In Proceedings of the 5th Working Conference on Reverse Engineering, Oct. 1998, pp. 125-134, IEEE.

PACKRAT: A Software Reengineering Case Study

Gerald C. Gannod*
Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406

Gora Sudindranath, Mark E. Fagnani
and Betty H. C. Cheng†
Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226

E-mail: {gannod,sudindra,fagnanim,chengb}@cse.msu.edu

Abstract

Reengineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form. Many approaches for design recovery or reverse engineering have been suggested, most with some type of support tool. Since a project's time constraints may prohibit use of sophisticated techniques and/or tools due to the learning curves associated with the techniques and tools, methods that can be applied in lieu of complex support tools may be required. This paper describes a case study project involving the reengineering of a network application used by Texas Instruments to monitor network traffic in a local area network.

1 Introduction

Software maintenance is considered to be the most costly phase of the software lifecycle. It has been estimated that approximately 60 percent of the effort in software development is in software maintenance [1]. Given that software maintenance is an inevitable activity, successful software maintenance of large systems depends on several factors including the existence of accurate documentation of the system design. In some cases, software and documentation fail to be consistent in that the documentation, and subsequently the designs, are rarely updated to reflect modifications made to the system. In other cases the original system does not have any type of existing documentation and, as such, any rationale behind the decisions made during the implementation of the system are lost. In either case, the lack of a consistent design has many impacts on the effectiveness of any efforts to maintain and modify existing systems.

Software reverse engineering is the process of analyzing the components and component interrelationships of a software system in order to describe that system at a level of abstraction higher than that of the original system [2]. Reengineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [2]. Software reengineering is considered to be a better solution for handling legacy code when compared to re-developing software from the original requirements.

This paper presents a case study that illustrates how a combination of object-oriented (OO) and structured analysis and structured design (SA/SD) techniques can be used in tandem to reengineer an existing application that is used by Texas Instruments to analyze traffic on local area networks. The remainder of this paper is organized as follows. Section 2 presents background information regarding software maintenance and software reengineering. The application and domain are described in Section 3. The methods used for software reengineering are introduced in Section 4 and applied in Section 5. Section 6 describes related work, and Section 7 draws conclusions and suggests further investigations.

2 Background

This section gives background information in the area of software maintenance and semi-formal analysis and design techniques.

2.1 Software Maintenance

Figure 1 contains a graphical depiction of a process model for reverse engineering and reengineering [3]. The process model is captured by two sectioned triangles, where each section in a triangle represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. Entry into this reengineering process model

*This work was performed while this author was a Ph.D. student at Michigan State University.

†Please address all correspondences to this author.

begins with system *A*, where *Abstraction* (or reverse engineering) is performed to a level of detail appropriate to the task being performed. For instance, if a system is to be reengineered in response to efficiency constraints, then abstraction to the design level may be appropriate. This task can also be considered a process of *design recovery*. The next step is *Alteration*, where the system is configured into a new form at the same level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*.

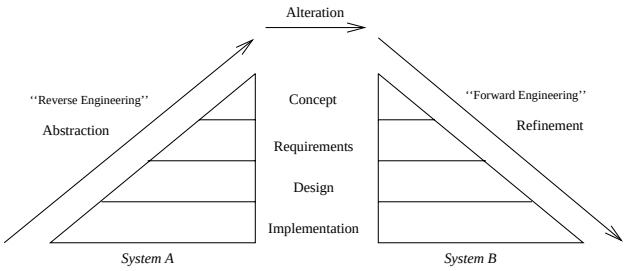

Figure 1: Reverse Engineering Process Model

This paper describes a case study investigation into the reengineering of a network application. The context of the investigations are applicable to all phases of the Byrne reengineering process model [3] and span all levels of abstraction.

2.2 Semi-Formal Analysis and Design

Object-oriented development techniques encompass analysis, design, and implementation. The *Object Modeling Technique* (OMT) [5] is commonly used in industry and academia. OMT comprises three complementary models, each of which are simple to use and understand. The *object model* describes the static, structural aspects of the system. The object model captures the objects of the system and the relationships between the objects. The *dynamic model* depicts the temporal and behavioral aspects of the system. Finally, the *functional model* describes the services provided by the system. Respectively, entity-relationship diagrams, state transition diagrams, and data flow diagrams are used to represent the object, dynamic, and functional models, and each model is only used to capture a specific perspective of the system. With recent work [6, 7], rigorous analysis of each of the models is possible, thus enabling consistency and completeness checks at the model level prior to the implementation phase.

3 Application

The PACKRAT tool monitors network traffic in order to provide debugging information to developers during the development of Ethernet drivers [8]. Currently, the PACKRAT tool is being used to investigate the development of internetworking protocol stacks [8]. In this section we describe the original PACKRAT system, and then the modifications that were required due to the changing needs of Texas Instruments.

3.1 Original System

PACKRAT is a network traffic monitoring application that was implemented for the Microsoft Windows 95 environment. The original system implementation provided a facility for communicating with the network interface driver via the Windows 95 Network Device Interface Specification (NDIS) interface. A graphical user interface (GUI) provided users with options for analyzing Medium Access Control (MAC) layer statistics, such as type and number of frames captured per second. Additional operations facilitated examination of decoded data captured from the network. No system documentation existed for the development of the original PACKRAT system. The source code for this system was approximately 5,500 lines of code and supported several users from the software development team at the Texas Instruments Network Business Unit.

3.2 New Requirements

Several modifications of the PACKRAT system were requested by the customers and consisted primarily of changes to two specific areas: user interface changes and network packet decoding enhancements [8]. The more significant modification requirements for PACKRAT involved support for the decoding of several Transmission Control Protocol/Internet Protocol (TCP/IP) application protocols such as Simple Network Management Protocol (SNMP) Version 1, Telnet, and Domain Name Service (DNS). The user interface changes dealt with visual presentation and ease of use issues, including the translation and display of numeric and character-based Internet Protocol (IP) addresses as well as Organizationally Unique Identifiers (OUIs). In this paper we focus primarily upon the reengineering of the network packet decoding functionality since the modifications, from a reengineering point of view, were more significant with respect to design recovery and system re-implementation.

In addition to the requirements described above, several constraints were placed on the reengineering project. These constraints were largely environmental; the original PACKRAT tool was implemented in the C programming language for Intel-based machines running the Windows 95 operating system. One of the constraints on the project was time-based; the project from original proposal to delivery was limited to fifteen weeks, where all five members had other commitments in addition to this project.

3.3 Reengineering Development Team

The reengineering development team consisted of five members, each of varying levels of software engineering experience. Although each member had experience with OMT, they had limited exposure to software maintenance and reengineering. As such, a short tutorial on the methods for reengineering was required. The methods, as described in Section 4, were developed with the intention of being lightweight in the amount of new techniques that would have to be learned by the developers. The primary goal of these techniques was to leverage the experience the team members had obtained from previous software analysis and design projects.

4 Method

In this section we present the methods used for design recovery and modification, and discuss the overall constraints on the project.

4.1 Process Overview

In the context of the reengineering process model presented in Section 2, the investigations described in this paper are depicted by the diagram shown in Figure 2. Specifically, this paper describes a case study that involved three distinct phases of investigation: 1) a design recovery, or abstraction step, 2) a design modification, or alteration step, and 3) a design implementation, or refinement step. In the diagram, the phases are depicted by the dashed arrows that originate from the implementation level to the design level in *System A*, from the design level of *System A* to the design level of *System B*, and from the design level of *System B* to the implementation level of *System B*, respectively.

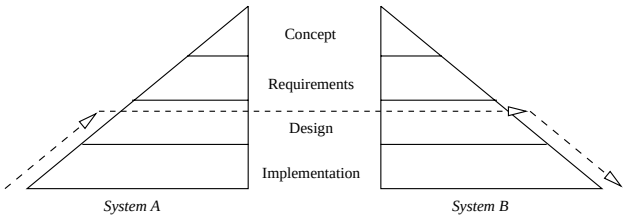

Figure 2: Project Context Diagram

4.2 Multi-level Approach to Design Recovery

Several techniques have been suggested for recovering designs from existing systems. These techniques range from formal approaches [9], to semi-formal functional abstraction [10], and structural abstraction [11]. The representations constructed by these techniques are often biased by the implementations, and as such, do not always correspond to existing high-level models.

The method used to perform the case study described in this paper is based on a combined top-down and bottom-up approach for design recovery. Recent investigations have suggested that this kind of approach is reasonable [11, 12].

For this project, the technique used for design recovery involved three distinct steps. First, a high-level concept model for the system was developed and refined based on customer interviews and empirical investigations involving the existing system. Second, a low-level source model, in the form of a call graph, was constructed. Finally, an iterative top-down and bottom-up abstraction and encapsulation step was performed. In this step, the low-level models were abstracted into higher-level entities and then verified against refinements of the high-level models that were constructed in the first step.

Many reengineering approaches have focused on the recovery of objects from existing code [13, 14]. In order to facilitate the recovery of an object-oriented model from the existing system, the use of the OMT method was combined with the SA/SD technique. For the high-level models of the system, OMT was used as the modeling notation. At the very low-levels of design, SA/SD was used as the modeling notation. The integrating mechanism between OMT and SA/SD is the data-flow diagram, a model used by both techniques. As such, the developers found this approach to be sufficient and effective.

Although many tools exist that support the activities described above, including Rigi [15], and the Reflexion Model (RM)-Tool [11], it was determined that due to time constraints and to avoid the learning curve associated with using a new support tool, that a simple source code browser would be used to derive the source models, and that the encapsulation of models into higher-level entities would be performed manually. In retrospect, as described in Section 5.4, the developers found that the results obtained due to this decision were the source of some confusion about the functionality represented by the recovered design.

4.3 Design Modification

The primary goal of the method described in Section 4.2 is to provide developers with a detailed understanding of the existing system. With this understanding, the developers could then modify the existing design in such a way that would facilitate the incorporation of the requested modifications to the system.

The next step in the process was to change the design of the current system by analyzing the modification requirements requested by the project customer. These requirements allowed the developers to identify the context for the requested modifications and to focus their efforts on impacted subsystems. Specifically, this step involved three distinct phases. First, a requirements analysis was performed that addressed the new requirements for the sys-

tem. Second, an impact analysis was performed on the recovered design to identify the parts of the system that were to be modified based on the new requirements. Finally, the recovered design was modified in order to incorporate the new requirements into the functionality of the existing system. For each of these phases, analysis and modeling were performed in the context of the OMT notation.

4.4 System Modification

Modification of the recovered designs to incorporate new requirements is referred to as the *alteration* step [3]. After a design has been altered to incorporate new requirements or constraints, traditional software development techniques can be used to develop the code that satisfies the modified design. An implicit constraint in the development phase of the project was to reuse the existing source code. In addition, the system was to target the same operating system (Windows 95) and source language (C).

5 Case Study Details

In this section we summarize the application of the methods described in Section 4 as they were applied to the PACKRAT system.

5.1 Design Recovery

As mentioned previously, the design recovery phase of PACKRAT consisted of three distinct steps: creation of a high-level concept model, construction of a low-level source code model, and finally, a top-down, bottom-up refinement of the resulting models until a medium-level model was derived. The first two steps were performed concurrently; the iterative third step was completed after both high-level and low-level models were constructed.

5.1.1 High-Level Models

The high-level concept models for the PACKRAT system were constructed using two sources of information. The first source of information was a result of data collection from an empirical analysis of the existing system. This consisted of operating PACKRAT in a manner similar to that of the typical user population and observing the output of the system. For instance, a typical scenario would consist of selecting a network device, capturing packets, and then viewing decoded information (output). From observing the various types of output of the system, the developers were able to abstract the output into classes that shared common characteristics (i.e., output in the same window, output at the same time, etc.).

The second source of information that was used to construct the high-level models was data gathered from interviews with the developers of the original software. However, in gathering design information from the original developer, a conscious effort was made to recognize implementation bias.

Figures 3 and 4 depict the high-level object and functional models of the PACKRAT system, respectively. Figure 3 is the model of the various objects extracted from the system based on empirical observations and developer interviews. The object model shows classes as labeled rectangles and relationships between classes as lines. In the notation, the diamond symbol at one end of a line is used to indicate an aggregation relationship. As indicated by Figure 3, PACKRAT is an aggregation of global data and various GUI components (i.e., Frame Window, View Window, etc.).

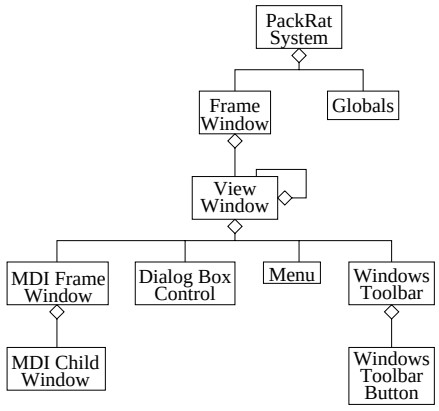

Figure 3: High-level Object Model

A functional model, or data flow diagram, uses circles to denote process, labeled arcs to denote data flow, labeled parallel lines to denote data stores, and rectangles to indicate external entities or actors. Figure 4 is a data flow diagram that shows three high-level PACKRAT modules: *WinMain*, *Core Process*, and *Core Process Management*. *WinMain* is the primary driver for PACKRAT and is responsible for the initialization of GUI components and acting as a communication mechanism between the GUI components of the application. *Core Process* is another major component of the system that handles all output interaction with the user, performs all data manipulation associated with processing packets, and interacts with the network device. *Core Process Management* is primarily concerned with the initialization and termination of the system.

5.1.2 Low-Level Model

The low-level source model, shown in Figure 5, depicts the call graph of the original PACKRAT system. In the diagram, a vertex represents a function, with the lines between vertices indicating a calling relationship. Specifically, a vertex *B* connected by a line to another vertex *A*, where *B* lies below *A* in the graph, is *called-by A*. Using a source browser (i.e., a program that facilitates hypertext source code traversal, where procedure calls serve as hypertext links), the developers constructed a call graph that repre-

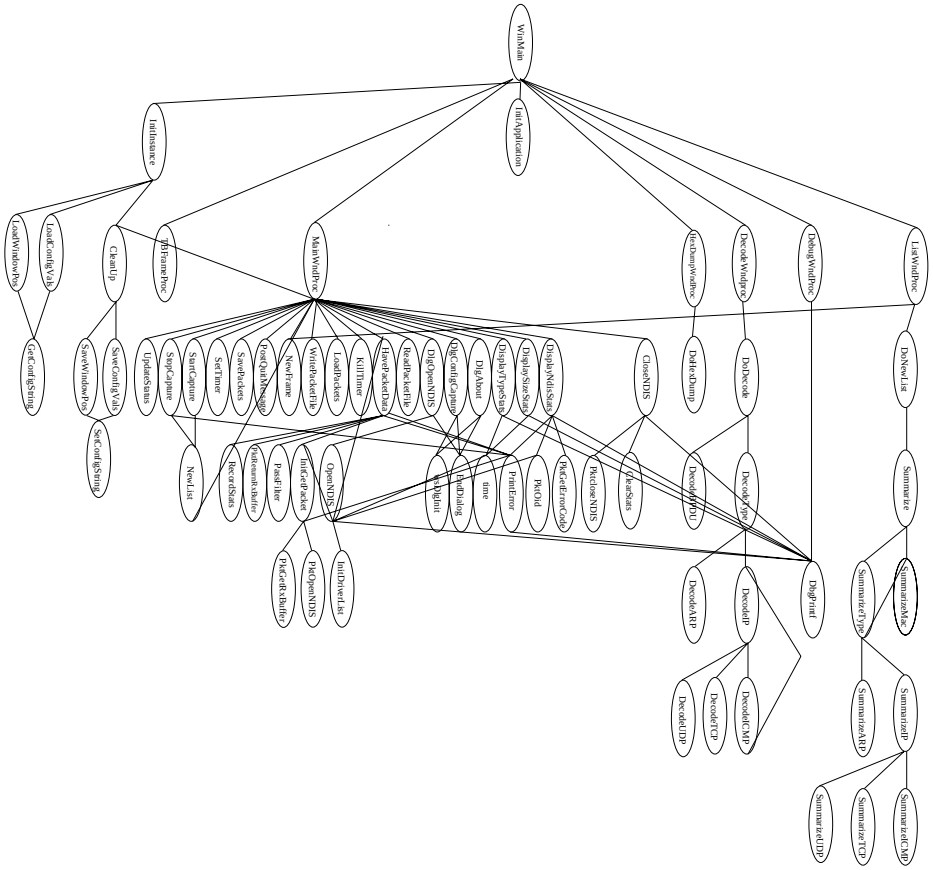

Figure 5: System Call Graph

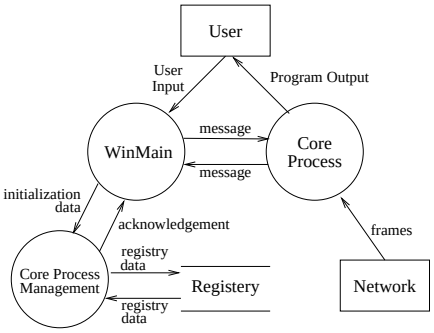

Figure 4: High-level Functional Model

sents the entire PACKRAT system. The call graph served as the entry point for the low-level reverse engineering and design recovery process.

The root module in the hierarchy of Figure 5 is **WinMain**, which maps directly to the high-level concept model (in Figure 4). Below WinMain, there exists a number of functions devoted to handling events for a particular window in the GUI. Among these are **MainWinProc** (handles events for the main window), **ListWinProc** (handles events for the List Window), **DebugWinProc** (handles events for the Status Window), **HexDumpWinProc** (handles events for the Hex Dump Window), and **DecodeWinProc** (handles events for the Decode Window). Most of the data processing for PACKRAT occurs in functions called by these event handlers, as can be seen in Figure 5.

After analyzing the various event handlers, it was determined that two particular routines, the **ListWinProc** and **DecodeWinProc**, had redundant behavior. These two event handlers were responsible for decoding packets that were related to the packets listed in separate windows. The **ListWinProc** performed decoding on packets for a scrollable list used for selecting packets. Information for each entry in the list included packet size, source/destination IP address, and an index uniquely identifying that packet. The **DecodeWinProc** performed more extensive decoding on packets that were selected from a different scrollable list. The decoded information had a dependency on the protocol of the selected packet, but there were fields that were similar to fields decoded by the **ListWinProc** handler such

as frame type, source/destination IP address, size of packet, and transport layer protocol, among others.

Global data was a particularly prominent feature of the PACKRAT system and consisted of various items that were critical to the operation of PACKRAT. Among such data were the capture buffer or Frame List that was used to store captured packets, various packet statistics, and a pointer to the most recently captured frame. Figure 6 shows an object model of a subset of the global data items. The object modeling of the global data provided a level of abstraction that facilitated the use of global data as "classes", thus providing a disciplined policy for access and modification of the global data. This model became of particular interest during the modification phase, as described in Section 5.2.1.

5.1.3 Medium-Level Model

The next step was to construct a *medium-level* design model from the high-level concept and low-level source models. The approach involved two concurrent tasks: abstraction of the low-level source model of Figure 5 into higher-level entities, and refinement of the high-level concept model in Figure 4 into lower-level entities. Both of these tasks were performed iteratively until they resulted in a single, medium-level model. The result of these activities is shown in Figure 7.

There were several levels of abstraction between the medium-level and low-level models. Due to space constraints, we focus on only one intermediate level of abstraction. Figure 8 shows the relationship between the medium-level (Figure 7) and low-level models (Figure 5), where the dotted lines enclosing groups of functions were abstracted into corresponding modules of the medium-level model (in Figure 7).

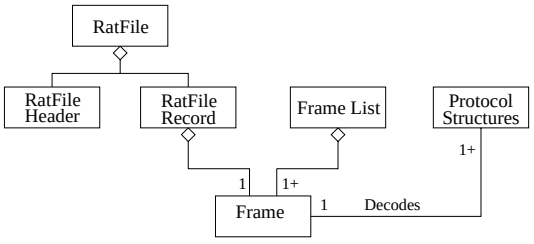

Figure 6: Subset of Global Data Object Model

Several features of the call graph in Figure 5 facilitated the abstraction of the various procedures into the modules shown in Figure 8. First, naming conventions provided the initial cues about the partitioning of the system into initialization routines and shutdown routines. As such, the **Cleanup** function and all functions that it directly or indirectly calls were abstracted into the *Shutdown module*. Similarly, **InitApplication**, **InitInstance**, and all

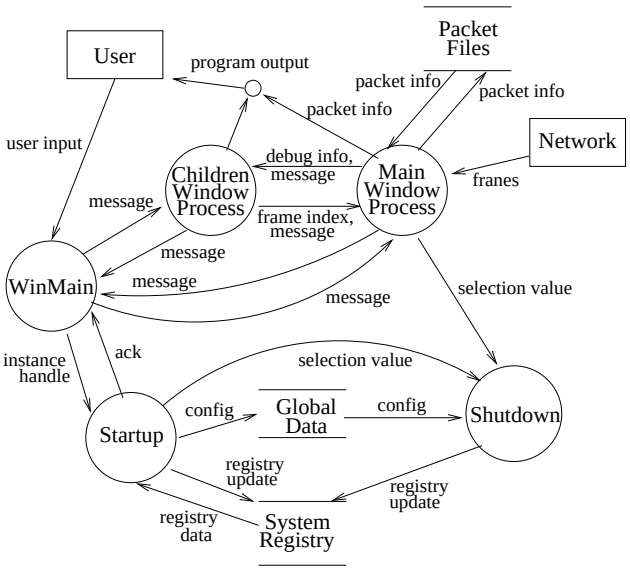

Figure 7: Medium Level Data Flow Diagram

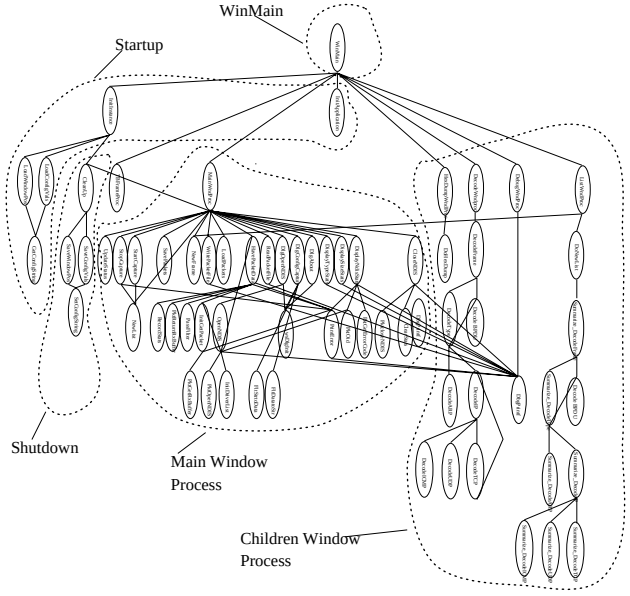

Figure 8: Call graph abstraction

functions called by InitInstance were abstracted into the *Startup* module.

Another cue provided by the call graph was a result of comparing the in-degree and out-degree of a vertex. If there exist vertices with a large difference in out-degree and in-degree, then they can be considered to be candidates for module drivers. Thus, the vertex (procedure) and all of its calls can be abstracted into higher-level modules. Based on this criterion, **MainWinProc** and all of the func-

tions it calls were abstracted into the *Main Window Process* module of the medium-level model.

The final cue that was used to abstract procedures into higher-level modules was provided by the information obtained from creating the high-level concept models. Using this information, all child window event handlers such as **ListWinProc**, **DecodeWinProc**, **DebugWinProc**, and **HexDumpWinProc** were abstracted into the *Children Window Process* module of the medium-level model.

The recovery of a design using reverse engineering techniques can facilitate many software maintenance activities, including the modification of systems to incorporate new requirements. Using the techniques described in Section 4.2, the development team found that the recovered design formed a solid basis for the next step of the reengineering process, the design modification phase.

5.2 Design Modification

This section describes the modifications needed to add the application layer protocols to the decode ability of the PACKRAT system. The technique used to introduce modifications into the system was based on a two-step process. First, the recovered models described in Section 5.1 were analyzed to determine what modifications, if any, could be made in order to facilitate the extension of system functionality. In essence, the models were analyzed in order to determine if any system *restructuring* could be used to increase system extensibility. Second, the recovered models were analyzed to determine the impact of changes due to the new decoding requirements. The impact analysis involved determining what parts of high-level, medium-level, and low-level models that needed modification in order to provide the newly requested functionality.

The proposed modifications, specifically, the addition of routines to support the SNMP, DNS, and Telnet protocols, presented two major design challenges. First, a bridge between the old and new systems that facilitated the use of the old system as a "black box" was desired. It was determined that the existence of such a bridge would facilitate the separation of new functionality from previous functionality both in concept and in the implementation. The second design challenge was to find a mechanism for incorporating several new application layer protocols to the decoding ability of PACKRAT that was extensible and minimized the modifications to the structure of the system. The remainder of this section discusses each of these challenges in detail.

5.2.1 Addressing the Design Challenges

The PACKRAT system uses a three step process for translating and displaying network frame information to users. First, frame data is captured from the network via the NDIS interface and placed into a global list, known as the **Frame List**. Next, the frame is decoded according to the protocol embedded in the frame. Finally, the decoded frame data is displayed to the user.

As shown in Figure 6, the original PACKRAT system relied heavily on the use of global variables and direct access to a buffer for storing raw (undecoded) frames, a technique for encapsulating the old system was needed in order to avoid affecting previously implemented functionality. As such, an object-oriented wrapper was created in the form of a list that encapsulated frame-related information.

The wrapper, called the Decode List, is essentially an ordered aggregation of Decode Information nodes, each of which contains all of the data (decoded and otherwise) pertaining to a particular captured frame. The Decode List, shown in Figure 9, would is instantiated whenever a new set of frames is loaded into the Frame List class of the old system. Each node is an individual object whose structure is based on the TCP/IP protocol stack. That is, each node is an aggregate of structures that correspond to the layers of abstraction present in the TCP/IP protocol stack. The instances of the structures were designed as subclasses and thus the exact composition of each node is dependent on the protocol sequence used by the frame to which it corresponds. The structures that correspond to the Data Link, Network, and Transport layers contain all of the decoded information relating to those layers. The Application Information class encapsulates information about the application layer and contains general information about the application portion of a frame. This information is used by the application decode modules in their individual decode processes. In Figure 9, the Frame class corresponds to the physical layer and has a direct relationship to the frames in the Frame List in Figure 6. Specifically, this class serves as a high level interface to the Frame List class of Figure 6 and makes the Frame List an implicit part of the Decode List shown in Figure 9. In this respect the Decode List replaces the original function of the Frame List, because it mediates any interaction with the Frame List once the Decode List has been initialized.

Another design challenge in the system was to support system extensibility so that new application protocols could be decoded by the PACKRAT system. By constructing the Decode List class as described above, all future extensions could access data in a uniform manner, thus separating the functionality of the old system, the current modifications, and all future modifications.

5.2.2 Adding Application Layer Functionality

Based on the requirements, the addition of application layer decoding was of primary importance. Figure 10 details the upper levels of the application decode modules and their place within the *Decode Module*. The nature of the interaction between the system and the decode modules

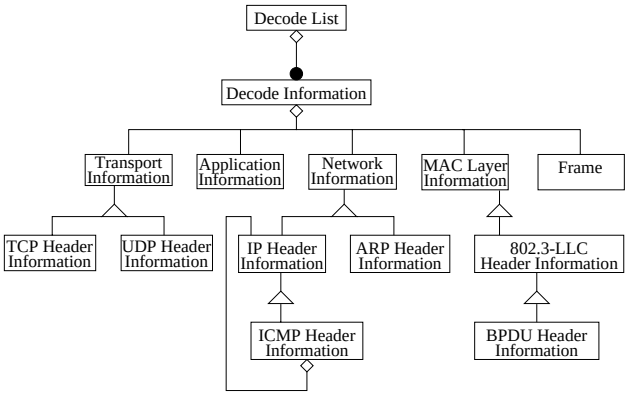

Figure 9: Decode List Object Model

is based on the Decode List and the output format used by the display windows. A particular decode submodule is passed a node in the Decode List that corresponds to the frame currently being displayed in the display windows. Once a frame has been decoded, the submodule returns a text buffer that contains a formatted set of decode information. The text buffers from each decode module are then concatenated and displayed.

The *Decode Module* of the new system contains four separate decode modules. Each of these decode modules are modified versions of the original decode module and correspond to three application layer protocols: *Telnet*, *DNS*, and *SNMP*. The fourth module, *Determine Application Protocol*, is used to identify the application frame type, like DNS, SNMP, and Telnet, before the decoding process begins.

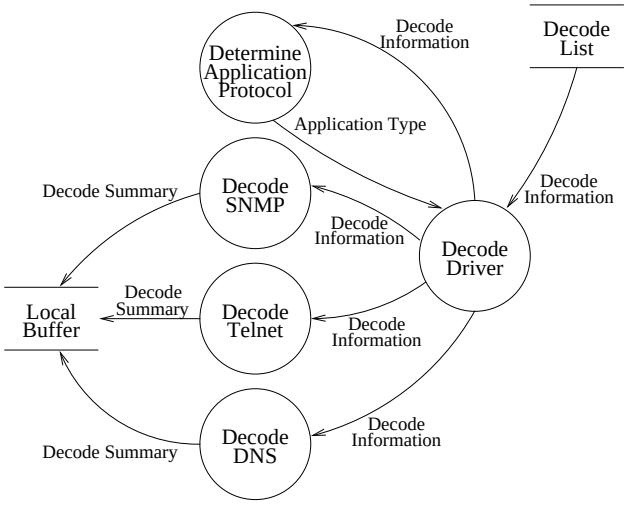

Figure 10: Application Decode Data Flow Diagram

Due to space constraints, we focus on the design of the *SNMP* module. Descriptions of the remaining modules are contained in a design document constructed by the development team [16]. The SNMP protocol is defined in such a way that each SNMP message is an ordered collection of Type/Length/Value (TLV) encoded fields. While the meaning and order of each field is dependent upon the message type, all fields are TLV encoded. One of the challenges this presented is that by definition all TLV encodings are of variable length. This means that each variable length message is made up a variable number of variable length fields.

Figure 11 shows the object model of the SNMP Encoding class. The SNMP Encoding Object is an aggregation of one or more TLV Encoding Objects. Each TLV Value subclass represents the different formats of an SNMP Encoding object, thus allowing the software to facilitate the issue of variable length TLV encodings. The advantages of this design include encapsulation, extensibility, and strict adherence to the SNMP standards [17]. The design currently handles only SNMP version one packets, but could be extended to handle version two of future versions with minor modifications to the model.

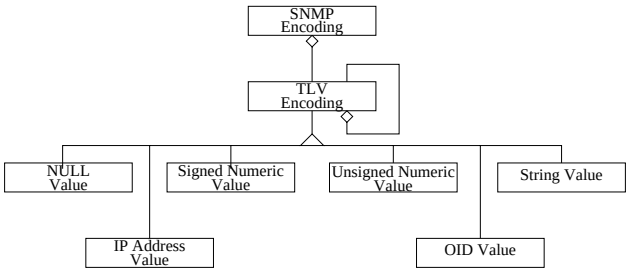

Figure 11: SNMP Object Model

5.3 Implementation

The implementation phase for the changes to the system, including testing, occurred over a period of approximately four weeks. The primary challenges in introducing the required modifications were related specifically to the language and environment constraints placed on the project. For instance, the language constraint placed upon the development team was that the system be built using the C programming language. As such, since the development team chose to perform an object-oriented analysis and design of the modifications, an object-based implementation using the C programming language was required.

Another explicit constraint placed on the project was that the system be developed for the Windows 95 environment, which was natural since the original PACKRAT system was a Windows 95 application. This presented a particularly significant challenge to the development team since their primary experiences had been limited to a Unix

environment.

The final system consisted of approximately 11,500 lines of code, a size that doubles that of the original PACKRAT. This fact is attributed to the complexity and number of modifications required by the customers. The project was completed well within the schedule for project deliverables and received positive feedback from the customers at Texas Instruments.

5.4 Lessons Learned

Several lessons were learned by the reengineering developers of the PACKRAT system. This section highlights several of those lessons.

- The adoption of and adherence to a systematic process for performing reengineering yielded many benefits and was the primary factor in the success of the project. Specifically,
  - The design recovery step facilitated gaining an understanding of how the required modifications fit into the context of the original system.
  - The design recovery step was useful for gaining an in-depth understanding of the PACKRAT system in a short amount of time. The recovered design artifacts such as call graphs, object models, and data flow diagrams, provided a mechanism for discovering information about system structure and provided context for the individual source code procedures.
  - The design recovery phase made the code implementation step much more straightforward and less prone to errors than trying to re-implement the system without understanding the design first.
- A reverse engineering technique that combines the use of an OO design methodology and a SA/SD design methodology can facilitate the transition of a system from a procedural style to an object-oriented style.
- Use of systematic reengineering methods facilitated the completion of the overall project well within the schedule for project deliverables.
- The lack of detailed documentation for the original system posed significant challenges in the design recovery step, particularly, given the time constraints.
- Overall, the lack of software tools to support the reengineering process made the process more difficult, time-consuming, and error-prone. In particular, source code analysis tools for generating call graphs would have expedited the design recovery stage. To address this concern, we are in the process of developing a suite of tools that support informal and formal reverse engineering techniques.
- The development team emphasized the need for a systematic process to recover and document the design prior to re-implementation. The development team found that the design recovery process and the by-products significantly reduced the amount of time it took to re-implement and test the system.
- Software reengineering is a very attractive method for making enhancements to an existing system. However, the learning curve for software reengineering is too high to perform and learn at the same. Users of the software reengineering process must be knowledgeable about the techniques and tools before applying them to a software system.

6 Related Work

Several reengineering efforts have been reported in the literature. Neighbors [18] described an approach for constructing reusable components from large existing systems by identifying the tightly coupled subsystems contained in the system. The approach is primarily based on the use of informal and experimental methods and the main objective was to make observations about the experiences encountered with a few large systems. Lewis and McConnell [19] describe a reengineering process and its application to a real-time embedded system. The seven step process describes high-level milestones that range from reverse engineering and reengineering to retargeting and final test. The reporting of our case study investigation differs from each of these in that we describe the reverse engineering and reengineering activity in detail while Neighbors describes observations that resulted from performing a reengineering activity, and Lewis and McConnell describe a high-level process that includes reverse engineering and reengineering as a step, but do not emphasize the experiences of performing the step.

7 Conclusion and Future Investigations

The ability to analyze software project requirements in order to identify a process model for the subsequent development of the software is an invaluable tool for any software developer. For traditional software development, identification of the appropriate process is straightforward as there are several well-documented development methodologies available. For software reengineering, the identification of a process is more difficult since the suite of available reengineering methodologies has not been standardized. However, there have been investigations into the generation of project-specific software reengineering process models [20].

The value of case studies is in their ability to provide a data point for assessing currently available techniques and for providing motivation for new techniques and tools. In this paper, several interesting and validating lessons were

learned: (1) a systematic process for reengineering is a fruitful activity, (2) a combined OO and SA/SD approach to reverse engineering can facilitate the transition of a system from a procedural style to an object-oriented style, and (3) the lack of tools to support reengineering can be inhibiting. In order to address the tools issue, we are investigating the creation of a maintenance workbench that incorporates the use of several classes of support tools including those that produce and verify structural abstractions [15] and functional abstractions [10, 21].

Software engineering courses in a typical undergraduate program focus mainly upon the construction component of the software development lifecycle. In order to better prepare the undergraduates with the inevitable software maintenance activity, we are investigating the creation of software maintenance courses that emphasize the use of reengineering techniques.

Acknowledgements. The authors would like to thank the following people for their participation in this project: John Edwards, Craig Neorr, and Jon Warren from Michigan State University, and Lynn M. Kubinec from Texas Instruments.

References

[1] M. Hanna, "Maintenance burden begging for a remedy," *Datamation*, pp. 53–63, April 1993.

[2] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, pp. 13–17, January 1990.

[3] E. J. Byrne, "A Conceptual Foundation for Software Reengineering," in *Proceedings for the Conference on Software Maintenance*, pp. 226–235, IEEE, 1992.

[4] E. Yourdon and L. Constantine, *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[6] R. H. Bourdeau and B. H. C. Cheng, "A formal semantics of object models," *IEEE Trans. on Software Engineering*, vol. 21, pp. 799–821, October 1995.

[7] Y. Wang and B. H. C. Cheng, "Formalizing and integrating the functional model within omt," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[8] L. M. Kubinec, *PackRat: A Windows-based Ethernet Sniffer*. Texas Instruments Network Business Unit, January 1998.

[9] G. C. Gannod and B. H. C. Cheng, "Strongest Postcondition as the Formal Basis for Reverse Engineering," *Journal of Automated Software Engineering*, vol. 3, pp. 139–164, June 1996.

[10] A. Quilici, "A Memory-Based Approach to Recognizing Program Plans," *Communications of the ACM*, vol. 37, pp. 84–93, May 1994.

[11] G. C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study," *Computer*, vol. 30, pp. 29–36, August 1997.

[12] B. H. C. Cheng and B. Auernheimer, "Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software," in *Proceedings of the 18th Annual NASA Software Engineering Workshop*, pp. 274–282, NASA, December 1993.

[13] G. C. Gannod and B. H. C. Cheng, "A Two Phase Approach to Reverse Engineering Using Formal Methods," *Lecture Notes in Computer Science: Formal Methods in Programming and Their Applications*, vol. 735, pp. 335–348, July 1993.

[14] R. M. Ogando, S. S. Liu, N. Wilde, and S. S. Yau, "An object finder for program structure understanding in software maintenance," *Software Maintenance: Research and Practice*, vol. 6, pp. 261–283, 1994.

[15] S. R. Tilley, K. Wong, M.-A. Storey, and H. A. Müller, "Programmable Reverse Engineering," *The International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, pp. 501–520, 1994.

[16] J. Edwards, M. Fagnani, C. Neorr, G. Sudindranath, and J. Warren, PACKRAT *System Design Document*. Michigan State University, March 1998. Available at http://web.cps.msu.edu/~cps478/S98/Projects/PackRat/web.

[17] M. Miller, *Managing Internetworks with SNMP*. M & T Books, 1993.

[18] J. M. Neighbors, "Finding reusable components in large systems," in *Proceedings of the Third IEEE Working Conference on Reverse Engineering*, pp. 2–10, IEEE, November 1996.

[19] B. Lewis and D. J. McConnell, "Reengineering Real-Time Embedded Software onto a Parallel Processing Platform," in *Proceedings of the Third Working Conference on Reverse Engineering*, pp. 11–19, November 1996.

[20] E. J. Byrne, "Generating project-specific reengineering process models," in *Proceedings of the 6th Annual DoD Software Technology Conference*, April 1994.

[21] G. C. Gannod and B. H. C. Cheng, "A Formal Automated Approach for Reverse Engineering Programs with Pointers," in *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference*, pp. 219–226, IEEE, 1997.