Object-oriented modular architecture for ground combat simulation[*]

V. Berzins, M. Shing, Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
{berzins, mantak, luqi}@cs.nps.navy.mil

M. Saluto
EECS Department
United States Military Academy
West Point, NY 10996
dm5447@exmail.usma.army.mil

J. Williams
JSIMS Joint Program Office
12249 Science Dr., Suite 260
Orlando, Fl 32826
julian_williams@jsims.mil

Abstract

This paper addresses the need to modernize the software of the US Army Janus(A) combat simulation system into a maintainable and evolvable structure. It describes the effective use of computer-aided prototyping techniques for re-engineering the legacy software and presents the resultant object models and modular architecture for the existing Janus(A) system. The object models produced in this project have proven invaluable to the contractors during code implementation phase of the US Army TRAC HLA Warrior project and beneficial to other simulation developers.

1. Introduction

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. Legacy systems embody substantial institutional knowledge, which include basic and refined requirements, design decisions, and invaluable advice and suggestions from domain users that have been implemented over the years. To effectively use these assets, it is important to employ a systematic strategy for continued evolution of the current system to meet the ever-changing mission, technology and user needs. However, knowledge embedded in these systems is difficult to recover after many years of operation, evolution, and personnel change. These software systems were originally written twenty or more years ago using what many now view as an archaic and ad-hoc methodology. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that now must be changed over large areas of the code. Re-

[*] This research was supported in part by the U.S. Army Research Office under contract # 35037-MA and in part by the U. S. Army Training and Doctrine Analysis Command.

engineering has frequently been proven to be more cost effective than new development and is also known to better promote continuous software evolution.

Software re-engineering can be defined as the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer. Such improvements often take the form of increased or enhanced functionality, better maintainability, configurability, reusability, and/or other software engineering goals. This process involves recovering existing software artifacts from the system and then re-organizing them as a basis for future evolution of the system. The re-engineering of procedural legacy software into modern object-oriented architectures introduces certain complexities into the software analysis process. Since typical legacy systems were not originally designed and implemented using an object-oriented approach, the products of reverse engineering, such as requirements or design specifications, will probably reflect a functionally based approach. As a result, some form of "transformation" of resultant information is necessary in order to use the specifications. Once a realizable specification based on the transformed object-oriented models is obtained, it is often very difficult to quickly determine if the specification is a true representation of the desired requirements. Since legacy systems are usually re-engineered only when the existing systems need some kind of improvement, it is unlikely that the initial version of the reconstructed requirements adequately reflects current user needs. Prototyping provides a means to validate new system requirements while simultaneously enabling prospective users to get a brief feel for aspects of the proposed system. It is a well-established approach that can be highly effective in increasing software quality [13]. When used in conjunction with conducting a major re-engineering effort, prototyping can be extremely useful in assisting in many areas of software modification, validation, risk reduction, and the refinement of user requirements.

This paper addresses the need to modernize the software of the Janus(A) systems into a maintainable and evolvable structure. It describes the effective use of computer-aided prototyping techniques for re-engineering the legacy software [14] to develop an object-oriented modular architecture for the Janus combat simulation system [16]. Janus(A) is a software-based war game that simulates ground battles between up to six adversaries [7]. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of a large number of FORTRAN modules (1918 FORTRAN routines, 115 C routines, and a total of 393,000 lines of source code). The FORTRAN modules are organized as a flat structure and interconnected with one another via 129 FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create a base-line object-oriented architecture that supports existing and required enhancements to Janus functionality.

2. **Reverse Engineering**

The re-architecturing process adapted in the project consists of 3 major phases: reverse engineering, object-oriented design and design validation via prototyping (Figure 1).

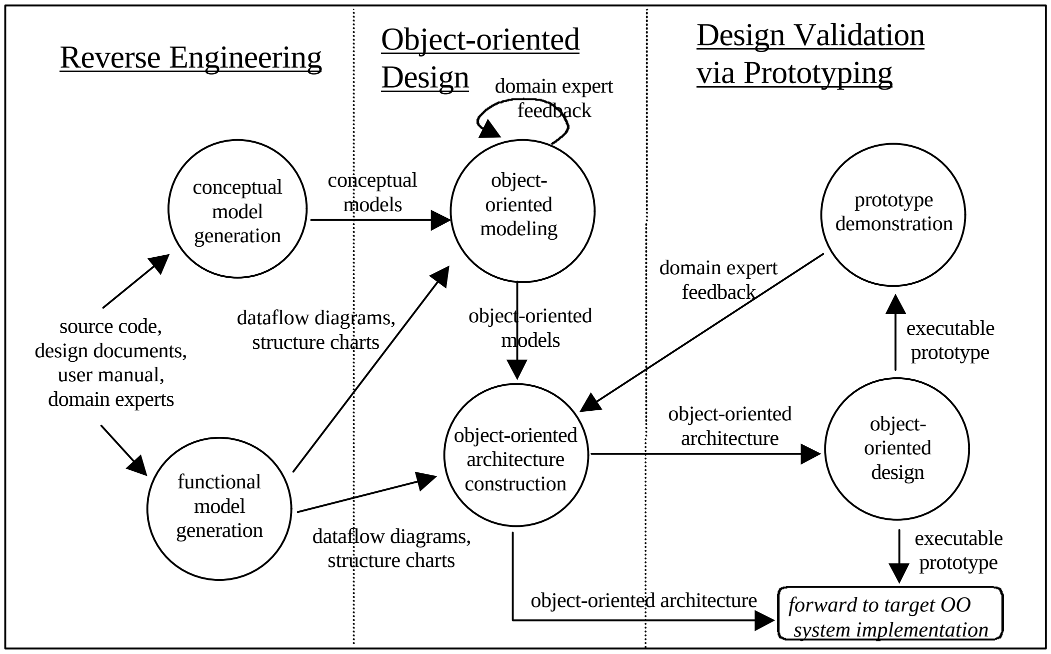
Reverse Engineering
Object-oriented Design
Design Validation via Prototyping
domain expert feedback
conceptual model generation
conceptual models
object-oriented modeling
prototype demonstration
domain expert feedback
source code, design documents, user manual, domain experts
dataflow diagrams, structure charts
object-oriented models
executable prototype
object-oriented architecture construction
object-oriented architecture
object-oriented design
functional model generation
dataflow diagrams, structure charts
executable prototype
object-oriented architecture
forward to target OO system implementation

Figure 1. The object-oriented re-architecturing process

The first phase is reverse engineering. Input to this phase includes the legacy source code, design documents, user manuals, and information from domain experts. Since the goal of the re-engineering effort is to duplicate the functionality of the existing system within a modular, extensible architecture and to reuse the domain concepts, models and algorithms rather than the existing code, we should avoid including any requirements/constraints that are consequences of the FORTRAN implementation. The best places to extract domain concepts from the existing system are the user manuals and the database management system manuals. These manuals were written using the lingo of the user community and should be relatively free of implementation details. We found the JANUS Data Base Management Program Manual [8] particularly useful because it contains detailed information on what kind of data are needed to model the battle field and how they are organized (logically) in the database. The top-level structure of the database is shown in Figure 2.

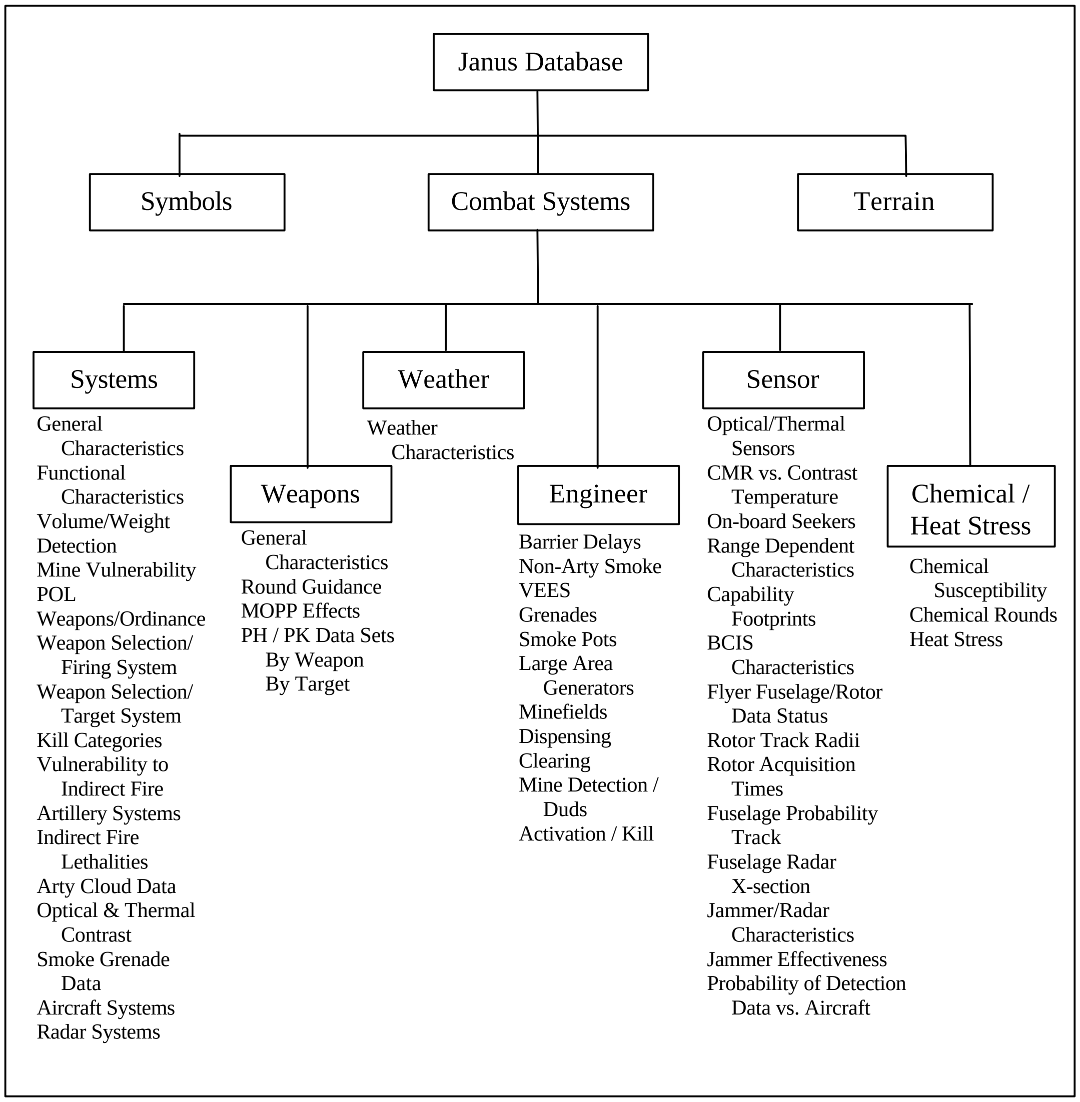

Figure 2. The top-level structure of the Janus Database

Not shown in Figure 2 are the interdependencies between the data, where data entered in one category affect directly or indirectly the data in other categories. For example, the barrier delays of the Engineer Data depend on specific weather condition specified by the Weather Data and system functional characteristics of the System Data of the database. The overall network of interdependencies is highly complex and can only be understood through construction and analysis of the functional model of the existing Janus software.

Analysis of 393K lines of legacy code is a daunting but inescapable part of the process. We recoiled from the magnitude of this effort in the beginning of the project and relied on information contained in the Janus manuals. In hindsight, it was a mistake that slipped the schedule of the project by several months. While these documents helped us get started because they contained higher level information and were much shorter than the code, they were much older and contained outdated information. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross-fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively.

Using manual techniques augmented with simple UNIX shell commands, we were able to walk through the code and get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [6] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs and develop functional models from the data flows. We used the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School, to assist in developing the abstract models [12]. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

We also had a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. These meetings were indispensable because they gave us information that was not present in the code. Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [1]. Domain analysis has been identified as an effective technique for software re-engineering [15]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

3. **Object-Oriented Design**

Next, we developed object models and architecture of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [4]. This was probably the most difficult and most important phase. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this phase, we used our knowledge of object-oriented analysis and applied the OMT techniques [17] and the UML notations to create the classes and associated attributes and operations [18]. This was a crucial phase because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software.

Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [5]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

The re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core data elements and the object-oriented architecture for the Janus System. We presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center project. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced in [9] that the involvement of domain experts is critical for nontrivial re-engineering tasks.

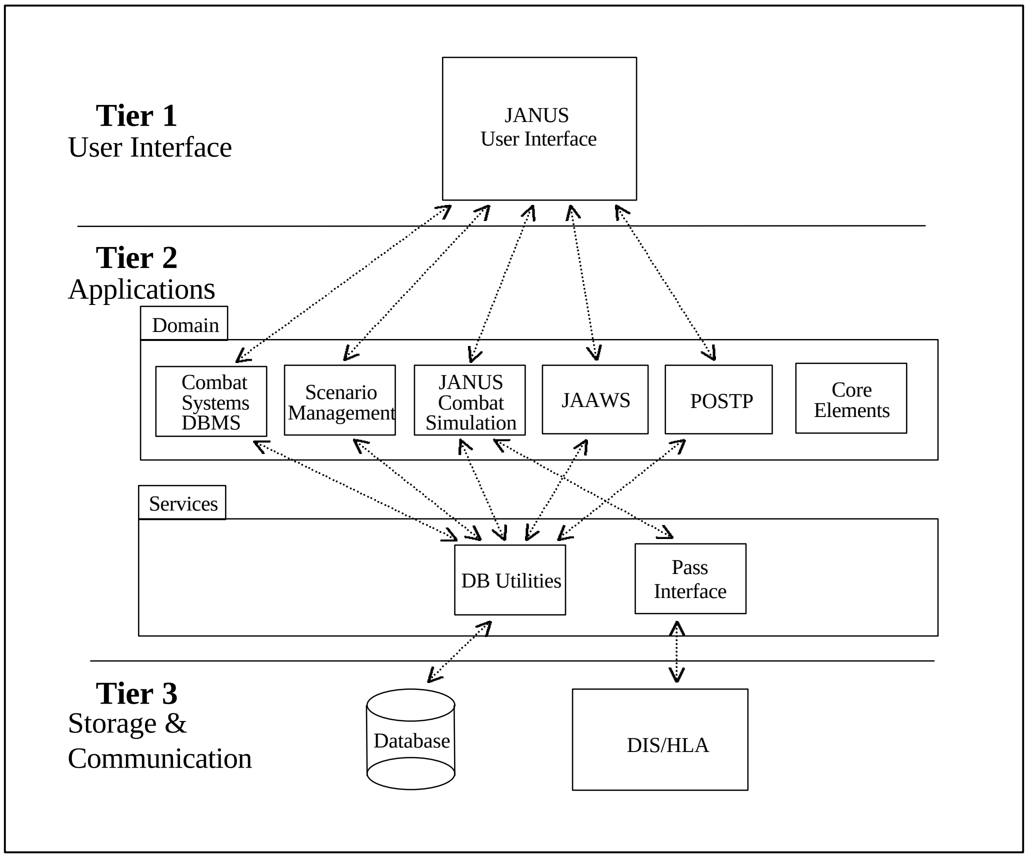

Figure 3. The proposed 3-tier object-oriented architecture

Early involvement of the stakeholders in the simulation community also pays off in the long run. Both the National Simulation Center and Combat21 projects were able to save time and money by reusing our work and came up with designs that look remarkably like ours (although much larger). Now, OneSAF developers have been directed to look at the Combat21 class design and reuse as much as possible. So, our efforts have directly benefited other simulation developers.

Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 3). We extracted most of the data and operations from the existing Combat System DBMS, Scenario Management, Janus Combat Simulation, JAAWS and POSTP subsystems and encapsulated them as simulation objects in the Core Elements package, leaving only application specific control codes that use the simulation objects in each of these five subsystems. Figures 4 and 5 show the top level class structures of the object models of the core elements. Details of the associated attributes and operations can be found in [2, 20] and are omitted from these diagrams due to space limitations.

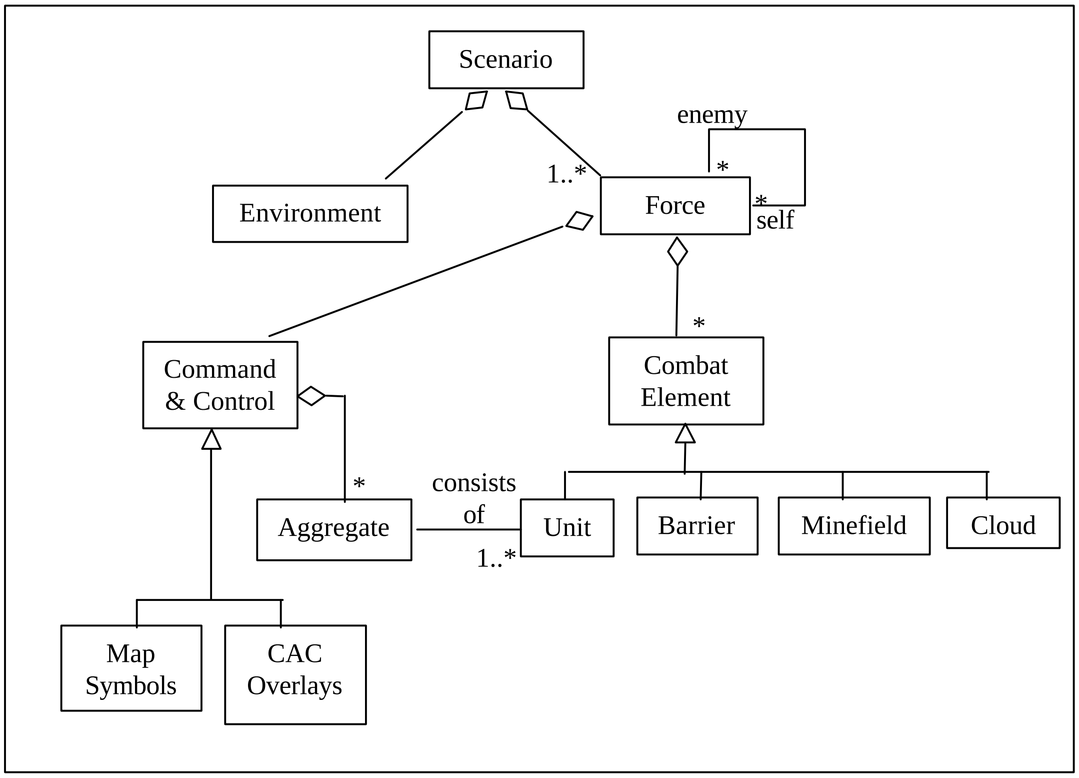

Figure 4. The top-level structure of the Janus Core Elements Object Model

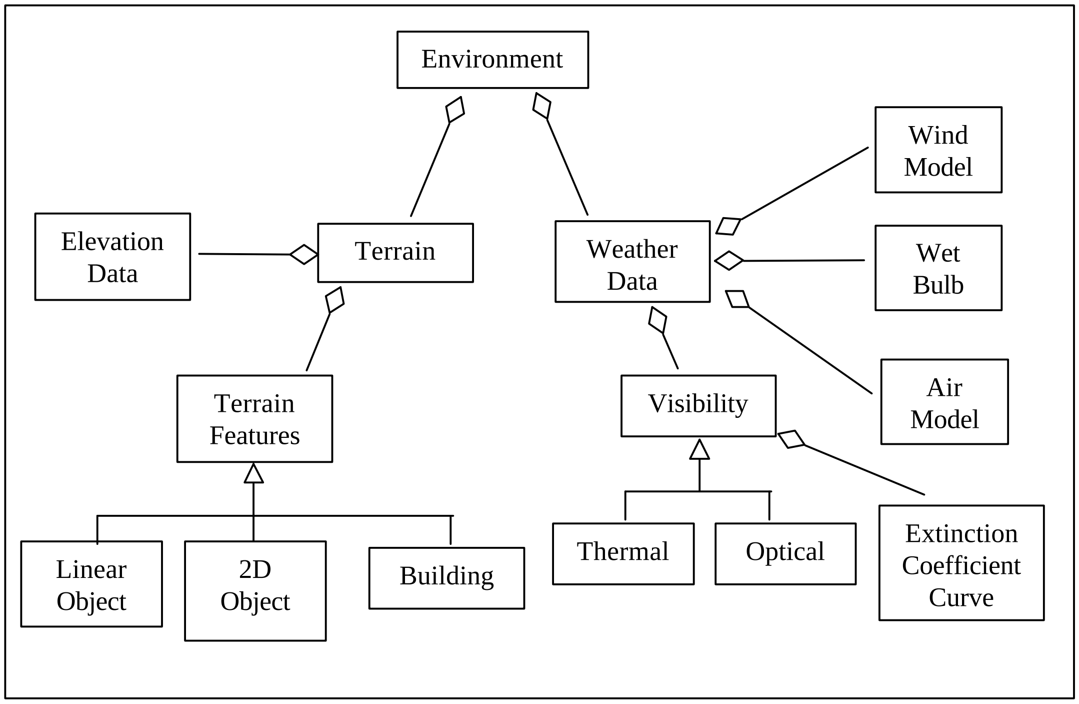

Figure 5. The Environment Object Class

Central to the Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advance the game clock to the scheduled time of the event and perform that event. The existing Janus Simulation System uses 17 different categories to characterize the events. RUNJAN then handles these 17 events using the following event handlers:

1) DoPlan - Interactive Command and Control activities
2) Movement - Update unit positions
3) DoCloud - Create and update smoke and dust clouds
4) StateWt - Periodic activity to write unit status to disk
5) Reload - Plan and execute the direct fire events
6) Intact - Update the graphics displays
7) CntrBat - Detect artillery fire
8) Search - Update target acquisitions, choose weapons against potential targets, and schedule potential direct fire events
9) DoChem - Create chemical clouds and transition units to different chemical states
10) Firing - Evaluate direct fire round impacting and execute an indirect fire mission
11) Impact - Evaluate and update the results of an indirect round impacting
12) Radar - Update an air defense radar state and schedule a direct fire event for “normal” radar
13) Copter - Update a helicopter states

14) DoArty - Schedule an indirect fire mission
15) DoHeat - Update units' heat status
16) DoCkpt - Activity to perform automatic checkpoints
17) EndJan - Housekeeping activity to end the simulation

The legacy event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation Subsystem was to distribute the event handling functions to individual objects. However, many of the current event handler categories contained redundant code and did not seem to be very coherent with respect to the class hierarchy we created. For example, the set of event handlers used to simulate the activities of a particular unit to search for targets, select weapons, prepare for a direct fire engagement, and then execute that direct fire engagement differs depending upon whether the unit has a normal radar, special radar, or no radar at all. The legacy Janus Simulation System uses the Radar event handler to carry out the entire procedure if the unit has normal radar. However, it uses the Search, Radar, and Reload event handlers to carry out the procedure if the unit has special radar. Finally the system uses the Search and Reload event handlers to conduct the procedure if the unit has no radar at all. We conjecture that this lack of uniformity is due to a series of software modifications made by different people at different times without full knowledge of the software structure.

It was necessary to redefine some event categories in order to reduce interdependencies between the event handlers, to factor simulation behavior into more coherent modules, to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Moreover, the Janus system was originally designed to work in isolation, and has since been adapted to interact with other simulation systems. Interactions between the simulation engine and the world modeler (the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation.

The new architecture uses an explicit priority queue of event objects to schedule the simulation events. We were able to reduce the total number of event handlers needed in the simulation, from 17 to 14, by eliminating identified redundant code (Figure 6). The 14 remaining event handlers are as follows:

1) DoPlan - Interactive Command and Control activities
2) MoveUpdateObj – Moves and update the objects in the simulation
3) Search – Searches for potential targets based on the detection devices available to the objects
4) ChooseDirectFireTargets – Once search is complete chooses best target to engage. In future simulations, implementations may allow users to choose targets
5) CounterBattery – Simulates counter battery radar to find potential targets
6) DoDirectFire – Executes direct fire events and updates ammunition status
7) DoIndirectFire – Executes indirect fire events and updates ammunition status
8) ImpactEffects – Calculates results of round impacting

9) UpdateHeatStatus – Updates units' heat status
10) UpdateChemicalStatus – Update unit's chemical status
11) Display – Updates the graphics display
12) WriteStatus – Periodic activity to write units status to disk
13) CheckPoint – Activity to perform automatic checkpoints
14) EndSimulation – Activity to end the simulation

We tried to make the actions of the new event handlers independent and orthogonal. Independent means that one event handler does not invoke or depend on the action of another. Orthogonal means that the purpose of one event handler is completely separate from that of another. Although our architecture does not completely meet these goals, it comes much closer to them than the legacy design does. We believe that these properties of the architecture are desirable because they impose a partitioned structure on the system that aids future enhancements and modifications. If an enhancement affects only one kind of event, then it becomes relatively easy to isolate the affected part of the code. If suitable naming conventions were followed, relatively low-tech tool support would be adequate for helping system maintainers find the parts of the code that must be understood and modified to make a future change to the system.

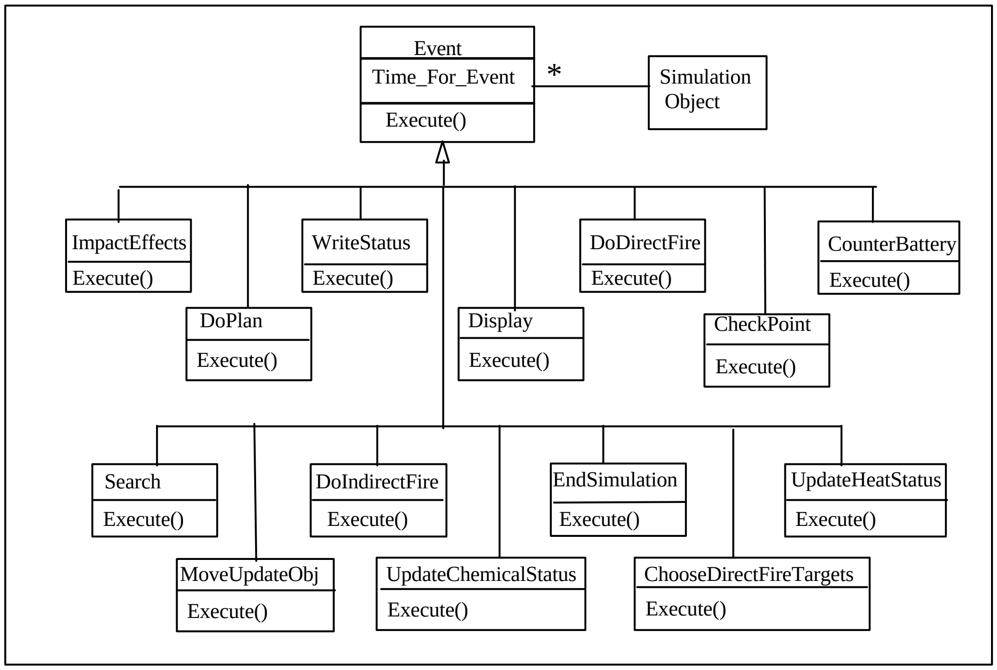

Figure 6. The event class hierarchy

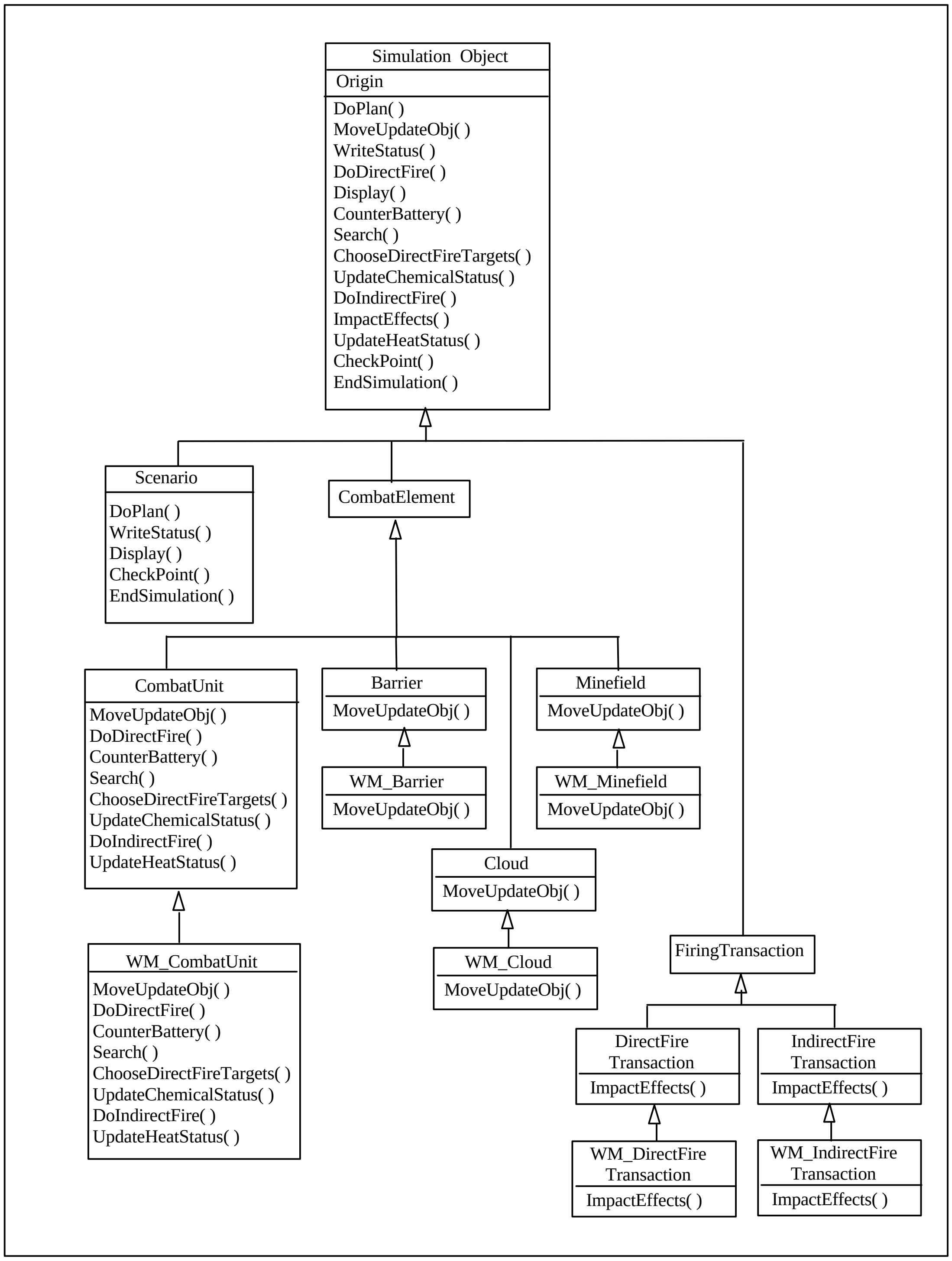

Figure 7. The simulation object class hierarchy

Every event now has an associated simulation object. This associated object is the target of the event. Depending on the subclass to which an event object belongs, the "execute" method of the event will invoke the corresponding event handler of the associated simulation object (Figure 7). The simulation object superclass defines the interface of the event handlers for the event groups. At the highest level, it provides an empty body as the default implementation for the event handlers. Events are dispatched to the appropriate subclass. If there is something more specific that needs to be done for instances of the subclass, the event handler of the subclass overrides the inherited method in order to simulate the desired behavior.

The above architecture enables a very simple realization of the main simulation loop:

```
initialization;
while not_empty(event_queue) loop
    e := remove_event(event_queue);
    e.execute( );
end loop;
finalization;
```

Note that this same code is used to handle all of the event handlers, including those for future extensions that have not yet been designed. Event objects with associated simulation objects are created and inserted into the event queue by the initialization procedure, the constructors of an object, and the actions of other event handlers. Depending on the actual event, it is inserted into an event priority queue based on time and priority.

Our newly designed architecture eliminates the need for the simulation loop to know what kind of object it is handling. Thus when adding an object type not yet designed, the simulation loop does not require additional code to invoke the new object's event handlers. This eliminates the possibility of introducing errors into the existing parts of the simulation by localizing all changes to the newly added object class.

4. **Design Validation via Prototyping**

In order to validate the proposed architecture and to refine the interfaces of the Janus subsystems, we developed an executable prototype using CAPS. Due to time and resource limitations, we only developed the prototype for a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (MoveUpdateObj, DoPlan, and EndSimulation), and one kind of post-processing statistics (fuel consumption) [3]. Figure 8 shows the top-level structure of the prototype, which has four subsystems: Janus, Gui, Jaaws and the Post-Processor. After we entered the prototype design into CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems. In addition, a simple user interface was developed using CAPS/TAE [19] (Figure 9). The resultant prototype has over 6000 lines of program source code, most of which was automatically generated, and contains enough features to exercise all parts of the architecture.

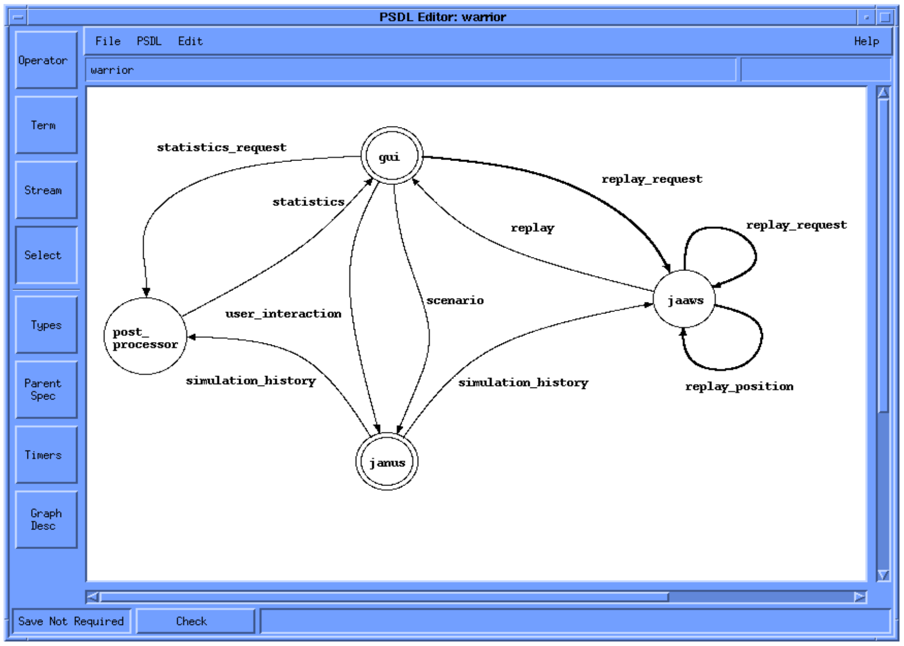

Figure 8. Top-level decomposition of the executable prototype

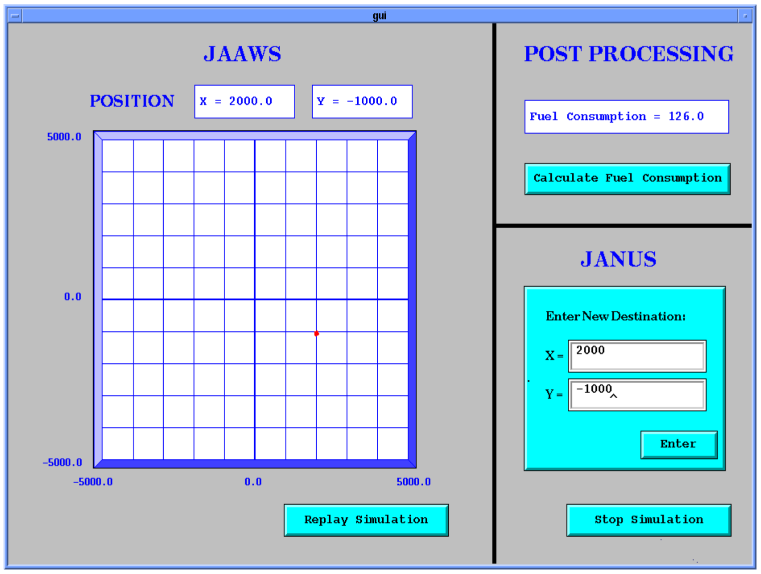

Figure 9. The Graphical User Interface of the executable prototype

5. **Conclusions**

Our conclusion is that substantial and useful computer aid for re-engineering is possible at the current state of the art. Human analysts must also play an important part of the process because much of the information needed to do a good job is not present in the software artifacts to be re-engineered. Success depends on cooperation between skilled people and appropriate software tools.

The missing information needed for re-engineering is related to deficiencies of the current system at all levels, from requirements through design and implementation. Thorough and accurate knowledge of these deficiencies is crucial for success. The clients never want the re-engineered system to have the exactly same behavior as the legacy system - if they were satisfied, there would be little motivation to spend time, effort, and resources on a re-engineering project. Even if a system is being re-engineered for the ostensible goal of porting to different hardware, the desired behavior at the interface to the hardware and systems software will be different.

In practical situations, the requirements for the re-engineered system are different from those for the legacy system. Key parts of the requirements for the new system are often missing or incorrect on the legacy documents. Some of that information is present only in the minds of the clients, often fragmented and scattered across members of many different organizations. Communication is a large part of the process, and that communication cannot be automated away, although it can be enhanced by appropriate use of prototyping.

Uncertainties about the true requirements play a central role in both re-engineering and the development of new systems. We therefore hypothesized that prototyping could play a valuable role in re-engineering efforts. Our experience supports that hypothesis.

We also found that prototyping can contribute substantially to the process of inventing, correcting, and refining the conceptual structures on which the architecture of the new system will be based. Most legacy systems are too complicated for individuals to understand. We found that constructing even a very thin skeletal instance of the proposed new architecture raised many issues and enabled us to correct, complete, and optimize the architecture for both simplicity and performance. (See [3] for lessons learned from the prototyping effort.) This was done before the architecture had grown into a maze of dependent designs and implementation details. Consequently, the changes could be realized without incurring the large cost and time delays typically encounted later in the development.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. The UML interaction diagrams lack the preciseness to support automatic code generation for the executable prototype. Such weakness can be remedied by the use of the prototype language PSDL [10, 11] and the CAPS prototyping environment, which provide effective means to model the system’s dynamic behavior in a form that can be easily validated by user via prototype demonstration.

8. **References**

[1] D. Berry, Formal Methods: The Very Idea, “Some Thoughts About Why They Work When They Work,” Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems, 1998, pp. 9-18.

[2] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, *Re-engineering the Janus(A) Combat Simulation System*, Technical Report NPS-CS-99-004, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.

[3] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, “Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping,” to appeare in the journal *Design Automation for Embedded Systems*. A preliminary version of the paper also appeared in *Proceedings of the 10th IEEE International Workshop in Rapid Systems Prototyping*, Clearwater Beach, Florida, 16-18 June 1999, pp. 216-221.

[4] O. Bray and M. Hess, “Reengineering a Configuration-Management System,” *IEEE Software*, Vol. 12, No. 1, Jan. 1995, pp. 55-63.

[5] V. Cabaniss, B. Nguyen and J. Moregenthaler, “Tool Support for Planning the Restructuring of Data Abstractions in Large Systems,” *IEEE TSE*, Vol. 24, No. 7, July 1998, pp. 534-558.

[6] *Janus 3.X/UNIX Software Programmer's Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[7] *Janus Version 6 User's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[8] *Janus Version 6 Data Base Management Program Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[9] S. Jarzabek and P.K. Tan, “Design of a Generic Reverse Engineering Assistant Tool,” *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, 1995, pp. 61-70.

[10] B. Kraemer, Luqi, and V. Berzins, “Compositional Semantics of a Real-Time Prototyping Language,” *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, May 1993, pp. 453-477.

[11] Luqi, V. Berzins, and R. Yeh, “A Prototyping Language for Real-Time Software,” *IEEE Transactions on Software Engineering*, Vol. 14, No.10, October 1988, pp. 1409-1423.

[12] Luqi and M. Ketabchi, “A Computer-Aided Prototyping System,” *IEEE Software*, Vol. 5, No. 2, 1988, pp. 66-72.

[13] Luqi, “System Engineering and Computer-Aided Prototyping,” *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, Vol. 6, No. 1, 1996, pp.15-17.

[14] Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo and B. Shultes, “The Story of Re-engineering of 350,000 Lines of FORTRAN Code,” *Proceedings of the 1998 ARO/ONR/NSF/ DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, 23-26 October 1998, pp. 151-160.

[15] M. Moore and S. Rugaber, “Domain Analysis for Transformational Reuse,” *Proceedings of 4th Workshop on Reverse Engineering*, IEEE Computer Society, 1997, pp. 156-163.

[16] L. Rieger and G. Pearman, “Re-engineering Legacy Simulations for HLA-Compliance,” *Proceedings of the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC),* Orlando, Florida, December 1999.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenzer, *The Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[18] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.

[19] *TAE Plus C Programmer’s Manual (Version 5.1*). Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

[20] J. Williams and M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X*, master’s thesis, Naval Postgraduate School, Dept. of Computer Science, Monterey, CA, March 1999.