DEVELOPMENT OF AN ABSTRACT USER INTERFACE TO SUPPORT MULTI-MODAL INTERACTION

David R. Pratt, Anthony J. Courtemanche and Mary Ann Pigora
Science Applications International Corporation
Orlando, Florida

Abstract

It is well known that there is no single optimal, universal user interface (UI) paradigm that can accommodate all the tasks a computer generated forces (CGF) user might be expected to perform. Plan view displays and buttons on two dimensional graphical user interfaces are common UIs for many CGF systems. However, other CGF users may require more diverse UIs such as three dimensional views, text message sending and receipt capabilities, and verbal interactions with the system. An abstract UI model can be used to provide flexible multi-modal support for various UI paradigms. It also isolates the interaction mode from the remainder of the system and thus contributes to system modularity and composability. Both of these abstract UI features are needed for future CGF systems that must satisfy multiple user needs.

The authors have investigated the usefulness of the abstract UI concept by developing an entity-based CGF system that uses an abstract UI to support a combination of graphical, textual and voice/speech synthesis UIs concurrently. Through the abstract UI, the most appropriate of the three UI paradigms for any given task can be selected. The abstraction was achieved by the use of a UI framework that supports all the user interactions within the CGF infrastructure. Specific UI classes were derived from the base abstract UI class using inheritance. Methods of the appropriate derived UI classes were invoked through polymorphism. Through the use of inheritance and polymorphism, the CGF infrastructure has a consistent view of the abstract UI component.

To further abstract the UI component, all user interface communications were carried out via a series of messages. Using this implementation of the Command Pattern, we were able to extend the functionality of one of the interface paradigms without affecting the implementation of the others. This allows for added flexibility of the user interactions and the construction of paradigm specific interactions.

This paper covers the development of the abstract UI. It discusses the base and derived UI classes and describes the use of messages. The benefits and limitations of this approach are presented as a foundation for future work.

Author Biographies

Dr. David R. Pratt is a Chief Scientist/Fellow at SAIC's Applied Software Systems Engineering Technology (ASSET) Group. Prior to joining SAIC, he served as the JSIMS Technical Director, a tenured Associate Professor of Computer Science at the Naval Postgraduate School, and a Captain in the Marine Corps. Dr. Pratt received a Ph.D. in Computer Science in 1993 and a Masters of Science degree in Computer Science in 1988 from Naval Postgraduate School.

Anthony J. Courtemanche has over ten years of experience in virtual entity simulations utilizing Computer Generated Forces, with contributions in software architecture, weapons systems simulation, targeting behaviors, network simulation protocols, and user interfaces. Mr. Courtemanche was one of the principal contributors to the ModSAF architecture and was the Project Engineer for ModSAF system development. At SAIC, Mr. Courtemanche is a Chief Scientist in the Advanced Distributed Simulation Research Team and supports research projects in the areas of object-oriented software architectures, advanced software technologies, and Computer Generated Forces. Mr. Courtemanche received his Master of Science in Electrical Engineering and Computer Science from MIT in 1987.

Mary Ann Pigora received her MS in Computer Science from the Georgia Institute of Technology. Currently she is employed by SAIC in the field of Human Simulation and Synthetic Natural Environment modeling. She has previously worked in the field of virtual reality and as a Technical Director for Walt Disney Feature Animation.

DEVELOPMENT OF AN ABSTRACT USER INTERFACE TO SUPPORT MULTI-MODAL INTERACTION

David R. Pratt, Anthony J. Courtemanche and Mary Ann Pigora
Science Applications International Corporation
Orlando, Florida

INTRODUCTION

It is well known that there is no single optimal, universal User Interface (UI) paradigm that can accommodate all the tasks a Computer Generated Forces (CGF) user might be expected to perform. Plan view displays and buttons on two dimensional graphical user interfaces are common UIs for many CGF systems. However, other CGF users may require more diverse UIs such as three dimensional views, text message sending and receipt capabilities, and verbal interactions with the system. An abstract UI model can be used to provide flexible multi-modal support for various UI paradigms. It also isolates the interaction mode from the remainder of the system and thus contributes to system modularity and composability. These abstract UI features are needed for future CGF systems that must satisfy multiple user needs.

The authors have investigated the usefulness of the abstract UI concept by developing an entity-based CGF system that uses an abstract UI to support a combination of graphical, textual and voice/speech synthesis UIs concurrently. The abstract UI allows the most appropriate UI paradigm for any given task to be selected. This paper covers the development of the abstract UI. It discusses the base and derived UI classes and describes the use of messages to carry out UI communications. The benefits and limitations of this approach are presented as a foundation for future work.

A SIMPLISTIC COMPONENT ARCHITECTURE

A simplistic view of the CGF testbed component design is shown in Figure 1. The Simulation Engine component is considerably more complex than portrayed, but we treat it as a black box for the purposes of this paper. The three other main CGF system components, Data Distribution Module (DDM), Command and User Interface, are discussed further below.

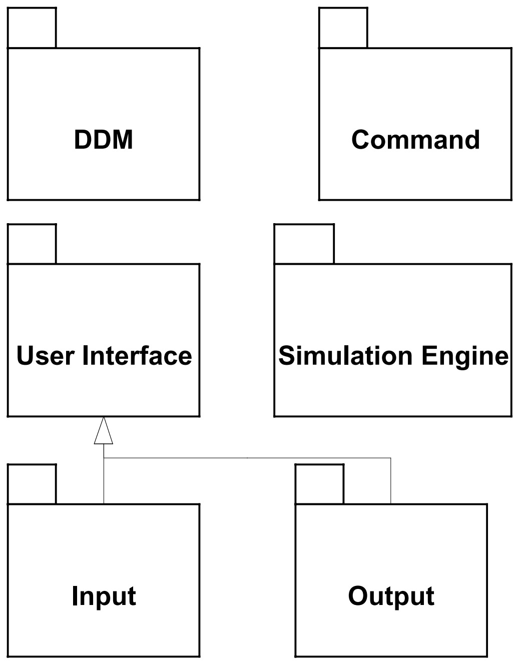

Figure 1. Simplistic Component Architecture

The fundamental aspects of a composable system that drove the design of our CGF system have been detailed in [1]. One of these was the use of documented design patterns [2]. Using documented design patterns, developers can step back away from the details of how a system is supposed to work and view the system from a broader perspective. They can examine the system s structure and better understand the coupling between its components.

A key to enabling composability is to have the Mediator Pattern act as the interface among components of a system. For our CGF system, the Data Distribution Module (DDM) provides this functionality. With the DDM acting as the interface among the CGF system components, the UI is completely decoupled from the Simulation Engine (e.g., no direct calls between the two components).

Ideally, the DDM should not possess any semantic knowledge of the data it is moving around the system, other than the source and destination of the data. Using the object oriented paradigm, one can encapsulate all aspects dealing with data transmission or routing in a base class and have messages that are derived classes with the semantic content. In our CGF system, this is accomplished by having all methods that deal with the DDM and all data transmission aspects of the message contained in the base class. All methods that have semantic content are in the derived classes, which deal with a particular type of message. The result is an implementation of the Command Pattern. The Command component implements the message, or token, portions of the pattern. The separation of the messages from the DDM helps reinforce the conceptual separation between the two components.

For simplicity, the User Interface component is comprised of two sub-components, Input and Output. In reality, however, the two sub-components have very little in common. When the input and output sub-components are viewed separately, the result is the traditional Model - View (output) - Controller (input) Pattern or MVC Pattern. In this abstract view, there is no direct relationship between the Input and Output sub-components. Instead, they interact through the Simulation Engine (model) via the DDM.

The unidirectional nature of the User Interface components provides a degree of abstraction that allows the various input and output paradigms to be mixed and matched. In reality, however, there are instances when a given user interface has to be bi-directional (say, for performance and simplicity reasons). The need for bi-directionality leads to a coupling between the instances of the Input and Output classes as will be discussed later in the paper.

INITIAL DESIGN

The CGF testbed was built using an evolutionary development model with two phases, initial and final design. As a research project, we did not envision bringing forward a significant amount of code from the first phase to the second. Hence, we purposely sacrificed performance for the sake of simplicity and sound software engineering principles during the initial design phase. Java was selected as the implementation language to provide a means to fully evaluate the language suitability for entity-based CGF systems.

In general, the choice of programming language should not influence the design of the architecture or of the class structure. However, the lack of multiple inheritance capabilities in Java forced several design compromises. In both the Input and Output Java packages, we made use of a Java interface to define the public face of the component (public methods). The need for the Java interface became apparent during the development of the PVDClass, a class to support the Plan View Display (PVD) user interface.

By its very nature, the PVD is a bi-directional interface. This class needed the functionality that was encapsulated in an abstract input class as well as an abstract output class. Since it was not possible to simply derive the PVDClass from the two abstract classes using Java, we arbitrarily decided to have the PVDClass implement an input class interface and extend (inherit from) an output class. In C++, we would have simply derived the PVDClass from both abstract classes using multiple inheritance with no need for using an interface construct.

The decision to implement the CGF system as a Discrete Event Simulation (DES) system patterned after the design contained in [3] was made early on. Specific event classes for the desired actions had to be created. Event classes that could be placed on an event queue so actions could take place at a certain time appeared in both the Input and Output packages.

To evaluate the modularity of the system, we implemented three different modes of user interaction. The first mode, a speech generation/voice input interface, was implemented using IBM’s ViaVoice and the Java Speech API (JSAPI)[4]. While this limited the system to Windows/Intel-based platforms, the implementation of the required infrastructure was remarkably straightforward. The second mode was a textual interface. This allowed the user to enter keyboard commands into a text input widget and to view the results in a scrolling window. The final mode of interaction was a Graphical User Interface (GUI). Widgets and the ability to select icons from a PVD comprised the input side of the GUI. The PVD also served as the principal means of output for the GUI.

Input Package

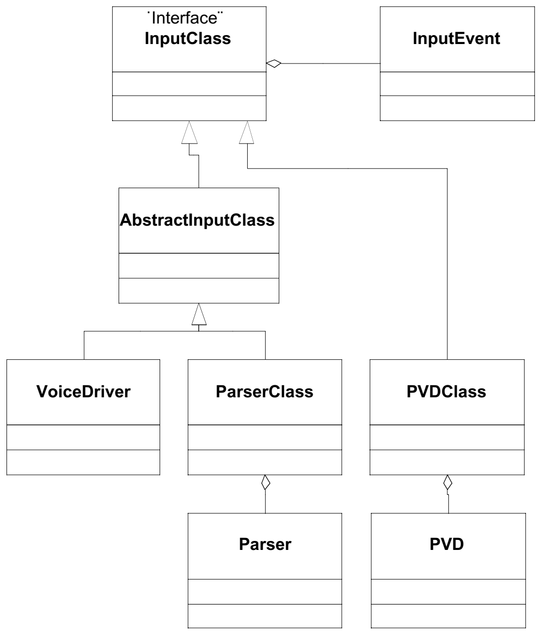

Figure 2. Initial Input Class Hierarchy

The classes that comprised the initial Input Package are shown in Figure 2. The AbstractInputClass contains the default implementations for all the public methods described in the InputClass Interface. With the exception of the communication methods that interact with the DDM, most of the methods were simply stubs (no code in the body of the method). This served two purposes, the most obvious is the implementation of a single DDM interface. Another rationale for the stubs was that they allowed the evolutionary construction and testing of the subclasses.

We arbitrarily decided that the PVD was primarily an output device. Since the PVDClass did not extend AbstractInputClass, it had to implement all of the methods contained in the InputClass interface explicitly. It is interesting to note that the Parser and PVD classes actually made use of composition vice inheritance to implement their functionality associated with the Input Package. For both classes, this was a direct result of Java s lack of support for multiple inheritance. Namely, the two classes required windows to open on the screen so both classes had to also extend the Java Frame class. Thus, additional class functionality had to be provided via class composition.

While not shown in Figure 2, the VoiceDriver also made extensive use of JSAPI's *javax.speech* package's classes and IBM's Via Voice Java speech processing classes.

Output Package

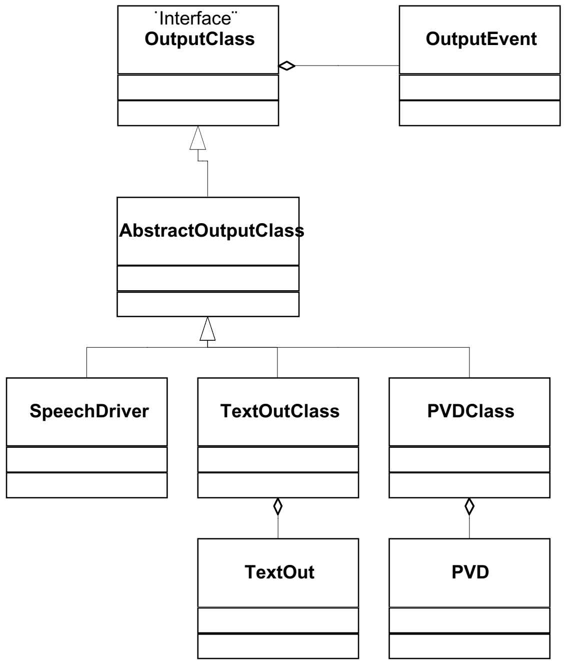

Figure 3. Initial Output Class Hierarchy

As expected, the initial Output Package was very similar to the initial Input Package (see Figure 3). The key difference was that the PVDClass extended AbstractOutputClass vice implementing the OutputClass interface. By doing so, PVDClass inherited the implementations defined in AbstractOutputClass.

Command Package

As mentioned previously, the Command Pattern was used to separate the transmission mechanism from the information content of the message. While use of the pattern significantly simplified the DDM, commands still had to be encoded and decoded. Early in the design phase, we researched current input and output paradigms to determine the optimal representation of the Command class. The synthesis of the Command class used in our CGF system was based upon existing systems, proof of principle demonstrators, and developer's insights.

We found that two native representations were commonly used. The first one was the traditional, explicit representation involving the use of a set of discrete fields. This drove the need for a command hierarchy, which was the preferred representation for the GUI and Simulation Engine. The second representation involved the use of a vector of tokens. In this representation, each token in the message was stored in its ASCII representation as an element of a vector. This representation was native for the Speech and Text interfaces. The Command class contained both representations internally as shown in Figure 4.

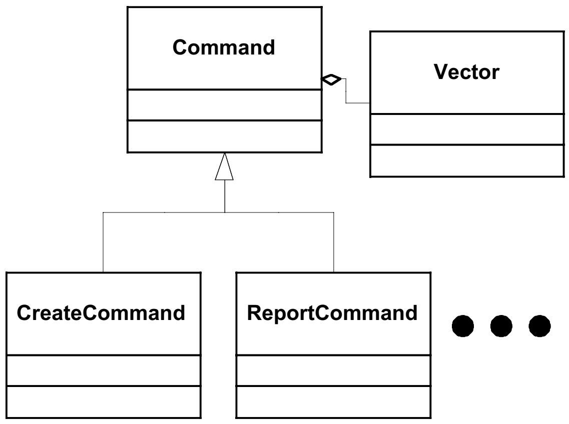

Figure 4. Initial Input Command Hierarchy

To ensure consistency of both representations, a command could be created using either format. It was then immediately converted to the other format. Thus, it was not possible to determine which interface generated the command.

The structure of the command s text content was encoded in a grammar. This same grammar was used to define the tokens recognized by the voice and text input systems. By limiting the grammar to a small number of relevant phrases, the voice recognition accuracy increased dramatically over the use of a free-form speech-to-text converter. The definition of the grammar and the parsing mechanism is an embedded feature of the JSAPI.

FINAL DESIGN

After the first build of the CGF system, we evaluated the architecture to determine which constructs should be brought forward, modified, or reimplemented. All of the components were modified somewhat. There was a major change in the fundamental DDM approach of the final system, as described in [5]. Rather than using the broadcast method, we implemented a publication and subscription multicast system. While this required more DDM infrastructure, it allowed for significantly improved scalability and simpler components.

In the initial system build, we also noticed frequent interactions among the User Interface, DDM and the Simulation Engine. As part of the abstraction, the DDM kept track of where an entity was computationally managed while the Simulation Engine kept track of all of the simulation aspects of the entity. Thus, when a user went to select an entity, the system had to look up where it was being managed and determine what type of entity it was. Calls to the DDM and to the Simulation Engine were also needed to determine the correct routing for the action. There were several suggestions about how we could improve the efficiency of these interactions. However, each of them were rejected because they violated our goals to achieve encapsulation and minimalist interfaces between the components. In this section, we discuss the resulting system and justify the modifications made to the initial design.

Input Package

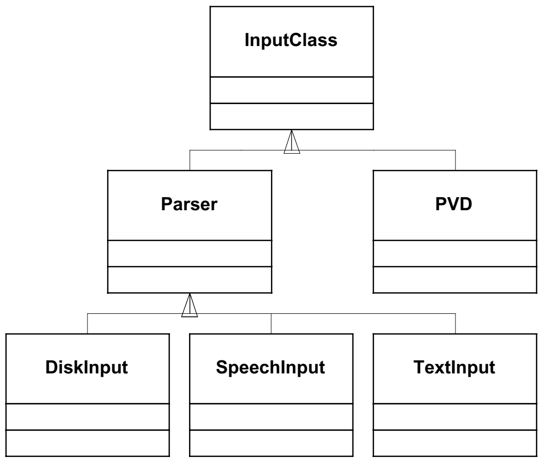

Figure 5. Final Input Hierarchy

Of the three packages discussed, the Input Package underwent the least change. As shown in Figure 5, the major changes were the elimination of the InputClass interface and the addition of the Parser class. The interface became redundant when we redesigned the classes so that the base class became the only class that implemented it (note, InputClass has been reused as the name of the base class in the final design). The Parser class was constructed to parse the textual input from the speech and text input modes. When we abstracted out the parsing process, it proved to be a simple matter to add the ability to read files from a disk. In many ways, this validated the flexibility and extensibility of our abstract UI design. While the system is DES based, the event classes are no longer tightly coupled to the Input and Output hierarchies. Thus, they are not shown in the diagrams.

Output Package

While the Input Package demonstrated a fair amount of commonality with its previous design, the Output Package exemplified greater differences in the implementation of the methods in the leaf classes. The output mechanisms ended up being sufficiently different that there was very little, if any, duplicated code in the classes. For this reason, we separated out the classes with a common interface specification. The final class hierarchy, shown in Figure 6, is much more like a classic component-based design than it is an object oriented design.

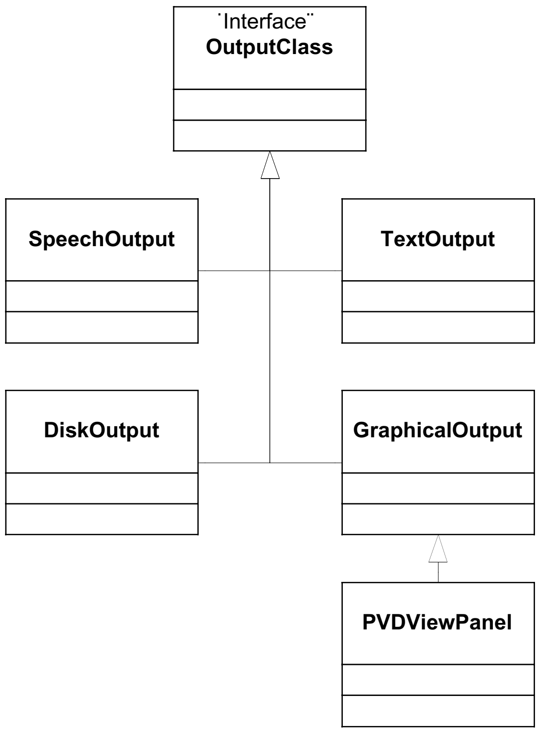

Figure 6. Final Output Hierarchy

Command Package

The implementation of commands in the final system was heavily influenced by the implementation of the revised DDM system. The DDM infrastructure supported the transmission of arbitrary Memos, labeled by some subject, delivered to a receiver, and sent from a sender. A Memo encapsulates MemoContents. The overarching system requirement imposed by the DDM infrastructure was that all parts of a Memo must be serializable. The Java serialization capability allows an object to be written to a file or a network stream, and have the object reconstituted when the file is read or the stream is received. In Java, serialization is enabled for a class when it implements the *java.io.Serializable* interface.

Since the DDM infrastructure required that MemoContents be serializable, this allowed native Java classes to be the representation of choice when communicating members of the Command class hierarchy. As long as the Commands were serializable, and as long as the Commands provided the appropriate accessors to extract the relevant command-specific information, no additional effort was required to package and extract command information.

To represent Commands between simulated entities, or between the operator and entities in the simulated world, a hierarchy of Orders and Reports were developed, as shown in Figure 7. Note that the commands identified in the figure are only a subset of the Orders and Reports implemented. Besides Orders and Reports, other Command hierarchies were developed. These included Warfighter Interactions (similar to DIS PDUs), Requests (such as requests for status information from simulated entities or system requests to create simulated entities or control measures), and Confirmations (to confirm the result of a Request).

Commands were created by the entities or the user interface, and transmitted to their destination though the interest management mechanisms in the DDM. Receivers desiring to be notified of Commands would subscribe their interest with the DDM, perhaps by specifying the Memo subjects that they were interested in, or by specifying interest in Memos from a particular sender. The DDM infrastructure would, in turn, notify these interested parties by delivering Memos of the appropriate type to them when available. Simple clients of these Memos, such as the TextOutput, would simply convert the Memo to a string (using native Java capabilities) and send the string to a textual output widget. Other classes, such as the PVDViewPanel, would use various methods in the Memo to extract the appropriate content and use that information appropriately.

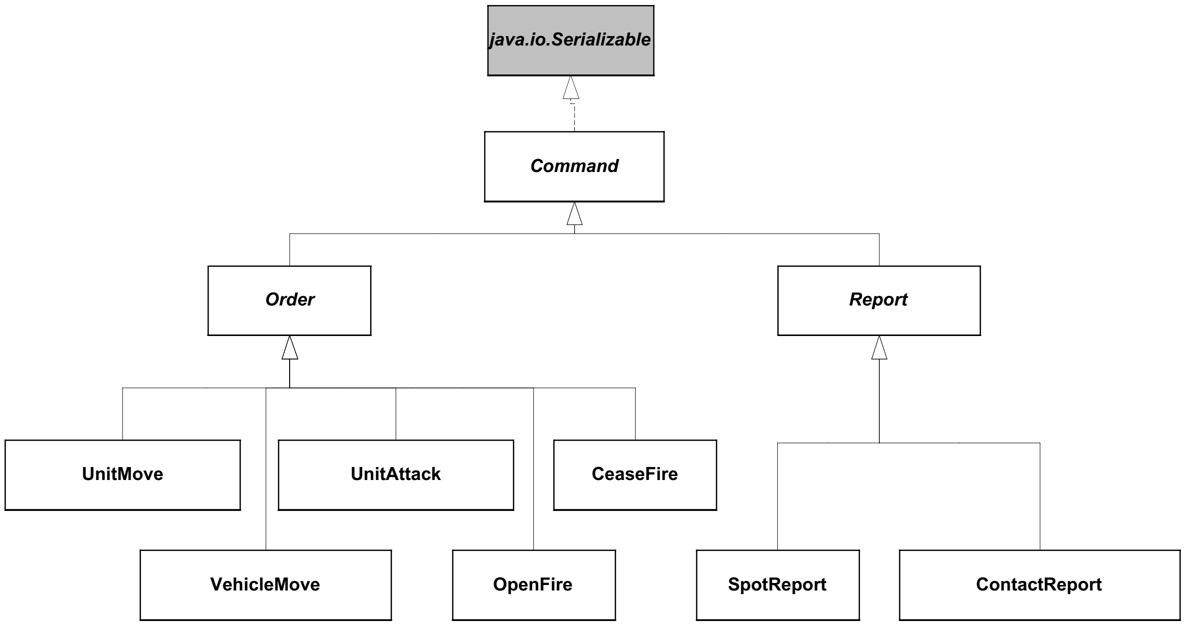

Figure 7. Order and Report Commands

While it was very straightforward to make any class transportable as a Command (by implementing the serializable interface), it was important to make sure that that the entire object was serializable. For example, an Order containing a formation parameter required that the object representing a formation was also serializable. In effect, the entire closure of referenced objects had to be checked to enforce proper serializability. In some cases, certain classes such as the FormationEnum used to describe formations had to contain additional deserialization methods in order to guarantee that the formation deserialized from a Memo transmission turned into a reference to one of the local valid formation enumerations, rather than just being a *copy* of one of the enumerations. This was required to make sure that one and only one copy of each element in an enumeration was in existence. Then enumeration comparisons could be made using equality checks.

USER INTERFACE METHODS

Throughout this project, we focused on adhering to good software engineering practices. Among them include the principles of encapsulation and information hiding. In doing so, we developed minimalist input and output interfaces that all of our UI components needed to support. We believe that while not formally constituted as such, the result is a Domain Architecture for the User Interface portions of CGF systems. The more important methods are listed below. For the sake of conciseness, the attributes of the methods have not been included. Many methods had multiple implementations with varying numbers and types of attributes.

Input Methods

The Input package was fundamentally a source of data; it operated in a only in a publish mode. As such, it did not subscribe to receive any messages from the interest management groups. It was just responsible for sending the correct messages to the DDM. The DDM would then figure out how to route the messages as appropriate. The Input package did have to interact with the DDM Identifier Management tables in order to request new entities to be constructed and to look up which entity the user desired to manipulate.

buildMemo() —This method is used to build a template for a memo. Via polymorphism it takes a variety of attributes and fills in the attribute values of the memo. In essence, this is a factory method for the Command class.

sendMemo() —This method passes a memo to the DDM.

createThing() —This method triggers a series of actions to create a new Thing (an entity or component of an entity). The Thing is created in the Input Package space and is not sent out to the rest of the system. This allows the user to assemble an entity from its component parts completely before passing control of it to the Simulation Engine and registering it with the DDM.

createThingRequest() —This method requests that a Thing be created in the Simulation Engine and registered with the DDM. This is the final step in the construction of a Thing.

validID() —This method determines whether a given ID is valid.

findID() —This method looks up an ID based upon some attribute. It allows the user to think of entities in domain terms vice DDM terms.

interfaceCommand() —This method passes the exec u-tion of a user input command to a different portion of the system. It is used for system level commands such as startup and shut down of display devices, pause, restart, etc.

panCommand() —This method shifts the focus of user interest in a given direction or to a given area. It forces the DDM to update the subscription and publications of the input device.

select() —This method has two different meanings. One is to select a given entity for query and update. The other is to choose something from a range of options to cause an event to transpire.

clearSelect() —This method is called when the user releases focus of interest from a given selection.

deleteSelected() —This method is called to delete the currently selected entity.

outputError() —This is a system level method that serves to pass the error messages along to the main system for processing. Since the Input Package is unidirectional interface, it has no way of notifying the user when an error has occurred.

Output Methods

Since we took advantage of the power of polymorphism, there were considerably fewer output methods needed. Each type of interface "knows" how to "display" the given content on the message.

subscribe() —This method expresses interest to the DDM about an area, event, or entity. This starts the information flow to the output class.

unsubscribe() —This method expresses discontinued interest in an area, event, or entity. This will stop the information flow from a particular subscription.

updateOutput() —This method causes an update of the entire output class. It is used for a device-wide refresh capability and is used most often for a device that has a large number of entities displayed at the same time.

updateSelected() —This method is used to update a selected item. On some devices, such as voice, only a selected entity should be updated at a time.

deallocateOutput() —This method shuts down the output channel in an orderly fashion.

SUMMARY

A user interface abstraction has been achieved through the use of a framework that supports all user interactions within the CGF infrastructure. Specific UI classes were derived from base abstract UI classes using inheritance to implement the specific UI paradigm. Methods of the appropriate derived UI classes are invoked through polymorphism. Through the use of inheritance and polymorphism, the CGF infrastructure has a consistent view of the abstract UI component. A set of base classes with a set of minimalist methods was developed that can serve as the foundation for future system development.

All user interface communications were carried out via a series of messages. We were able to extend the functionality of one of the interface paradigms without affecting the implementation of the others by using an implementation of the Command Pattern. This allowed for added flexibility of the user interactions and the construction of paradigm specific interactions.

This project exhibited success on several levels. The most notable success was the implementation of a component-based system that exploited the benefits of object oriented programming. However, the abstraction often forced a more computationally expensive solution. Historically, performance degradation has been a reason for not using good software engineering practices. But with the increases in performance and reduction in cost of hardware, we feel this argument no longer holds true. Rather, the cost of programmer time to develop and maintain a system is fast becoming the driving factor in system costs. Object oriented, modular systems, like the one described in this paper, represent a compromise between the user's functionality and composability requirements and the programmer's ability to effectively develop and maintain increasingly larger software systems.

ACKNOWLEDGMENTS

This work was funded by the SAIC Industrial Research and Development (IRaD) program.

REFERENCES

[1] Pratt D., Ragusa, L.C., and von der Lippe, S. "*Composability As An Architecture Driver*", Interservice/Industry Training, Simulation and Education Conference, Orlando, Florida, November - December 1999.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns – Elements of Reusable Object-Oriented Software , Addison-Wesley Publishing Company, Inc. 1995.

[3] Altman, M. Writing a Discrete Event Simulator: ten easy lessons, http://www.labmed.umn.edu/~michael/des/

[4] Sun Microsystems, Inc. Java Speech API Refe r-ence Page http://www.javasoft.com/products/java-media/speech/

[5] Courtemanche, A. *The Incorporation of Val i-dated Combat Models Into a Discrete Event Simulation (DES) CGF* , Proceedings of the Ninth Con - ference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, May 2000.