculate range between two targets given their coordinate
values
    Private Shared Function range(ByVal aX1 As Double, ByVal aY1 As Double,
ByVal aZ1 As Double, _
        ByVal aX2 As Double, ByVal aY2 As Double, ByVal aZ2 As Double) As
Double
        Return Math.Sqrt((aX1 - aX2) * (aX1 - aX2) + (aY1 - aY2) * (aY1 - aY2)
+ (aZ1 - aZ2) * (aZ1 - aZ2))
    End Function
    ' Method: calculateSignalExcess
    ' Purpose: Calculate acoustic recieve level, in excess of noise
    ' Implementation Notes: Equation based on SE = SL-TL-NL
    '   where SE is signal excess, TL is transmission loss, NL is noise level,
DI is directivity index
    Public Shared Function calculateSignalExcess(ByVal aX1 As Double, ByVal
aY1 As Double, ByVal aZ1 As Double, _
        ByVal aX2 As Double, ByVal aY2 As Double, ByVal aZ2 As Double, ByVal
aSourceLevel As Double, ByVal aNoise As Double)
```

```
        Return aSourceLevel - transmissionLoss(range(aX1, aY1, aZ1, aX2, aY2,
aZ2)) - aNoise
    End Function
End Class
```

```
'*****************************************************************
'*
'* Module Name: AlphaMine.vb
'* Module Type: Class
'*
'* Purpose: Provides implmentation of a Mine logic.
'*
'* Supertypes/Interfaces: Mine
'* Known subtypes:  None
'*
'* Implementation Notes: This mine works as a simple energy detector. I
'*      recieve level is larger than threshold, mine will fire. No integration
of signal.
'*
'*****************************************************************
Public Class AlphaMine
    Implements Mine
    Private Const cThreshHold = -7.0    'Firing threshold
    Private mFire As Boolean
    Private Property fire() As Boolean
        Get
            Return mFire
        End Get
        Set(ByVal value As Boolean)
            mFire = value
        End Set
    End Property
    Public Sub reset() Implements Mine.reset
        fire = False
    End Sub
    ' Method: update
    ' Overrides: Mine.update
    ' Implentation Notes: Works as simple energy threshold detector. If RL >
Threshold, fire.
    Public Sub update(ByVal aRecieveLevel As Double) Implements Mine.update
        If aRecieveLevel > cThreshHold Then
            fire = True
        End If
    End Sub
    ' Method: Constructor
    Public Sub New()
        mFire = False
    End Sub
    ' Property: hasFired
    Public Property hasFired() As Boolean Implements Mine.hasFired
        Get
            Return fire
        End Get
        Set(ByVal value As Boolean)
            fire = value
        End Set
    End Property
End Class
```

```
'*****************************************************************
'*
'* Module Name: BetaMine.vb
'* Module Type: Class
'*
'* Purpose: Provides implmentation of a Mine logic.
'*
'* Supertypes/Interfaces: Mine
'* Known subtypes:  None
'*
'* Implementation Notes: This mine works as a simple energy detector with
integration.
'*      If the integration period is full and the integrated signal is greter
than
'*      threshold, the mine will fire
'*
'*****************************************************************
Public Class BetaMine
    Implements Mine
    Private Const cThreshHold As Double = -10.2     'Firing threshold, in
decibels
    Private Const cSampleThreshold As Int32 = 30    'Number of samples to
integrate
    Private mIntegratorBins As System.Collections.Generic.LinkedList(Of
Double) 'List of stored values for integrator
    Private mFire As Boolean        'Mine has fired
    Private Property integrator() As System.Collections.Generic.LinkedList(Of
Double)
        Get
            Return mIntegratorBins
        End Get
        Set(ByVal value As System.Collections.Generic.LinkedList(Of Double))
            mIntegratorBins = value
        End Set
    End Property
    Private Property fire() As Boolean
        Get
            Return mFire
        End Get
        Set(ByVal value As Boolean)
            mFire = value
        End Set
    End Property
    Public Sub reset() Implements Mine.reset
        fire = False
        integrator.Clear()
    End Sub
    Public Sub update(ByVal aRecieveLevel As Double) Implements Mine.update
        'Power levels, not DB levels are stored into the list so convert, then
add most current to list

integrator.AddFirst(AcousticCalculator.convertDbToPower(aRecieveLevel))
```

```
        'Ensure integration buffer is full before attempting to fire
        If integrator.Count > cSampleThreshold Then
            'Remove the last item (first stored, FIFO) from the integrator
            integrator.RemoveLast()

            Dim lCurrent As Double
            Dim lSum As Double

            'Sum up contents of integrator
            lSum = 0.0
            For Each lCurrent In integrator
                lSum = lSum + lCurrent
            Next

            'Test for threshold
            If (AcousticCalculator.convertPowerToDB(lSum /
CDbl(cSampleThreshold))) > cThreshHold Then
                fire = True
            End If
        End If
    End Sub
    Public Sub New()
        mFire = False
        integrator = New System.Collections.Generic.LinkedList(Of Double)
    End Sub
    Public Property hasFired() As Boolean Implements Mine.hasFired
        Get
            Return fire
        End Get
        Set(ByVal value As Boolean)
            fire = value
        End Set
    End Property
End Class
```

```
'*****************************************************************
'*
'* Module Name: Form1.vb
'* Module Type: Form
'*
'* Purpose: Provide GUI for managing simulation
'*
'* Supertypes/Interfaces: Parameterization.ParameterizationObserver
'* Known subtypes:  None
'*
'* Implementation Notes: Registers as 'observer' portion of observer design
pattern.
'*
'*****************************************************************
Public Class Form1
    Implements Parameterization.ParameterizationObserver

    'Member attributes
    Private mParam As New Parameterization.Parameterization 'Parameterizations
    Private mVehicle As Vehicle     'Single vehicle in the simulation
    Private mMine As MinePosition   'Single mine in the simulation
    Private Const cNoise As Double = 98.0   'Environmental acoustic noise
level
    ' Method: iterationComplete
    ' Overrides: Parameterization.ParameterizationObserver.iterationComplete
    ' Purpose: When a loop iteration of the parameterization has completed,
update mine logic
    '   to determine if mine fires
    Public Sub iterationComplete() Implements
Parameterization.ParameterizationObserver.iterationComplete
        Dim lSignalExcess As Double
        Dim i As Double

        mine.mine.reset()

        'Run vehicle along track beside mine. Note: This loop could be
accomplished via the
        'parameterization layer as well. However, a design choice was made to
short-circuit a mine
        'firing. Once a mine fires, there is no need to continue for the
current track.
        For i = 500 To -500.0 Step -1
            vehicle.x = i

            'Determine singal excess at the mine
            lSignalExcess =
AcousticCalculator.calculateSignalExcess(vehicle.x, vehicle.y, vehicle.z, _
                mine.x, mine.y, mine.z, vehicle.signature, cNoise)

            'Update mine logic
            mine.mine.update(lSignalExcess)
```

```
            'If the mine has fired, display results on the screen and run next
track
            If mine.mine.hasFired Then
                TextBox2.Text = TextBox2.Text & "Fire: " & mine.mineType & " "
& " SL:" & vehicle.signature & " X:" & vehicle.x & " Y:" & vehicle.y & " Z:" &
vehicle.z & vbNewLine
                mine.mine.hasFired = False
                Exit For
            End If
        Next
    End Sub
    ' Method: parameterizationComplete
    ' Overrides:
Parameterization.ParameterizationObserver.parameterizationComplete
    ' Purpose: This is called when all loops have been exhaused in a
parameterization.
    '   Currently nothing happens, short of a message being displayedon the
screen.
    Public Sub parameterizationComplete() Implements
Parameterization.ParameterizationObserver.parameterizationComplete
        MsgBox("Simulation completed")
    End Sub
    ' Property: vehicle
    ' Purpose: Accessor for single vehicle playing in simulation
    Public Property vehicle() As Vehicle
        Get
            Return mVehicle
        End Get
        Set(ByVal value As Vehicle)
            mVehicle = value
        End Set
    End Property
    ' Property: mine
    ' Purpose: Accessor for single mine playing in simulation
    Public Property mine() As MinePosition
        Get
            Return mMine
        End Get
        Set(ByVal value As MinePosition)
            mMine = value
        End Set
    End Property
    ' Method: PropertyGrid1_MouseDown
    ' Purpose: Handle event for MouseDown on PropertyGrid1. This will pop-up a
menu allowing
    '       user to choose parameterizations
    Private Sub PropertyGrid1_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles PropertyGrid1.MouseDown
        If e.Button = Windows.Forms.MouseButtons.Right Then
            ContextMenuStrip1.Show()
        End If
    End Sub
    ' Method: Button3_Click
```

```
    ' Purpose: Handle event for button click on Button3. This causes editing
of parameterization list
    Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button3.Click
        mParam.edit()
    End Sub
    ' Method: Button5_Click
    ' Purpose: Handle event for button click of Button5. This causes the
simulation to execute
    Private Sub Button5_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button5.Click
        TextBox2.Clear()
        mParam.incrementAll()
    End Sub
    ' Method: createParameterization
    ' Purpose: This makes calls ot the parameterization layer to cause the
creation of a new iteration
    Private Sub createParameterization(ByVal aType As
Parameterization.IteratorFactory.etIteratorType)
        If PropertyGrid1.SelectedObject Is Nothing Then
            Exit Sub
        End If
        If Not PropertyGrid1.SelectedGridItem.Value Is Nothing Then
            Dim lParam As Parameterization.Iterator

            lParam = mParam.AddTail(aType, PropertyGrid1.SelectedObject,
PropertyGrid1.SelectedGridItem.Label)
            lParam.edit()
        End If
    End Sub
    ' Method: LinearParameterizationToolStripMenuItem_Click
    ' Purpose: Menu event handler. User has chosen to create a linear iterator
for a property
    Private Sub LinearParameterizationToolStripMenuItem_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
LinearParameterizationToolStripMenuItem.Click

createParameterization(Parameterization.IteratorFactory.etIteratorType.eLinear
)
    End Sub
    ' Method: DiscreteParameterizationToolStripMenuItem_Click
    ' Purpose: Menu event handler. User has chosen to create a discrete
iterator for a property
    Private Sub DiscreteParameterizationToolStripMenuItem_Click(ByVal sender
As System.Object, ByVal e As System.EventArgs) Handles
DiscreteParameterizationToolStripMenuItem.Click

createParameterization(Parameterization.IteratorFactory.etIteratorType.eDiscre
te)
    End Sub
    ' Method: Form1_Load
    ' Purpose: Event handler for form loading. Initialize the simulation
```

```
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        initializeSimulation()
    End Sub
    ' Method: initializeSimulation
    ' Purpose: Simulation initialization. Add a single vehicle and mine to the
treeview. This will
    '   allow them to be edited by the propertyGrid control and be
parameterized as well.
    Private Sub initializeSimulation()
        Dim lNode As System.Windows.Forms.TreeNode

        lNode = tvData.Nodes.Add("Vehicle")
        vehicle = New Vehicle
        lNode.Tag = vehicle

        lNode = tvData.Nodes.Add("Mine")
        mine = New MinePosition
        lNode.Tag = mine
    End Sub
    ' Method: tvData_AfterSelect
    ' Purpose: When the user selects a node in the tree view, pass the data
contained to the
    '       propertyGrid control. This will allow the control to edit this
instance.
    Private Sub tvData_AfterSelect(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.TreeViewEventArgs) Handles tvData.AfterSelect
        If Not tvData.SelectedNode Is Nothing Then
            PropertyGrid1.SelectedObject = tvData.SelectedNode.Tag
        End If
    End Sub
    ' Method: Constructor
    ' Purpose: Register as an observer for the parameterizations
    Public Sub New()
        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        mParam.registerObserver(Me)
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: Mine.vb
'* Module Type: Interface
'*
'* Purpose: Provides common interface for Mine logic.
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  AlphaMine, BetaMine
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Interface Mine
    Sub update(ByVal aRecieveLevel As Double)
    Property hasFired() As Boolean
    Sub reset()
End Interface
```

```
'*****************************************************************
'*
'* Module Name: MinePosition.vb
'* Module Type: Class
'*
'* Purpose: Positional information for a mine, along with linking to a mine's
logic
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  AlphaMine, BetaMine
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class MinePosition
    'Member attributes
    Private mXPos As Double 'Positional Data
    Private mYPos As Double
    Private mZPos As Double
    Private mType As String 'String representing mine type
    Private mMine As Mine   'Reference to underlying mine logic
    'Property: Accessors for positonal data
    Public Property x() As Single
        Get
            Return mXPos
        End Get
        Set(ByVal value As Single)
            mXPos = value
        End Set
    End Property
    Public Property y() As Single
        Get
            Return mYPos
        End Get
        Set(ByVal value As Single)
            mYPos = value
        End Set
    End Property
    Public Property z() As Single
        Get
            Return mZPos
        End Get
        Set(ByVal value As Single)
            mZPos = value
        End Set
    End Property
    ' Method: mineType
    ' Purpose: Accessor for mine logic
    ' Implementation Notes: The mine type is set via string. However, this
string is in turn
    '   converted to a class via reflection. This was done to display
parameterization of strings.
```

```
    '   If you perform a discrete parameterization of "AlphaMine", "BetaMine"
on this property then
    '   the parameterization will iterate over both mine types.
    Public Property mineType() As String
        Get
            Return mType
        End Get
        Set(ByVal value As String)
            If mType <> value Then
                mType = value
                ' system.Reflection.Emit.ConstructorBuilder.
                Dim t As Type = System.Type.GetType(Application.ProductName &
"." & mineType, True, True)

                If t Is Nothing Then
                    Err.Raise(513, Me, "Invalid mine type")
                End If

                Dim c As System.Reflection.ConstructorInfo =
t.GetConstructor(New Type() {})
                mine = c.Invoke(Nothing)
            End If
        End Set
    End Property
    ' Property: mine
    ' Purpose: Accessor to underlying mine logic
    Public Property mine() As Mine
        Get
            Return mMine
        End Get
        Private Set(ByVal value As Mine)
            mMine = value
        End Set
    End Property
    Public Sub New()
        mineType = "AlphaMine"
    End Sub
End Class
```

```
'*****************************************************************
'*
'* Module Name: Vehicle.vb
'* Module Type: Class
'*
'* Purpose: Represents a vehicle
'*
'* Supertypes/Interfaces: None
'* Known subtypes:  None
'*
'* Implementation Notes:
'*
'*****************************************************************
Public Class Vehicle
    'Member attributes
    Private mXPos As Double   'Positional data
    Private mYPos As Double
    Private mZPos As Double
    Private mSignature As Double  'Acoustic signature of vehicle (assumed to
be a single dB level rather than freq dependent)
    Public Property x() As Single
        Get
            Return mXPos
        End Get
        Set(ByVal value As Single)
            mXPos = value
        End Set
    End Property
    Public Property y() As Single
        Get
            Return mYPos
        End Get
        Set(ByVal value As Single)
            mYPos = value
        End Set
    End Property
    Public Property z() As Single
        Get
            Return mZPos
        End Get
        Set(ByVal value As Single)
            mZPos = value
        End Set
    End Property
    Public Property signature() As Single
        Get
            Return mSignature
        End Get
        Set(ByVal value As Single)
            mSignature = value
        End Set
    End Property
End Class
```

REFERENCES

[1] These, A., and Travis L. Introduction to Simulation. In *Proceedings of the 1991 Winter Simulation Conference*, 1991.
[2] Biles, W. E., Design of Simulation Experiments. In *Proceedings of the 1984 Winter Simulation Conference*, 1984.
[3] www.nswcdc.navy.mil/mast/dahlgren/Aredept.pps
[4] Mace, R., Howell, L., and Gilman, G. A Composite HWIL/Event Driven Federation for Mine Warfare Analysis. *SISO Simulation Interoperability Workshop (SIW), Fall 1998.* 1998.
[5] Sammelmann, G. Propagation and Scattering in Very Shallow Water. *OCEANS 2001 MTS/IEEE Conference and Exhibition*. Nov 2001.
[6] Sammelmann, G. High-Frequency Images of Proud and Buried 3D Targets. In *Oceans 2003 Proceedings, Volume 1*. 2003.
[7] Eadie, J., and Mace, R. Autonomous Littoral Warfare Systems Evaluator-Engineering Simulation (ALWSE-ES*). OCEANS 2001 MTS/IEEE Conference and Exhibition*. Nov 2001.
[8] Kleijnen, J. Sensitivity Analysis and Optimization in Simulation: Design of Experiments and Case Studies. In *Proceedings of the 1995 Winter Simulation Conference*, 1995.
[9] Roser, C., Nakano, M., et al. Throughput Sensitivity Analysis Using a Single Simulation. In *Proceedings of the 2002 Winter Simulation Conference*, 2002.
[10] Smith, B.C. Reflection and Semantics in a Procedural Programming Language. Phd thesis, MIT, January 1982.
[11] Maes, P. Concepts and Experiments in Computational Reflection. In *OOPSLA '87 Proceedings*, 1987.
[12] Chiba, S., Load-Time Structrual Reflection in Java. In *ECOOP 2000*, pp. 313-336. 2000.
[13] Malenfant, J., Jacques, M., and Demers, F. N. A Tutorial on Behavioral Reflection and its Implementation. http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/ref96/ref96.html . 1998.
[14] Friedman, D. P. and Wand, M. Reification: Reflection without Metaphysics. In *Conference Proceedings of Lisp and Functional Programming*, pp 348-355, ACM, 1984.
[15] Weyrauch, R. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artifical Intelligence*, 13, 1980.
[16] Gamma, E., Helm, R., et al. *Design Patterns Elements of Resuable Object-Oriented Software*. Addison-Wesley. 1995.

BIOGRAPHICAL SKETCH

George Gilman was born in Keene, NH on May 1, 1972. He attended Florida State University, graduating with honors in the summer of 1994 with a Bachelor of Science Degree in Electrical Engineering. In the fall of 1998, he graduated from Florida State University with a Master of Science Degree in Electrical Engineering. Currently he is pursuing a second Master of Science Degree at Florida State University; this degree in Computer Science. He has been employed at Naval Surface Warfare Center, Panama City (NSWCPC) since 1990 where he has investigated and developed a number of simulations related to unmanned underwater vehicles (UUVs), robotic vehicles, and other areas of mine warfare. His main areas of interest are simulation and robotics.