REVERSE ENGINEERING TO SUPPORT MODIFICATION OF JSAF

Michael Echevarria
VisiTech, Ltd.
Alexandria, Virginia

Stephen Kasputis, PhD
VisiTech, Ltd.
Alexandria, Virginia

ABSTRACT

The Joint Semi-Automated Forces (JSAF) simulation is a large program that has been modified across many developers. A complete and concise software architecture map of it has not previously existed. Such a map would assist new developers in understanding JSAF and speed development by allowing developers to quickly see relationships without having to conduct time-consuming searches of the documentation of several libraries. It also supports design of efficient regression tests of modifications by highlighting specific interactions that needed to be tested. JSAF was reversed engineered and organized with the assistance of software tools into a model. Requirements were then entered into the model and graphically traced down to the software. This end result provides a clear, graphical picture of the JSAF architecture. In addition to the benefits previously stated, this model helps document the connection between the software and requirements. This paper details the process of reverse engineering JSAF and the advantages of using a software model to implement development.

ABOUT THE AUTHORS

Michael Echevarria is a scientist/engineer with VisiTech, Ltd., in Alexandria, VA. He has worked as an intern and employee Digital Systems Resources and Planning Systems Incorporated. At those positions he conducted JSAF testing and debugging along with Rational Rose re-design and implementation of large legacy systems into modern object-oriented systems. He is currently a JSAF program developer for a prototype Marine Corps training system, and Rational Rose designer/analyst of the overall system. He has a BS in computer science from the University of Virginia/George Mason University and a minor in mathematical science from George Mason University.

Stephen Kasputis is the Chief Scientist of VisiTech, Ltd., in Alexandria, VA. He had held numerous positions in the Undersea Surveillance Program Office including Technical Director of the Fixed Distributed System. He has been the systems engineer for numerous simulation efforts. He is currently the systems engineer for the modification of JSAF for a prototype Marine Corps training system and directing development of advanced validation techniques. He has a BS in physics from Penn State, an MS in engineering acoustics from The Naval Postgraduate School, and a doctorate in acoustics from The Catholic University of America.

REVERSE ENGINEERING TO SUPPORT MODIFICATION OF JSAF

Michael Echevarria
VisiTech, Ltd.
Alexandria, Virginia

Stephen Kasputis, PhD
VisiTech, Ltd.
Alexandria, Virginia

I PROLOGUE

Software development involving legacy systems differs from the general process of software development. When working with legacy systems, the software design phase of development first requires recreation or reverse engineered. A good example of this is our work with the Joint Semi-Automated Forces (JSAF) simulation. We needed to modify this simulation for its role in the Distributed Virtual Training Environment (DVTE), a prototype Marine Corps training system.

JSAF is a large-scale legacy system that has been developed and modified across many software developers from many companies. These developers did not work in unison nor did they adhere to a uniform documentation standard. The end result is that it is difficult to develop a programming level understanding of the code from the documentation. Using a hands-on process of modifying the code is typically the best was to become familiar with it. To become truly adept at modifying JSAF requires several months of continually working with it because of intricacies of a system that contains 915 libraries. To bring new programmers up to speed on the system faster as well as to improve the effectiveness and efficiency of our testing of modifications, we sought development of a software architecture model that shows the class interaction between the many libraries of JSAF.

II INTRODUCTION

The software development process is generally described by two models. The first, the waterfall model (figure A), is a linear process. The second, the spiral model (figure B), is similar to the waterfall model, but is cyclic. Analysis of these models shows that software development models can be generalized into the following steps: requirements analysis, analysis, design, programming, and testing.

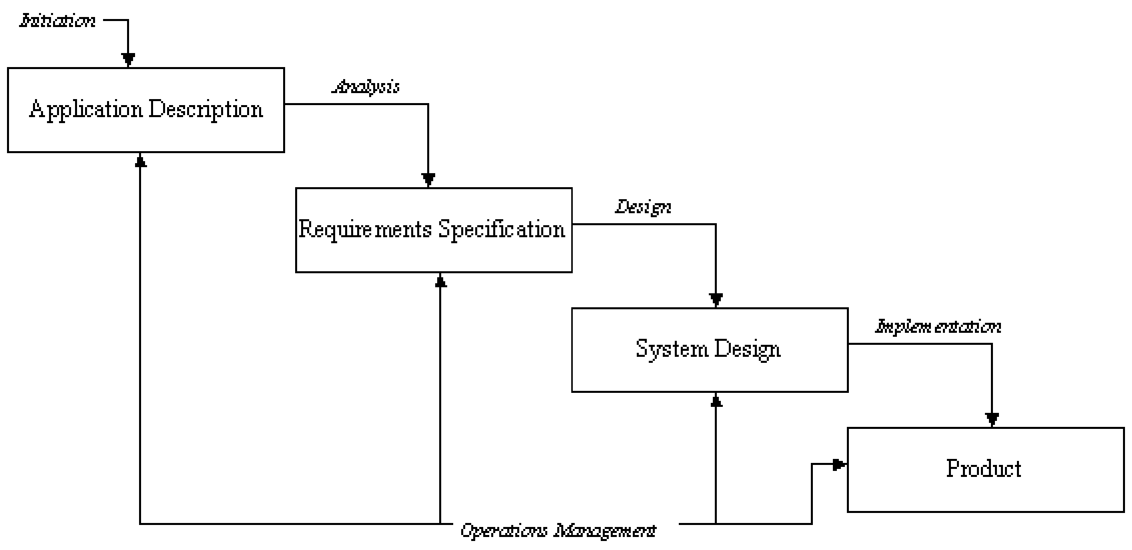


Figure A: The Waterfall Model

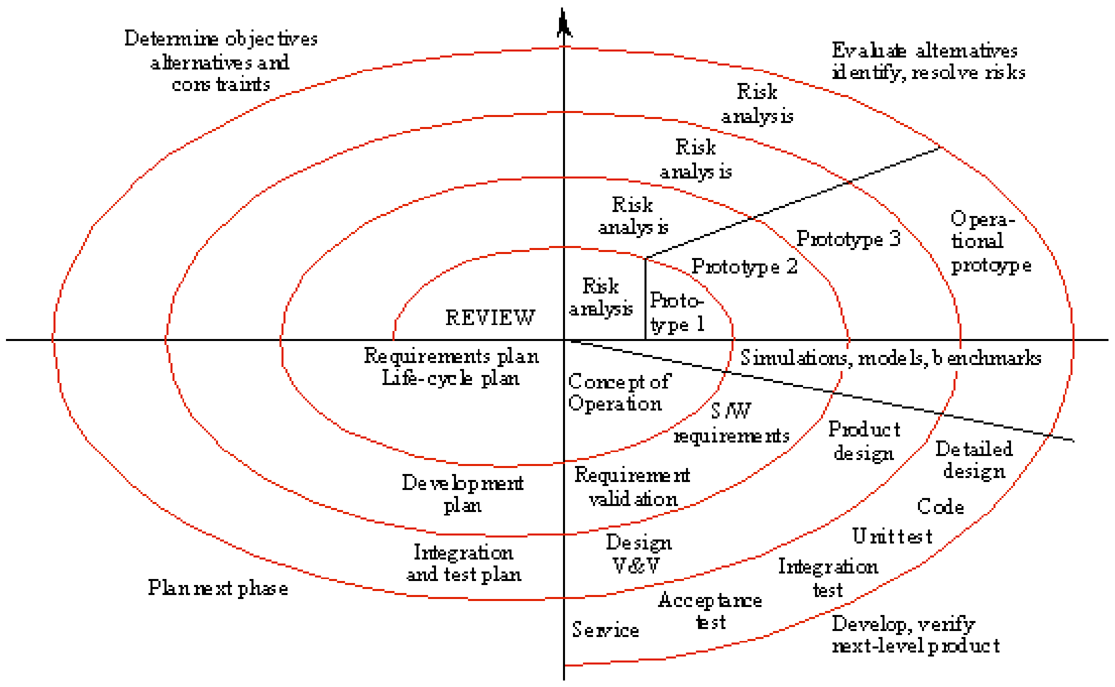


Figure B: The Spiral Model

Briefly, in the context of object oriented programming, the phases of software development can be described as follows. The requirements analysis phase defines what the user expects from the system. The analysis phase is concerned with the primary abstractions (i.e. classes and objects) and mechanisms that are present in the problem domain. The classes that model these are identified, along with their relationships to each other. In the analysis, only classes that are in the problem domain (real-world concepts) are identified. What is not identified at this level are technical classes that define details and solutions in the software system (i.e. classes for user interfaces, databases, etc.). In the design phase, the result of the analysis is expanded into a technical solution. New classes are added to provide the technical infrastructure, for example: the user interface, database handling to store objects in a database, communication with other systems, or interfacing to devices in the systems. The domain problem classes from the analysis are embedded into this technical infrastructure. The design results in detailed specifications for the programming phase. In the programming phase, the classes from the design phase are converted to actual code in an object-oriented programming language. The testing phase consists of unit tests, integration tests, system tests, and acceptance tests. The unit tests are of individual classes or a group of classes, and are performed by the programmer. The integration test integrates components and classes in order to verify that they cooperate as specified. The system test views the system as a 'black box' and validates that the system has the end functionality expected by an end user. The acceptance test are conducted by the customer to verify that the system satisfies his requirements in a typical operational context.

III RESEARCH FOCUS

The difficulties encountered with modifying legacy systems come in the analysis, design and testing phases. The analysis phase is difficult with JSAF because the documentation of library interactions is often outdated and incomplete. Compounding this is that fact that redundant instances of classes may appear in different libraries. The design phase is then affected by the lack of clarity from the analysis phase. Identifying the effects of modifying or adding classes and objects is very tedious and complex. Finally, the testing phase is convoluted by the lack of knowledge preceding it. End user tests can be performed to check for overall results, but class interactive tests are difficult to evaluate because it is not known how many or which classes are affected by the changes made.

The solution to problems created in the software development process by legacy systems such as JSAF has been to reverse engineer the system to aid in the analysis, design and testing phases. Our goal of the reverse-engineering process was to provide a clear understanding of the software architecture.

This was difficult for a system as large as JSAF. Complicating things further is the fact that JSAF is system comprised of several languages. Libraries in JSAF are written in C, C++, JAVA, and MYSQL. The reverse engineering of JSAF, therefore, required the aid of software tools. These tools facilitated checking each of the 915 libraries for references among the others. Additionally, selection of a common representation language for showing the linkage among the libraries of JSAF was essential. The Unified Modeling Language (UML) was chosen to represent the architecture of JSAF. Rational Rose, as an application of UML, was chosen to perform the process tasks for the reverse engineering.

IV IMPLEMENTATION DESCRIPTION

A software model of JSAF is an organized depiction of its interactions and classes. Rational Rose Enterprise Edition's functionality provides the data structure/class relationships in terms of UML and the framework for the user to create the system organization. We then provided organization and analysis of the resulting data. The creation the JSAF software model required us to organize the 915 software libraries into components, process the data of the components, and develop a use-case model explaining the overall functionality of the system.

V IMPLEMENTATION

The JSAF library documentation does have a system for organizing the libraries. The libraries are categorized into a three levels, each level becoming more specific. For example, the library DamEdit, a GUI for viewing and modifying component and subcomponent damage, is categorized as a GUI COMMAND SERVICE (see figure C). The documentation shows a three-tier architecture with a total of 300 categorizations. Looking through the actual library documentation shows that only 61 of the 300 categorizations are used. Also, some libraries are labeled with categorizations that are not in the overall hierarchy. An example of this is in the library EnvDust (library for modeling vehicle dust), which is labeled as SIM ENVIRONMENT SERVICE. The organization documentation does not contain a subcategory labeled 'environment'. Due to these discrepancies, we created an organization that more accurately reflected the JSAF system. The hierarchy keeps the original organization of a three level categorization, but excludes unused categories. The hierarchy was slightly reorganized to reflect the actual categorization of the JSAF libraries. The library hierarchies we developed for the software model of JSAF are show in Figures C through F.

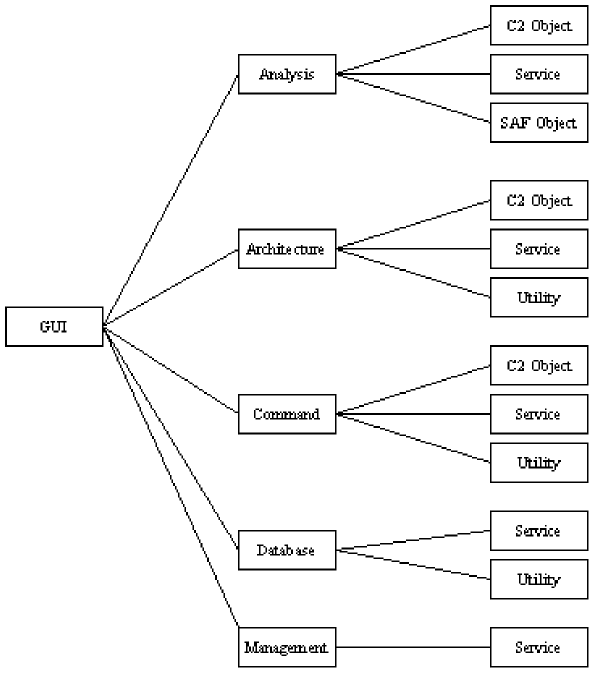


Figure C: GUI libraries

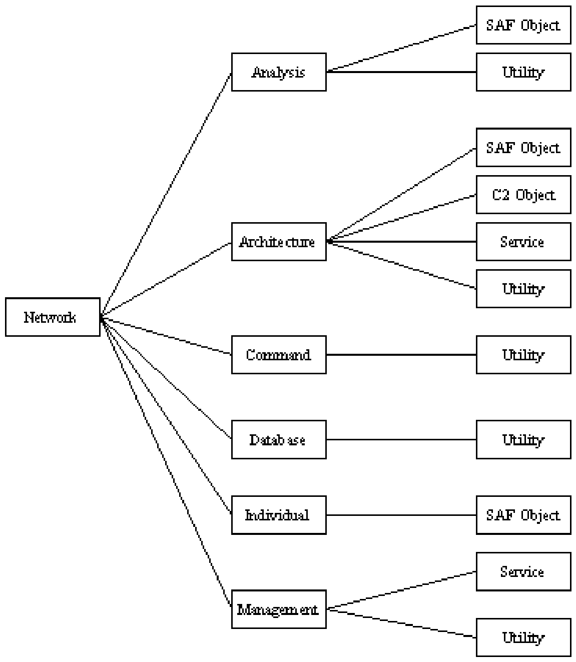


Figure D: Network Libraries

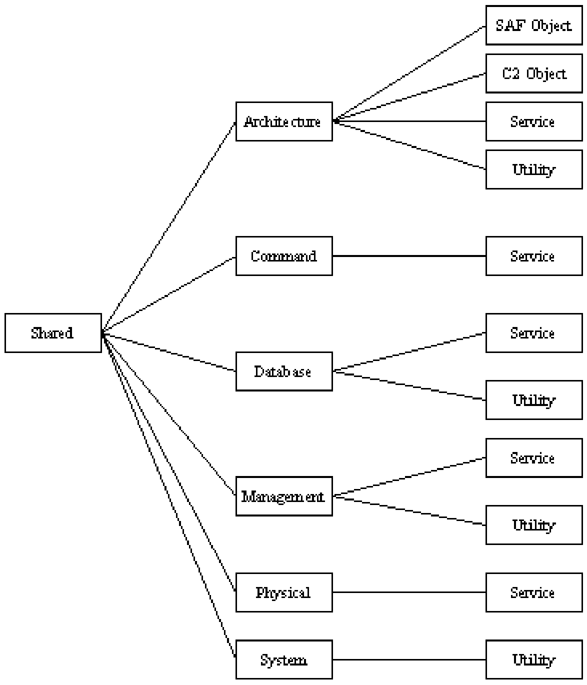


Figure E: Shared Libraries

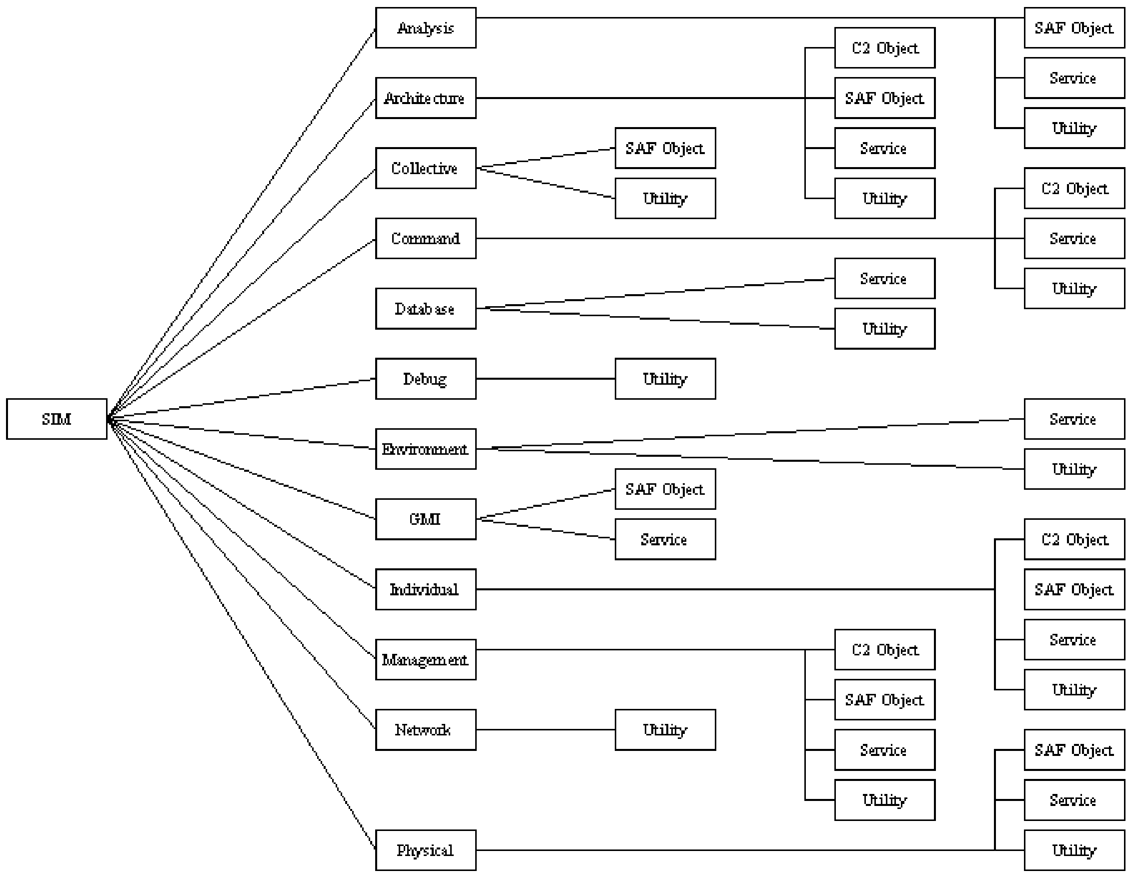
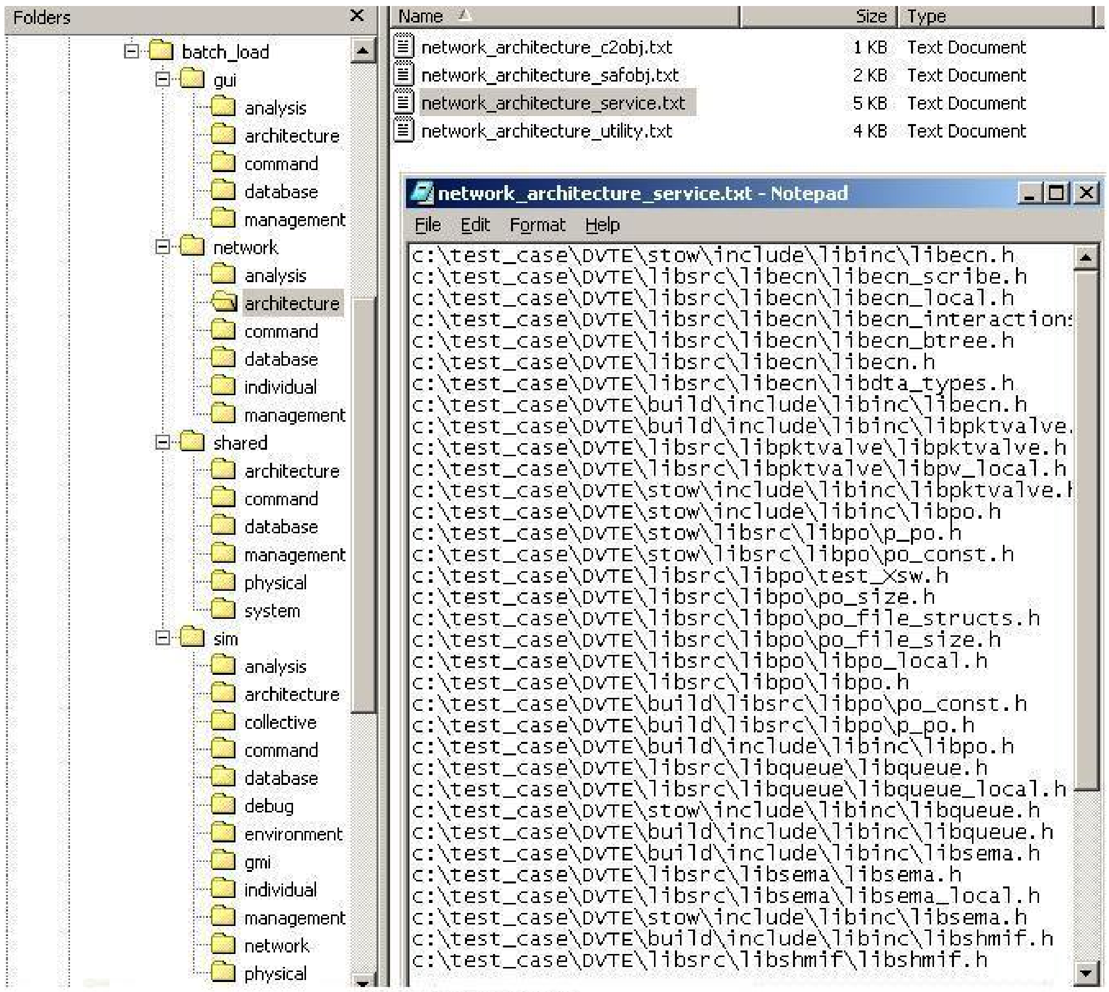


Figure F: Simulation Libraries

Once the components had been created within Rational Rose Enterprise under the 'component view' section, data to be modeled had to be provided for those components. The necessary data is the header files from the various classes within each library. For JSAF there are a total of 5863 header files corresponding to classes. Entering each library's header files into the component structure by hand would have been tedious. Rational Rose Enterprise does provide a 'batch load' option. To exercise this option, a text file with a list of the full paths of all the files was needed. To provide a listing of all the header files within JSAF, a JAVA program was written to capture the paths of all the header files and place the output into a text file. This large file was then parsed into smaller files that mapped to components within the software model. Figure G shows the Rational Rose screen for the batch load operation. The NETWORK ARCHITECTURE SERVICE component is shown with the full paths of the batch load files shown in the open text file.



Name	Size	Type
network_architecture_c2obj.txt	1 KB	Text Document
network_architecture_safobj.txt	2 KB	Text Document
network_architecture_service.txt	5 KB	Text Document
network_architecture_utility.txt	4 KB	Text Document

```
c:\test_case\DVTE\stow\include\libinc\libecn.h
c:\test_case\DVTE\libsrc\libecn\libecn_scribe.h
c:\test_case\DVTE\libsrc\libecn\libecn_local.h
c:\test_case\DVTE\libsrc\libecn\libecn_interaction:
c:\test_case\DVTE\libsrc\libecn\libecn_btree.h
c:\test_case\DVTE\libsrc\libecn\libecn.h
c:\test_case\DVTE\libsrc\libecn\libdta_types.h
c:\test_case\DVTE\build\include\libinc\libecn.h
c:\test_case\DVTE\build\include\libinc\libpktvalve.
c:\test_case\DVTE\libsrc\libpktvalve\libpktvalve.h
c:\test_case\DVTE\libsrc\libpktvalve\libpv_local.h
c:\test_case\DVTE\stow\include\libinc\libpktvalve.h
c:\test_case\DVTE\stow\include\libinc\libpo.h
c:\test_case\DVTE\stow\libsrc\libpo\p_po.h
c:\test_case\DVTE\stow\libsrc\libpo\po_const.h
c:\test_case\DVTE\libsrc\libpo\test_xsw.h
c:\test_case\DVTE\libsrc\libpo\po_size.h
c:\test_case\DVTE\libsrc\libpo\po_file_structs.h
c:\test_case\DVTE\libsrc\libpo\po_file_size.h
c:\test_case\DVTE\libsrc\libpo\libpo_local.h
c:\test_case\DVTE\libsrc\libpo\libpo.h
c:\test_case\DVTE\build\libsrc\libpo\po_const.h
c:\test_case\DVTE\build\libsrc\libpo\p_po.h
c:\test_case\DVTE\build\include\libinc\libpo.h
c:\test_case\DVTE\libsrc\libqueue\libqueue.h
c:\test_case\DVTE\libsrc\libqueue\libqueue_local.h
c:\test_case\DVTE\stow\include\libinc\libqueue.h
c:\test_case\DVTE\build\include\libinc\libqueue.h
c:\test_case\DVTE\build\include\libinc\libsema.h
c:\test_case\DVTE\libsrc\libsema\libsema.h
c:\test_case\DVTE\libsrc\libsema\libsema_local.h
c:\test_case\DVTE\stow\include\libinc\libsema.h
c:\test_case\DVTE\build\include\libinc\libshmif.h
c:\test_case\DVTE\libsrc\libshmif\libshmif.h
```

Figure G: Batch load files

Next, the interactions for each component were processed. Again each component was processed individually. Rational Rose Enterprise was used to check for dependencies, associations and other interactions among the classes within the model. Because of the way in which Rose processes data, this check was a two-pass operation. That is, once all the components were checked for dependencies, they check was run again. This was necessary because when the first component is reverse-engineered it takes each class and check for associations against all the other classes listed in the component (loaded from the batch file) and also checks for association against any other classes listed in the 'logical view' of Rational Rose Enterprise. Since none of the other components have yet to be reverse-engineered and have not placed the output in the 'logical view' of Rational Rose Enterprise, there are no other classes for the selected component to check against. In other words, when the first component is reverse-engineered, it is not aware of the classes within the other 60 components. Next, when the second component is reverse-engineered, that component is aware of the classes within one of the components, but is not aware of the other 59 components. The second time the components are reverse-engineered, the first component can now check against the other 60 components since their output is now in the 'logical view' section of Rational Rose Enterprise.

At this point, the reverse engineering and the software model of JSAF were complete. Rational Rose Enterprise provided for various graphical representations of the model. One such view, of the interactions of the GUI COMMAND SERVICE component, is shown in Figure H. The actual model elements are contained in the left window. The large window on the right is a viewing window and only displays the classes dragged into it by the user or by the system when the 'expand selected elements' function is called. The viewing plane shows some of the GUI classes, their associations, and class data structures in terms of UML

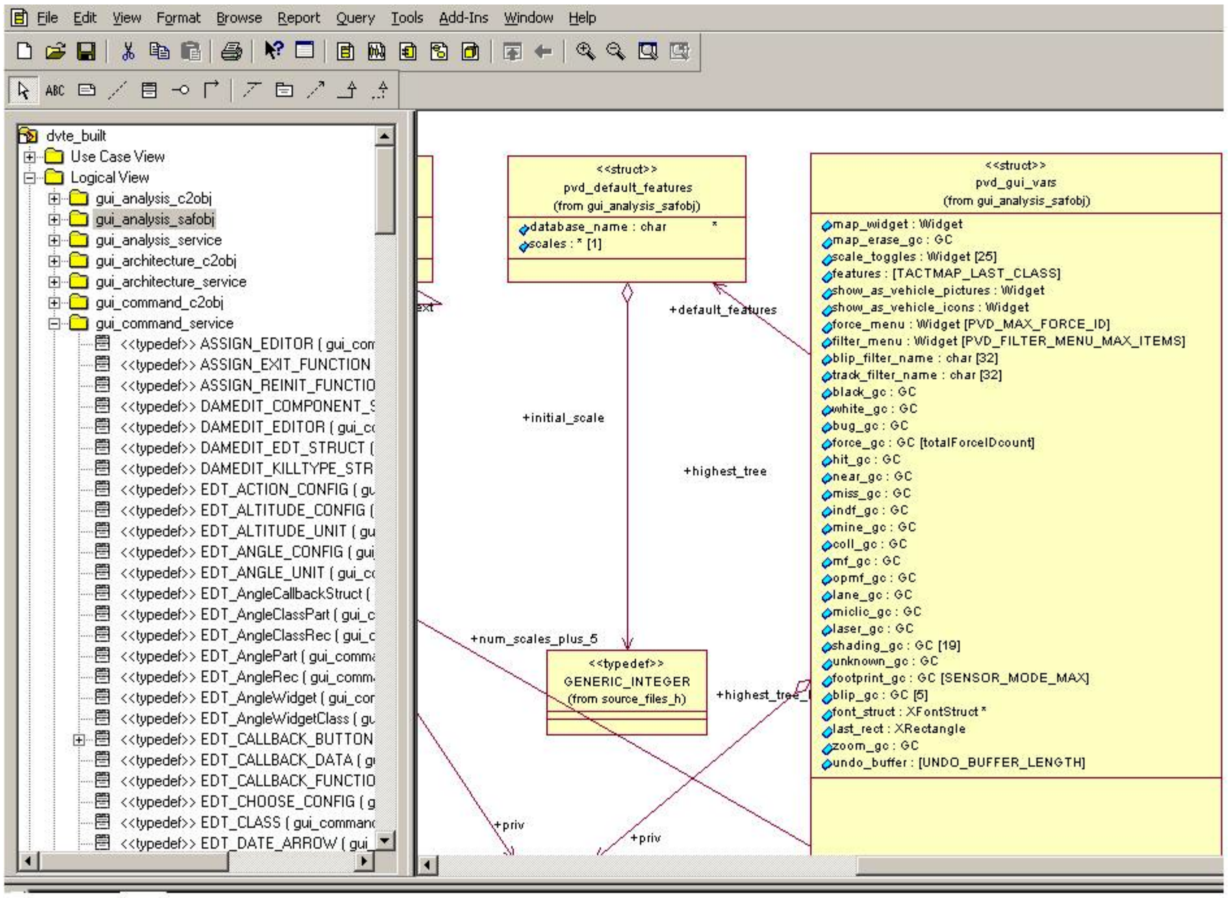

File Edit View Format Browse Report Query Tools Add-Ins Window Help
dvte_built
Use Case View
Logical View
gui_analysis_c2obj
gui_analysis_safobj
gui_analysis_service
gui_architecture_c2obj
gui_architecture_service
gui_command_c2obj
gui_command_service
<<typedef>> ASSIGN_EDITOR (gui_com
<<typedef>> ASSIGN_EXIT_FUNCTION
<<typedef>> ASSIGN_REINIT_FUNCTIO
<<typedef>> DAMEDIT_COMPONENT_S
<<typedef>> DAMEDIT_EDITOR (gui_co
<<typedef>> DAMEDIT_EDT_STRUCT (
<<typedef>> DAMEDIT_KILLTYPE_STR
<<typedef>> EDT_ACTION_CONFIG (gu
<<typedef>> EDT_ALTITUDE_CONFIG (
<<typedef>> EDT_ALTITUDE_UNIT (gu
<<typedef>> EDT_ANGLE_CONFIG (gui
<<typedef>> EDT_ANGLE_UNIT (gui_co
<<typedef>> EDT_AngleCallbackStruct (
<<typedef>> EDT_AngleClassPart (gui_c
<<typedef>> EDT_AngleClassRec (gui_c
<<typedef>> EDT_AnglePart (gui_comma
<<typedef>> EDT_AngleRec (gui_comm
<<typedef>> EDT_AngleWidget (gui_cor
<<typedef>> EDT_AngleWidgetClass (gu
<<typedef>> EDT_CALLBACK_BUTTON
<<typedef>> EDT_CALLBACK_DATA (gi
<<typedef>> EDT_CALLBACK_FUNCTIO
<<typedef>> EDT_CHOOSE_CONFIG (g
<<typedef>> EDT_CLASS (gui_commanc
<<typedef>> EDT_DATE_ARROW (gui
<<struct>>
pvd_default_features
(from gui_analysis_safobj)
database_name : char *
scales : * [1]
<<struct>>
pvd_gui_vars
(from gui_analysis_safobj)
map_widget : Widget
map_erase_gc : GC
scale_toggles : Widget [25]
features : [TACTMAP_LAST_CLASS]
show_as_vehicle_pictures : Widget
show_as_vehicle_icons : Widget
force_menu : Widget [PVD_MAX_FORCE_ID]
filter_menu : Widget [PVD_FILTER_MENU_MAX_ITEMS]
blip_filter_name : char [32]
track_filter_name : char [32]
black_gc : GC
white_gc : GC
bug_gc : GC
force_gc : GC [totalForceIDcount]
hit_gc : GC
near_gc : GC
miss_gc : GC
indf_gc : GC
mine_gc : GC
coll_gc : GC
mf_gc : GC
opmf_gc : GC
lane_gc : GC
miclic_gc : GC
laser_gc : GC
shading_gc : GC [19]
unknown_gc : GC
footprint_gc : GC [SENSOR_MODE_MAX]
blip_gc : GC [5]
font_struct : XFontStruct *
last_rect : XRectangle
zoom_gc : GC
undo_buffer : [UNDO_BUFFER_LENGTH]
+default_features
+initial_scale
+highest_tree
+num_scales_plus_5
<<typedef>>
GENERIC_INTEGER
(from source_files_h)
+highest_tree
+priv
+priv

Figure H: DVTE/JSAF Software Model

VI CASE STUDY

A utility case study was performed as a first check of the validity and usefulness of the model by comparing the method currently employed for a JSAF modification with the method possible using the JSAF model.

The modification selected for this study was a change in the GUI default preferences. These preferences include how much control the user has over unit creation (the ability to create friendly units only, enemy units only, all units, or no units), which units are visible to the user (friendly, enemy, neither), what map features the user sees initially (trees, buildings, contour lines, etc.), the values in which quantities are expressed (gallons as apposed to liters, miles instead of kilometers, feet instead of meters), and so on. The goal was to have a standard default setting for the DVTE program.

Initially the JSAF searches for a preferences file in the user's home directory. If that file does not exist, the JSAF loads the default system preferences. In order to minimize the changes to the DVTE version of JSAF and not eliminate any of the user flexibility it currently has, we did not want to change the default system preferences nor eliminate the ability of a local user to store his own preferences. Still, we needed the ability to start the system with a standard DVTE GUI default setting. We therefore needed to establish a DVTE preference file with a precedence between the user preference and standard system defaults.

To modify the preference file, one needs to understand how JSAF references this file and where it is stored. Without the aid of the software model, a first step for someone without extensive JSAF experience would be to conduct a key word search of the source file. Doing this identifies instances of the GUI objects that represented the initial values, but provides no indication of what other classes changed the values, or what other implications changing the system defaults within the code would have. It is, therefore, necessary to find the area of code where the user preferences are read. A search for that area of the source was performed and the code located. Additional code was added to this section to search the system folder for the DVTE preference file. Because of the location of the modification in the code, this search would occur immediately after the search for the user preference file in the home directory. With the addition of this code, the creation and distribution of a preference file in the system folder achieved the desired results.

Using the JSAF software model, the process was much different. First, a search through the documentation of the JSAF libraries was done for a generic library. This library, SAF_GUI, was organized as GUI COMMAND SERVICE. This organization was mapped to the GUI COMMAND SERVICE component in the JSAF software model. The associations with the SAF_GUI object were easily viewed with the Rational Rose tool. The result was a picture of the associations among the GUI objects that related to the SAF_GUI structure. Among these objects was a pvd_gui_vars object (see figure H). It contained the variables that needed to be modified as defaults. Also, a default_features pointer is the common object that communicates between the pvd_gui_vars object and the pvd_default_features object. This highlighted the specific area of code that needed to be modified to insert the search for the DVTE preferences file.

The final implemented solution is the same whether using the model or not. However, use of the model saved considerable time over not using it. In this instance, for an inexperienced JSAF programmer, the search time was reduced by more than a day. Addition time was saved in the testing of the modification. Because the model showed that the only association between pvd_gui_vars and the pvd_default_features object is the common parameter default_features object, the programmer could be confident that his changes were localized. The obviated the need for extensive testing. Lastly, use of the model required skill in using the associated software tool rather than familiarization with the JSAF code itself. This skill is much more transferable and likely to be of use on a wider range of projects

VII USE OF THE MODEL

With the completion of the JSAF software model, it is now available to improve the efficiency of the modifications we will make to JSAF in support of the DVTE program. The model will be used to support the analysis, design, and testing of required modifications. Analysis will be enhanced as there will be a clearer vision of what specific libraries, components, and objects need to be modified to implement any given requirement. Senior JSAF programmers rely on experience and familiarity with the code to identify where changes need to be made. For them, the model can be used to quickly validate their assessment. The model will also provide junior programmers without extensive JSAF experience the ability also reliably identify where changes need to be made. This will make a steep JSAF learning curve much shallower and greatly lessen the time required for a programmer to become confident in mapping JSAF requirements to specific libraries and objects.

The design phase of JSAF modifications will be improved since it will be known with greater certainty what the ripple effects of any modifications will be. Modifying JSAF currently has some trial and error flavor to it. Changes are made and the software is tested to see if any undesirable effects have been introduced. Use of the model will better highlight what components of the software will be affected by changes to given components and allow for a more complete design. It will also assist in ensuring that the design of modifications preserves the current system architecture as that architecture is captured by the model and designs will be in the context of that model.

The procedures for the most effective use of the model for analysis and design will need to evolve over time, as the model itself is better understood. Perhaps the greatest immediate benefit of the model, therefore, will be in the area of testing. Having a map of all the component interactions identifies what which of them may have been affected by a modification to another. Tests can therefore be constructed that ensure testing of the aspects that might have been affected. Time need not be spent testing aspects that the model shows will not have been affected. Regression testing of modifications should therefore be more efficient and effective.

Lastly, use of the model will assist in the documentation of modifications we have performed to JSAF. Being aware of all the interactions of any component of the code will assist in the bookkeeping of documenting how those interactions are affected by any given modification. Also, we are required as part of the DVTE program, to deliver documentation on the changes we have made to the baseline from which we started. This deliverable will be organized according to the library hierarchy as previously presented. Such an organization will make the modifications easier to understand for both the technical and non-technical recipients of the document.

VIII CONCLUSION

A software model of JSAF was recently completed and will be employed to assist in its modification. However, ultimate validation of the model will only come with time and use. The initial utility test of the model demonstrated that it does have value and it undoubtedly holds great promise. To fully exploit the model's potential will require adaptation of the current process used to modify JSAF and, to some extent, the culture of JSAF programmers. Once the skills of using the model are mastered, however, they are can almost immediately be applied to similar models of other systems. Familiarity with the use of such models, therefore, is a much more portable skill than is familiarity with a specific application.

The process used to produce the model of JSAF can be used for other legacy systems. Key to this process was the use of software tools. These tools have their limits, which may limit the utility of the reverse engineered software model. For example, much of JSAF's behavior is determined by parameter files, not class structures. The software tool we used to support the reverse engineering process currently does not have the capability to organize these files into a pictorial hierarchy to aid development. Therefore, the model will continue to evolve as the tools to support reverse engineering increase in capability. It is apparent that the software models produced through reverse engineering have intrinsic value. They should be capable of enhancing lifecycle management and reducing the lifecycle cost of complex legacy software systems. It is also apparent that more valuable models will be produced as the capability of the support tools improves.

REFERENCES

Eriksson, Hans-Erik & Penker, Magnus. (1998). UML Toolkit. New York, NY: Wiley Computer Publishing.

Sommerville, Ian. (1998). Software Engineering. Harlow, England. Addison Wesley.

ModSAF Programmer's Reference Manual Index.

ACKNOWLEDGEMENTS

The authors wish to thank the program manager of the DVTE, Mr. Donald Decker, and that program's technical director Dr. Michael Bailey. The modification to JSAF required for their program provided the motivation for the development of this model. The authors also wish to thank Mr. Stanley Grigsby of VisiTech, Ltd., Ms. Vicky Rowley of Sagacitech, and Mr. Gerald Stueve of VisiTech, Ltd., for their extensive guidance, counsel, and insight.