The Advanced Battlefield Architecture for Tactical Information Selection (ABATIS)

J. Sean Keane, Jerzy W. Rozenblit
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721-0104

and

Michael Barnes
ARL Field Unit - Ft. Huachuca
ATTN: AMSRL-HR-MY
Ft. Huachuca, AZ 85613-7000

Abstract

Modern warfare requires the understanding and management of increasingly complex assemblages of resources. The Advanced Battlefield Architecture for Tactical Information Selection (ABATIS) is introduced. It provides a framework for testing various display strategies. Its design, which uses object-oriented and hierarchical design methodologies, is flexible and extensible. It assures that a working program can be rapidly developed for comparing alternate display strategies. This report defines an overall architecture for battlefield visualization and then focuses on a detailed design of its display layer, called the Process Centered Display (PCD). The design is specified using the Object Modeling Technique (OMT) notation. The complete class diagrams for the PCD are presented and an illustrative example is given.

1. Introduction

Despite major changes in the political makeup of the world's nations, armed forces will continue to be necessary for the foreseeable future. While their emphasis has shifted from strategic deterrence to smaller localized conflicts, events such as Operation Desert Storm clearly emphasize the need for large, well-coordinated forces. In a world of NATO forces assembled from many countries, the variety and number of resources important to the battlefield will only increase. New systems must be designed and implemented to help military decision-makers understand the battlefield situation quickly.

There are a number of themes that should permeate any new software architecture for battlefield visualization. Most importantly, the architecture must facilitate understanding of the *process* of the battle, rather than simply the current location of various forces. This requirement implies that the software must somehow "understand" how the user assimilates battlefield state information into a process-centered viewpoint. One aspect of this problem is the assembling of individual units of information into context-rich, higher-level composites. Another is the presentation of this derived information in a way that is intuitive to the human user. The former aspect involves artificial intelligence and knowledge-based design techniques. The latter aspect is the focus of the work shown herein.

In this report, the focal point is the software design aspects of the battlefield visualization architecture. Object-Oriented Design (OOD) is used to achieve flexibility and extensibility. Design patterns are used to guide the software architect in the proper use of common object-oriented structures. They also suggest solutions

This research is sponsored by the Army Research Laboratory in Ft. Huachuca, Arizona under contract W26236-5048-7002.

0-8186-7889-5/97 $10.00 © 1997 IEEE

that truly allow the system to be flexible and extensible, capabilities demanded by the purpose for this architecture.

Based on these principles, the overall Advanced Battlefield Architecture for Tactical Information Selection (ABATIS) is presented. This architecture will be refined and implemented to conduct experiments quantifying the effectiveness of various display strategies.

2. Battlefield Visualization

2.1 Definition and Objectives

Battlefield visualization implies much more than the display of icons on a computer map. The goal is to present the user with an *understanding* of the battle. The processes underlying the battle must be exposed, as well as the past and desired future states of the battlefield. Presenting a display that is closer to the mental vision of the user helps accomplish these goals.

These requirements guide the design of a battlefield visualization system in certain ways. To expose the process underlying the actions of a particular battlefield object, dynamic motion is perhaps the best tool in the designer's tool kit. The movement of objects on the display can correspond to the movement of physical objects on the battlefield. However, motion can be used to quantify variables other than physical displacement. Spinning motions, expansion and contraction, animated substructures, and color changes may all be used to indicate a change in some variable of interest for the object displaying these behaviors.

Increasing the complexity of an object's representation carries certain risks. If the various representations are not orthogonal, they can interfere with one another and produce a confusing display. Even well designed object motion can suffer from being overly cryptic. If the user must make an effort to remember what a particular motion represents or has to look it up in a manual, the motion has failed to provide additional information in an easily accessible form. A mechanism is needed to insure that the user can easily grasp the new wealth of information being displayed.

2.2 Current Systems

Examples of two current systems that share some similarities with ABATIS are JANUS(A) [1] and PCCADS 2000 [2]. JANUS(A) is "an interactive, computer-based, war-gaming simulation of combat operations conducted at the brigade and lower level in the United States Army" [3]. It consists of two opposing forces that are controlled by two players who interact with the system. Developed by the United States Army, JANUS(A) concentrates on ground combat.

One problem with basing a new system on JANUS(A) is immediately apparent: it is composed entirely of algorithms and data written in a structured language. The programs which belong to JANUS(A) consist of approximately 200,000 lines of code written entirely in VAX-11 Fortran, a structured Digital Equipment Corporation (DEC) extension of ANSI standard FORTRAN-77 [4]. This technology seriously impedes any efforts to implement the OOD concepts required by ABATIS.

Another shortcoming of JANUS(A) is its static script files. Motion parameters such as speed and direction of travel are not included in these scripts; only the static location of objects is provided at various points in time. Attempts to parse these files into ones that include motion information have had some success [5], but clearly such a system is inappropriate for the dynamic experiments to be conducted with ABATIS.

Another military system used for visualization is the Air Force's PCCADS 2000 cockpit display system [6]. This system is more similar to ABATIS due to an object-oriented design using PHIGS+ middleware for graphical functions. However, the PCCADS 2000 system specifies much more than a software architecture. The concerns of size, weight, power consumption, and rendering performance are absent in the requirements for ABATIS. PCCADS 2000 is optimized for terrain rendering in three dimensions, a display option that may not even be present in ABATIS. PCCADS 2000 highlights many of the tradeoffs associated with software engineering and provides one possible way to implement such a system, but it is too specialized for reuse in ABATIS.

2.3 Abatis Requirements

Following the dictates of goal-driven design, ABATIS must accomplish a specific set of predefined goals. Put simply, ABATIS must allow the simulation of dynamic battlefield objects in a way that exposes their processes and is intuitive to understand. The ultimate simulation of these objects to evaluate various display strategies is of central importance.

All major elements of the system should be hierarchical, allowing collections of objects to be composed and treated the same as individual objects.

Hierarchical structure is naturally encouraged by OOD through inheritance, and allows a relatively small set of objects to create a rich base of composite objects that are equally adequate as the basis for the rest of the software.

Finally, the system must be extensible. By definition, the comparison of various display strategies requires more than one such strategy to exist. ABATIS must be designed to allow these different strategies to be substituted for one another at runtime. New strategies should be easy to add, and have little or no impact on the rest of the software.

2.4 Design Patterns

Experience with early object-oriented programming led to the design of *architectural frameworks*, i.e., reusable software structures for a particular application domain. Database frameworks, word processing frameworks, and Graphical User Interface frameworks have all been applied with great success. The fundamental problem with frameworks is their limited generality. Their association with a particular domain of interest restricts their use in other areas.

Despite its domain dependency, a framework has to be general enough to be reusable. Designing good frameworks has helped expose recurring patterns of interaction or structure that meet this goal. Abstracting the common mechanisms used to implement frameworks and other complex object-oriented software has led to the idea of design patterns. A commonly accepted pattern definition follows:

> *"A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue." [7]*

3. System Architecture

A software architecture is a definition of a software system in terms of its components and their interactions. The architecture defines the system's structure, topology, and semantics. A good object-oriented architecture provides a correspondence between the objects and requirements of a particular domain and the software that implements them.

3.1 Abatis Architecture

A specific architecture is now proposed for ABATIS. The architecture given is for a complete system, capable of processing raw information and using it to drive the process centered display (PCD). In this use, ABATIS is not a simulator, but an actual battlefield tool to be used by those in command. As is common in software engineering, the architecture is arranged into *levels of abstraction*, and separated into physical and procedural layers, Figure 1.

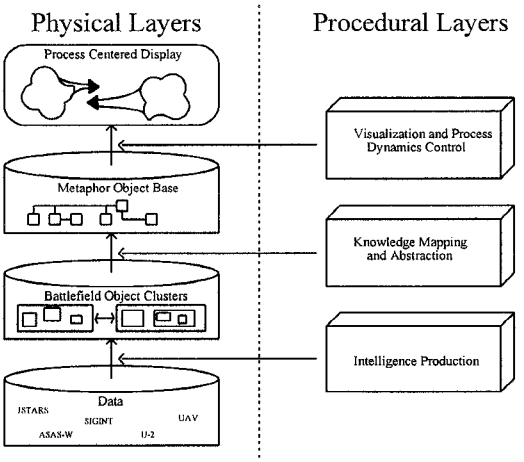

Figure 1 ABATIS High-Level Architecture

The physical layers comprise:

a) Data base: contains intelligence data collected through various sources, e.g., imagery, HUMINT, JSTARS, etc. This is "raw" data.
b) Battlefield Object Clusters: collection of battlefield objects abstracted through the process of intelligence production (please refer to the description of procedural layers below).
c) Metaphor Object Base: metaphors are model engines that embody procedural mechanisms for display of battlefield state.
d) Process Centered Display: the visualization interface with graphical elements created for the purpose of the processes that the metaphors underlie.

The procedural layers of the architecture enable the transitions through the physical levels. Through intelligence production, data can be clustered, categorized, and amalgamated into objects that will

eventually underlie the metaphors. Knowledge abstraction and mapping procedures will facilitate this by providing mechanisms that will associate metaphors with the battlefield object clusters. The Visualization and Process Dynamics Control is a set of procedures and rules governing the change of graphical elements states on the PCD.

The effort described in this paper focuses on the detailed design specifications of the process centered display. Our approach is to prototype the remaining elements of the architecture through simulation. Simulating the model components at first allows the PCD to be developed and tested independently. This produces a battlefield visualization testbed that is useful for conducting experiments concerning which visualization techniques are the most effective.

Such a testbed may be incrementally developed by substituting real-world data for simulated data in stages. The procedural and physical layers are organized as separate objects that communicate by sending commands. The source of those commands can be a simulator, or some other existing military software system adapted to that function.

The clear migration path from the simulator to a complete battlefield system is a good example of *code reuse*, a major goal of object-oriented design (OOD). Code reuse helps insure that the maximum benefit is derived from every line of debugged code.

3.2 Model Base

The three lowest physical layers are the basis for the construction of a model base intended to dynamically control the PCD. The lowest level is the raw data as it is acquired from the battlefield. This data has many different formats, and may be valid for varying times in the past. For example, some data may be current, while other data comes from sources that may be an hour old. Data at this level is relatively unorganized and unstructured.

Through the procedural application of intelligence production, the raw data is clustered or processed in some other way to produce the first level of abstraction. Battlefield object clusters are more closely related to the types of objects that commanders consider when making tactical decisions. If a conventional user interface were applied to this level of the model, a display showing battlefield state but not battlefield processes would result.

The key to ABATIS is the metaphor object base. The goal of this highest level of the model is to capture the process of the battle. The battlefield objects are used to create metaphor objects. If these objects accurately reflect the thought processes of the system's users, they encourage a deeper understanding of the battle that should result in better performance when predicting future events.

As previously suggested, the initial work on ABATIS involves constructing a simulator metaphor object base. This metaphor object base should present the same interface to the Process Centered Display (PCD) as the final, working version. In other words, they are of the same *type*. The simulator model implements an algorithm for updating metaphor objects based on a simulation scenario, while the working version does so based on the knowledge mapping and abstraction process.

3.3 Process Centered Display

As implied by the last letter in ABATIS, the process centered display is highly concerned with the *selection* of tactical information display strategies. Multiple views of a particular situation are possible within this system. The creation of metaphors, their animation, and the task of updating them to reflect changes in the model (or actions by the user) are the responsibility of the PCD.

Some of these responsibilities are common to many GUI designs. Providing concrete software solutions for the animation of metaphor depictions is an important design task that is presented later in this report.

4. ABATIS-PCD Architecture

The Process Centered Display (PCD) must display the battlefield so that in addition to the current state of the battle, the processes by which battlefield objects evolve are also made apparent. Understanding how a display can meet these requirements leads to an object-oriented software architecture that may be used in a full implementation of ABATIS, or in a battlefield visualization emulator.

The PCD is developed using goal-driven design. An optimum design will result by focusing on the project goals and allowing them to define which methods and tools to employ. This is in contrast to first constraining the design by implementation language and then seeing if an acceptable design is still possible.

4.1 PCD Goals

The main goal of the PCD is to convey the *processes* that are occurring on the battlefield. Since battlefield processes evolve and change as the battle unfolds, the

software architecture must also support dynamic change and evolution at runtime. Given the vast range of possible battlefield scenarios and objects, the architecture must also be flexible enough to allow the quick creation of new library objects from old ones.

A secondary goal is to focus on the possibility of using motion, color changes, or other types of animation to convey information. Some uses of animation are obvious, such as moving a symbol upward on the screen when an actual battalion moves North. However, abstract quantities can also be tied to motion. A simple example would be allowing the strength of a ground force to be represented by the speed of rotation of its symbol. When done in a way that matches the intuitive notions of the user, such a presentation of information becomes a *metaphor*. The metaphor correlates familiar experiences with the actions of symbols on the computer display.

A final goal is to allow arbitrary levels of complexity in both the battlefield objects and their associated process dynamics. This complexity is needed to accurately model the intricate dynamics of a real battlefield and its metaphorical representation.

Mutually compatible solutions for reaching these goals exist and can be incorporated into a single architecture. The architecture for the ABATIS Process Centered Display, or ABATIS-PCD, is presented in later sections.

4.2 ABATIS-PCD Requirements

The software architecture for the ABATIS-PCD has to incorporate the ability to display complex, evolutionary processes as well as simple, repetitive changes. Every graphical element has some sort of *behavior* associated with it. Here, a *behavior* is anything that can cause a change in how an element is displayed. If a graphical element changes its color, then some behavior must have initiated that color change. Similarly, an element that is moving in a straight line has a behavior for moving in straight lines associated with it. A graphical element without any behaviors may be visible, but it will be static in appearance until some behavior is initiated.

Considering possible battlefield displays, it becomes quickly apparent that groups of graphical elements with a common behavior may be desirable. Thus, it is insufficient to simply associate behaviors with graphical elements; a more abstract construct is needed. This construct is named an Actor. As with their metaphorical counterparts, Actors in ABATIS-PCD come in a variety of "skill levels". Some Actors may be completely static, while others exhibit behaviors so subtle and evolutionary that they suggest information to the user in novel ways.

The algorithms associated with the data vary widely in complexity. When viewed from the object-oriented perspective of Actors, however, the basic software constructs are more similar than different. The problem domain encourages thinking of the various graphical components as largely autonomous but occasionally interacting. This suggests an object-oriented approach.

4.3 Process Centered Display Design

The process centered display should have a single object that interfaces the PCD with the external software components. This allows the same interface to be presented to a directly implemented simulation scenario, or the knowledge synthesis engine of a full-blown ABATIS implementation. Since this object coordinates the activities of Actors, it is called a Director.

Actors are objects that combine the ability to change with some means of visual representation. The objects that cause Actors to change are Behaviors. They are abstracted into their own objects to promote flexibility. Rather than writing new code to implement a new Behavior into every Actor that might need it, separate Behavior objects can be attached to any Actor dynamically. For similar reasons, the visual representation of an Actor is also abstracted into its own object. These objects are names Grels, a contraction of "graphical elements". Dynamically altering an Actor's Behaviors and Grels allows the process represented by the Actor to change easily.

The essential components of the process centered display are shown in Figure 2. Another object, the Viewport, is also shown. This is a class of objects present in most graphical user interfaces, and provides a means of separating the Grels from the direct methods used to display them on a particular system.

The design of the rest of the process centered display is motivated by an emerging conceptual tool known as *Design Patterns* [8]. Object-oriented design has matured to the point where a survey of successful object-oriented programs reveals certain recurring architectural similarities. Analyzing these similarities and abstracting them into Design Patterns creates a catalog of software constructs. Selecting the correct Design Patterns from the catalog to use in a particular piece of software is still a matter of judgement and skill for the software architect. If that selection is done well, however, the patterns provide a well-documented framework from which to

develop the architecture.

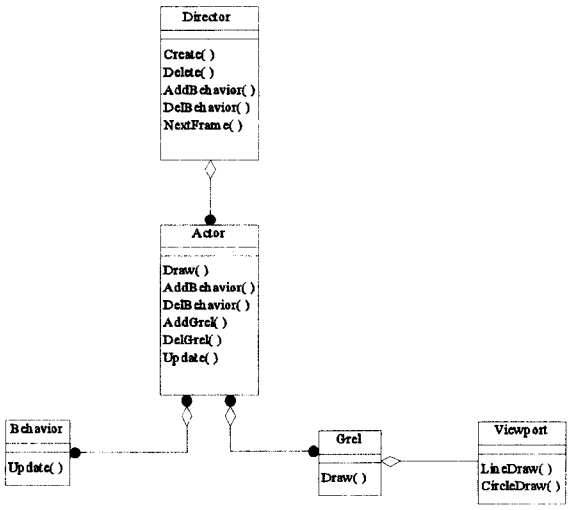

Figure 2 ABATIS-PCD Essential Components Class Diagram

The first design pattern to be used in ABATIS-PCD is named Composite, and will be applied to Actors. As stated in the section on PCD Goals, the Actors need to support hierarchical construction. Said another way, the software interface for an individual Actor should be the same as for a group of Actors. This technique is often seen in CAD software, where individual CAD elements may be grouped together and manipulated in concert using the same commands as are used for individual elements. The Composite pattern provides the concrete mechanism for implementing the idea of constructing objects hierarchically. Unlike the inheritance hierarchy which is fixed at compile time, the composition hierarchy may be altered or extended during the program's execution. This allows the potential for Actors to be dynamically modified by the Director, as well as by one another.

Composition is necessary in ABATIS-PCD because it ultimately allows the creation of metaphors for battlefield processes. Simple Actors have simple Behaviors and Grels, which are inadequate for the purposes of battlefield visualization. Allowing Actors to be combined into more complex aggregates provides a structure for making the processes and representations for Actors to become more complex as well. If designed well, the Composite Actors become metaphors for other systems in the mind of the user. They can assist in the comprehension of the current battlefield state and the likelihood of specific future states.

An abstract representation of the Composite pattern is shown in Figure 3 [9]. The class Component, from which the other classes are inherited, defines the interface for this type of object. The Leaf class implements the basic operation of this object. These are the lowest-level objects of this type, the ones that are desirable to build into composite structures. Finally, the Composite class allows Leaf and other Composite objects to be combined, creating the next level of the hierarchy.

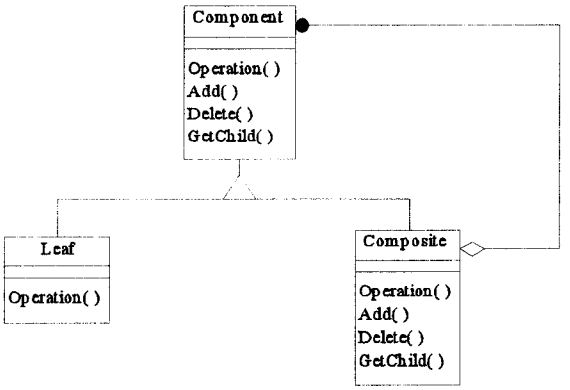

Figure 3 Composite Design Pattern Class Diagram

In the case of ABATIS-PCD, the Component objects correspond to Actors. Derived from this class are the classes ActorLeaf and ActorComposite. As more ActorLeaf objects are defined, a library of them is built which may be used to carry out various simulations. Similarly, a library of ActorComposites results from building the ActorLeafs into more complicated structures. The implementation of the Composite pattern to ABATIS-PCD Actors is shown in Figure 4.

Vars, shown in the attributes section of the Actor, is a data structure that contains information about the actor that might affect the way it is displayed on the screen. Examples of the kinds of variables in this data structure are the actor's position, color, size, and orientation. This same data structure will be used to hold information about Grels, as will be seen later. Therefore, the information it contains must be compatible with both object types. Actors also store their class name and individual object name. Methods for accessing this information are not shown in the diagram, but can be easily imagined and should be implemented in many of

the classes of ABATIS-PCD. This allows the director to find and deliver objects to specific Actors, and to delete specific Behaviors and Grels.

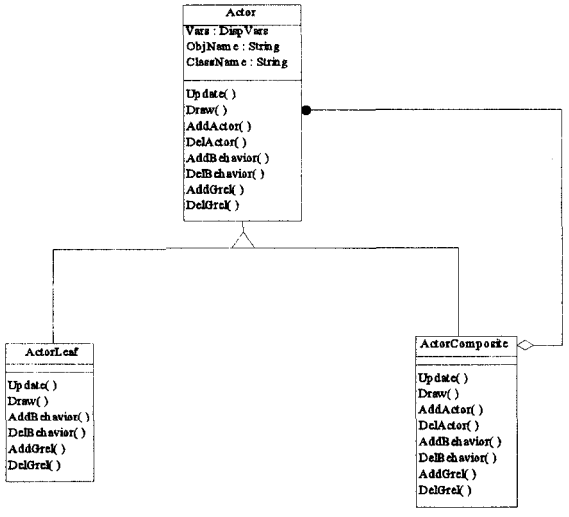

Figure 4 Composite Actor Class Diagram

The ActorLeaf class does not support the methods AddActor and DelActor. These methods are used by the ActorComposite to maintain its list of Actors. By definition, ActorLeaf objects are atomic and do not have such a list.

It must now be determined whether or not any other objects in the ABATIS-PCD software architecture should also take advantage of the Composite pattern. Since Behaviors and Grels are attached to Actors, and Actors may be hierarchically composed, this mechanism alone could be used to build up hierarchical Behaviors and Grels. However, there is another reason to consider using this pattern. Utilizing the Composite structure can make libraries of objects easier to create and maintain. While a complex tree of behaviors may act no differently than a simple list of them, the lists are often more cumbersome for the programmer when creating simulations. Therefore, a similar structure is proposed for the Behaviors and Grels. Figure 5 is a class diagram for Behaviors, and Figure 6 is one for Grels.

Finally, it has been mentioned previously that the Director maintains a list of Actors, and each Actor maintains a list of Behaviors and another list of Grels. While a list data structure may indeed be used to store information about a collection of objects, the architecture should not be constrained to a particular data structure. It may also be convenient to traverse the collection of Behaviors or Actors or Grels in more than one way. Traversing the list in ascending versus descending alphabetical order is an example of two different ways of accessing the collection. Rather than hard-coding the means for accessing members of a collection, this idea may be abstracted into separate objects. Such an object, which provides a common interface for traversing groups of other objects, is called an *Iterator*.

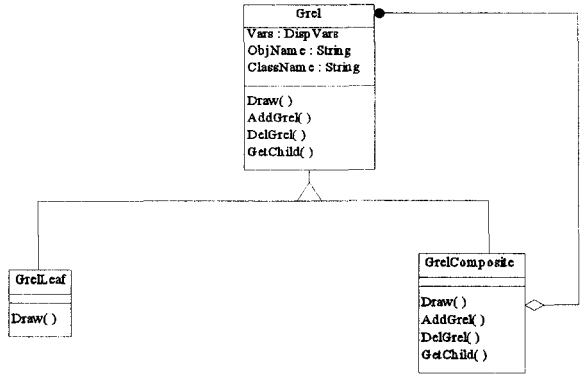

Figure 5 Composite Grel Class Diagram

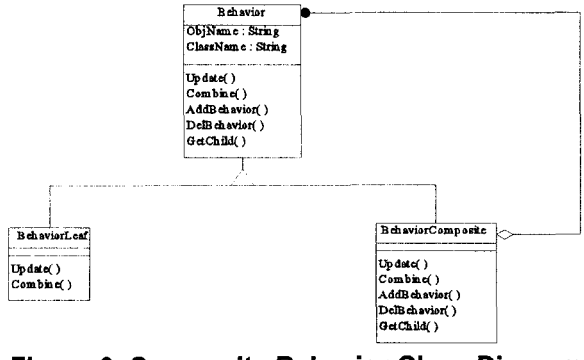

Figure 6 Composite Behavior Class Diagram

Iterators have the class relationships shown in Figure 7 [10]. Other program objects use only the *abstract* classes Aggregate and Iterator. These are abstract because no objects of this type are ever actually created; only the derived classes are instantiated. Once again, inheritance is being used to specify an interface in the parent classes which is used by all of the derived child classes. Here, ConcreteAggregate inherits from Aggregate, while ConcreteIterator inherits from Iterator. When the ConcreteAggregate creates the ConcreteIterator at the request of some other client

object, the ConcreteAggregate passes the iterator a reference to itself so that the iterator has an aggregate with which to work.

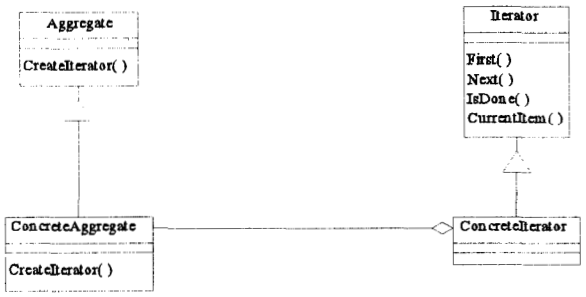

Figure 7 Iterator Design Pattern Class Diagram

This Design Pattern is applied to all of the classes that support group structures, namely Actors, Behaviors, and Grels. Should one choice for the data structure used to hold a group (such as a linked list or a hash table) demonstrate weaknesses in access speed or some other performance variable, it may be easily changed by defining new concrete classes for aggregates and iterators. Since all of the information concerning which type of data structure is used has been encapsulated into new objects, changing that data structure will have no effect on the rest of the system.

One final new class is needed. A Grel specifies the shape of a graphical element, such as a square. However, depending on the capabilities of the Viewport, a square may have different representations. An example is a 3D view as opposed to a 2D view. A new class, placed between the Grel and the Viewport, can change how the Grel should be "painted" onto the Viewport. Due to this function, the new class is called an Artist. Its primary function is to make old Grels work with new Viewports, although they could also be changed during execution if desired.

Figure 8 shows the main class diagram for the ABATIS-PCD. Along with the other diagrams presented, it completely specifies the class relationships in the PCD. The Director has an ActorGroup. The members of this group are accessed by requesting an iterator (in this case, an AGIterator) and using it to get each Actor. The Actors each have a BehaviorGroup and a GrelGroup to store the two important items of information about an Actor: how it acts, and how it looks. These groups each have their own iterators to give transparent access to the members of the group. Behaviors update the state of an Actor or its Grels, while Grels draw themselves on Viewports with the help of Artists.

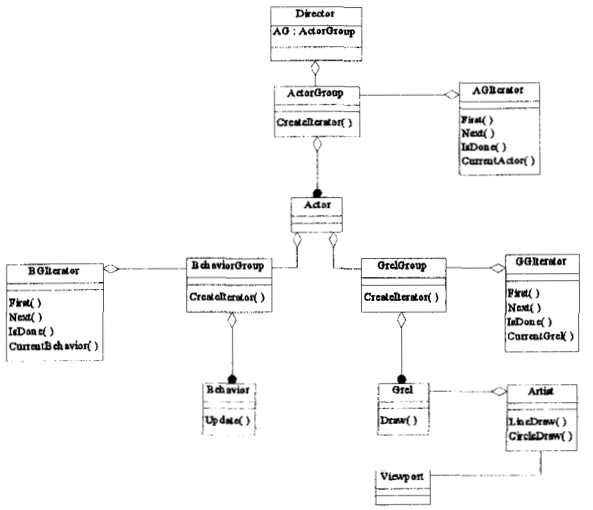

Figure 8 Main PCD Class Diagram

4.4 Example

A simple example will help illustrate the mechanisms by which Actors are created, animated, and controlled. Extensions to this example are then suggested for implementation in a prototype system. However, the ultimate goal of the ABATIS-PCD will be to accommodate virtually any idea for Actor behavior and appearance. It is envisioned that with a basic catalog of models, simple simulations will be quickly implemented and evaluated. This process generates new ideas for display dynamics, including ones that may not have been envisioned when the system architecture was developed. Only a flexible system will adapt gracefully to these new requirements. It is at this level of development that the design methods used in ABATIS-PCD will be most beneficial.

Consider the representation for a battalion in ABATIS-PCD. While the goal of the system is to help develop innovative display mechanisms, it can also easily accommodate current symbols. Figure 9 shows an Actor for one particular type of battalion and its corresponding representation.

The battalion has no behaviors associated with it as shown. There are three Grels in the GrelGroup. One

draws the rectangular box, another draws the diagonals of the box in the shape of an "X", and the third makes two marks on the top of the box. If desired, these Grels could be given names that correspond more closely to their symbolic meanings.

Assume that the Director is told to create this actor. The object is instantiated, and the Director tells the Actor to draw itself. At this point, the default display color might be white. The Actor passes along the draw request to the Grels, which use their Artists and Viewports (not shown in Figure 9) to paint pixels on the screen.

Now assume that it is determined that the battalion is friendly to the "blue" forces and not the "red". This information causes a message to be sent to the Director saying that the battalion should be permanently changed to the color blue. This behavior, called TurnBlue, is directed (hence the Director's name) to the proper battalion, where it is attached to the BehaviorGroup. At the next
screen update time, the Director tells all Actors to update their behaviors. The battalion checks its group of behaviors and sees that it now has one, so the behavior is updated by passing it the Actor. Each Behavior decides for itself internally whether it acts upon the characteristics of the Actor or the Grels. In this case, the TurnBlue Behavior changes the color of all Grels to blue.

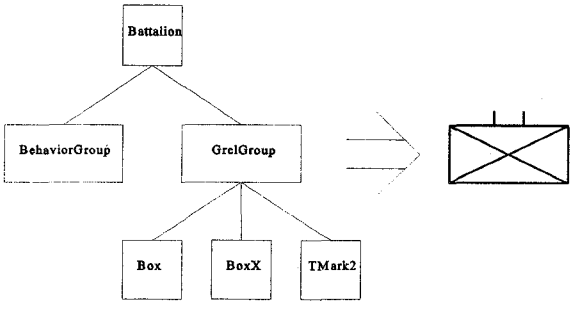

Figure 9 Battalion Actor

Upon completion, the TurnBlue object returns control to the Actor. However, this particular behavior is designed to only be executed once, so the return value tells the Actor (the battalion in this case) to delete the TurnBlue behavior. However, the next time the Director asks for all Actors to display themselves, this battalion will change from white to blue.

A more enduring behavior is one that causes the Actor to move to a new location in a specified amount of time. This behavior is created and sent to the battalion through the Director in much the same way as the TurnBlue behavior. Based on the parameters given it when created, the MoveTo behavior calculates how much the Actor should move each display frame and makes the required changes to the Actor's location. Each time the display is updated, the Actor (and its Grels, which base their locations on the Actor's) move in a straight line towards the destination. At the completion of the move, the MoveTo behavior also destroys itself. A more complicated behavior, MoveToBSpline, could be given parameters that accomplish the move along more complicated curved paths than a straight line. Since these behaviors do not care (or even know) whether the Actor they are moving is a single "leaf" object or a complex Composite of many other Actors, the same behavior can be used to animate a simple box on the screen as well as a complex assemblage of graphical elements.

Finally, consider the behavior Rotate. This behavior modifies the orientation of an Actor to successive positions to make it spin on the screen. The rate of spin may indicate some parameter of interest to the user. This behavior never destroys itself, although it could be removed by sending the appropriate request to the Director.

This simple example, while actually useful and certainly part of any prototype implementation, shows how varied the actions of Behaviors can be. From one-time changes to temporary Behaviors to ones that last as long as the Actor does, the important point is to notice and utilize the flexibility of this system. None of these examples use hierarchical construction, a technique that will unlock the true power of ABATIS-PCD.

5. Future Directions

A prototype implementation of the ABATIS-PCD has been written in C++. This prototype incorporates all of the ideas of the design presented in this report. The simulation scenario is directly implemented at present. Different views of the same simulation can be displayed in multiple windows, and two simulations can be run side-by-side for comparison.

The next stage in the development of this prototype will allow new variations of simulations to be created without compiling any code. Pop-up menus can be created to add Actors, Behaviors, and Grels using an entirely graphical interface. Once created, the simulation would be saved to disk and retrieved later when running cognitive experiments. It may also be possible to process scripts from other systems, such as JANUS, into scenarios that are readable by ABATIS.

ABATIS provides an opportunity to rapidly develop a battlefield visualization simulator that may ultimately be reused in a display system. Its reliance on design patterns guides the future stages of development, and promotes reliable and extensible operation. Modern techniques are needed to meet the challenge of displaying the modern battlefield. ABATIS address those needs, and will accommodate new requirements that may become evident in the future.

References

[1] U.S. Army TRADOC Analysis Command, WSMR, *JANUS(A) Version 2.0 Information Letter*, March 1991.

[2] Pratt, David R., et al, "NPSNET: JANUS-3D Providing Three-Dimensional Displays for a Two-Dimensional Combat Model", *Fourth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, IEEE, September 1993, pp 31-37.

[3] U.S. Army TRADOC Analysis Command, WSMR, *JANUS(T) Documentation Manual*. June 1986.

[4] U.S. Army TRADOC Analysis Command, WSMR, *JANUS(A) Version 2.0 Information Letter*, March 1991.

[5] Pratt, David R., et al, "NPSNET: JANUS-3D Providing Three-Dimensional Displays for a Two-Dimensional Combat Model", *Fourth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, IEEE, September 1993, pp 31-37.

[6] Hancock, William R., et al, "Meeting the Graphical Needs of the Electronic Battlefield", *13th Digital Avionics Systems Conference*, AIAA/IEEE, 1994, pp 465-470.

[7] Gamma, Erich, et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, pp 3-4.

[8] Ibid.

[9] Ibid., pp 163-173.

[10] Ibid., pp 257-271.