A Principled Analysis of THS(X) 3.2.0

William R. Ommert
Mark L. Ross
Catherine O. Lyons
Scott M. Thayer

Submitted : March 31, 2004
By: The Robotics Institute, Carnegie Mellon University

Abstract

This report presents the results of an analysis of the Target Handoff System (eXperimental) (THS(X)) software package. THS(X) is the software component of a handheld digital tool to enable a Forward Observer to get a comprehensive view of the tactical situation, identify targets, and communicate with various support agencies and platforms. The analysis of this software focused on five primary areas: Reliability, Extensibility, Maintainability, Testability, and Portability. Each of these areas is defined and discussed in Section One of the report. Section One also contains a discussion of several other key concepts in use in Software Engineering today. Section Two contains the core findings of the analysis. The analysis was not intended to be comprehensive, thus this section limits itself to a treatment of the four major fatal flaws discovered over the course of the analysis. Section Three summarizes the analysis and provides recommendations. The final section of the document contains the Appendices. The five appendices present the following information: anecdotal information from our experiences trying to install THS(X) 3.2.0, anecdotal information from our experiences trying to run THS(X) 3.2.0, a list of point defects found during the review of the code that comprises THS(X) 3.2.0, a detailed explanation of ATL COM, and lastly a sample coding standard.

THS(X) 3.2.0

CMU-RI-TR-04-43

Carnegie Mellon

THE ROBOTICS INSTITUTE

Carnegie Mellon

A Principled Analysis of THS(X) 3.2.0

Submitted : March 31, 2004
By: The Robotics Institute, Carnegie Mellon University

AUTHORS
William R. Ommert, *Senior Research Programmer*
Mark L. Ross, *Software Engineer*
Catherine O. Lyons, *Research Programmer*
Scott M. Thayer, Ph.D *Systems Scientist*

THS(X) 3.2.0

THE ROBOTICS INSTITUTE

Executive Summary

We strongly recommend that no effort be expended attempting to convert the existing THS(X) implementation to Block 3 THS. The present design and implementation of THS(X) is not reliable, extensible, maintainable, testable, or portable.

This recommendation is the result of a principled analysis of THS(X) version 3.2.0. The industry standards which inform the analysis are described in the first section of this document, *Foundations of Good Software*.

In Section 2, *Analysis of THS(X) 3.2.0*, we detail four fatal pervasive issues relating to THS(X). The materials analyzed included: executables, source codes, and documentation; all of which were produced by Stauder Technologies, Inc.

➦ **Issue #1:** The design of the core object model used in THS(X) is not extensible, portable, or maintainable.
The architectural defects in this system are so inextricably intertwined in the core of the system that simple fixes are not an option. A new design concept, based on the selective use of composition instead of the exclusive use of inheritance, is required for this system.

➦ **Issue #2:** The pervasive use of **friend** in the core object model of THS(X) shatters encapsulation and exponentially increases unit testing.
THS(X) is not testable because of the combinatorial expansion of time required to do testing. It is not extensible because of poorly structured inheritance. It is not maintainable because the design of the system depends heavily upon coupling between classes and poor inheritance hierarchies. The result is an extremely complex system. To achieve the requisite level of understanding would require an unreasonable investment in time on the part of the system's maintainer.

➦ **Issue #3:** The use of dynamic linking in THS(X) results in reduced portability and reliability.
A high level decision was made to use dynamic linking in THS(X). There are times when dynamic linking is unavoidable. The interface to C2PC requires dynamic linking. Applications use dynamic linking in order to interact with the core services provided by the operating system. Dynamic linking should be avoided whenever possible, but most especially in fires applications, because dynamic linking introduces into the system an unacceptably high risk of failure.A runtime failure of a dynamically linked system will manifest itself as unexpected application death, or unexpected application behavior. Both of these things are unacceptable in a deployed safety critical system. The problems introduced by dynamic linking are completely avoidable through the use of static linking.

➦ **Issue #4:** The documentation for the THS(X) system is inadequate and lacks fidelity.
The documentation for THS(X) is internally inconsistent and lacks rigor. The requirements documents encode design and the design documents contradict each other. Timely, relevant, clear, and accurate documentation is required when designing, implementing, testing, and supporting a software system. The documentation for THS(X) is none of these things.

Point Defects:

In addition to the four pervasive defects detailed in Section 2, a large number of point defects were identified during the evaluation. These point defects are cataloged in Appendix C. Each of these defects was evaluated for its impact on the THS(X) system. While these point defects are quite severe, many of them can be fixed. It is not possible to fix the systemic defects of THS(X) by addressing any number of individual point defects; the problems with THS(X) are pervasive and flow from poor design decisions.

In Summary:

Through review of THS(X) and its associated documentation, we conclude that the design and implementation of THS(X) are fatally flawed. THS(X) is not reliable, testable, extensible, maintainable, or portable. If deployed today, THS(X) will fail in operational environments. We strongly recommend a wholesale replacement of THS(X).

Forward

A Principled Analysis of THS(X) is comprised of five sections. Readers may chose to read this report in the order in which is appears, or the reader may wish to focus on a particular section. In order to facilitate the goals of the reader, we provide this brief overview.

The section which follows immediately is entitled *Foundations of Good Software*. Orthogonality, the concept that motivates most of the current thinking in the software industry today, leads this section. Orthogonality is followed by brief discussions on the five pillars which define a high quality software product: Reliability, Extensibility, Portability, Testability, and Maintainability.

Good Practices is a primer on the techniques and concepts that professional systems architects and development teams use to achieve high quality software. We discuss Object Oriented design, and the three principle concepts of Object Oriented design: Encapsulation, Abstraction, and Inheritance. We also discuss a well-known software pattern, Model-View-Controller. This section begins on page 20.

Our analysis of THS(X) relies on an understanding of these terms and concepts. Clearly, the THS(X) software was based upon
the Object Oriented design paradigm. The THS (X) design and architecture documents suggest that the developers attempted to employ the Model-View-Controller design pattern.

Analysis of THS(X) 3.2.0 begins on page 29. We discuss the fatal flaws of THS(X). This is an in-depth technical discussion detailing the violations of the Object Oriented paradigm and how those violations impact Reliability, Extensibility, Portability, Testability, and Maintainability.

Conclusions and Recommendations begins on page 57. These pages sum up the analysis, and provide our recommendations for moving forward.

The Appendices start on page 71. Our experience with the installation process is detailed. A copy of the CMU Coding Standard
is included. A list of some of the point defects found in the software follows. We spent some small amount of time running THS(X). Our findings appear in Appendix D. THS(X) makes extensive use of COM. We provide a comprehensive discussion
on the workings of ATL COM in Appendix E.

Table of Contents

Foundations of Good Software

Orthogonality

While there is no single, authoritative definition of correctness for application structure, or for a "good application," there is a set of concepts and practices that are widely accepted as best.These concepts and best practices are not in and of themselves enough to guarantee a "good" application. However, their appropriate use will strengthen any application.

The common thread that runs through many of these concepts and practices is the concept of orthogonality. From this concept we derive many of the core principles of good software design. The term orthogonality has its roots in Mathematics. In Statistics, a branch of applied Mathematics, two pieces of data are orthogonal when they are statistically independent. In Geometry, two lines that are perpendicular to one another are said to be orthogonal. In Linear Algebra, two or more vectors are orthogonal when changes to one vector have no effect on the other vectors. In Software Engineering, we say two things are orthogonal when they are independent of one another; when changes to one thing have no effect on the other.

A real-world example

Consider the real world example of an automobile. A car may be thought of as one large system comprised of many sub-systems. There is the steering system, the brake system, the electrical system, the sound system, etc. These systems are largely orthogonal to one another; they operate independently of each other. Orthogonality – the isolating of one sub-system from another – makes the automobile more reliable. If the brakes fail, the steering system will continue to function; if the CD player fails, the tail lights continue to function.

But orthogonality is not just present between subsystems. There are modules within the subsystems that are orthogonal to one another. A failure in the right rear brake will not immediately affect the operation of the left rear or the front brakes. A damaged headlight does not interfere with the continued operation of the other headlight. Orthogonality – isolating modules from one another – protects the system from the failure of a single module.

Orthogonality in software development

The application of orthogonality to software design is what makes today's large and complex software applications maintainable, portable, extensible, reliable, and testable.

Consider a simplified version of a banking application. Such an application tracks information relating to customers and accounts. For each customer, the following information must be stored in a Customer Data Store: name, address, telephone number, account numbers, and password. The application must also store all of the following account information in an Account Data Store: the owner of the account, the type of account, the current balances, and a record of the transactions on those accounts: interest accrued, withdrawals, deposits, etc.

The information in this Account Data Store is not static. It changes with each deposit and withdrawal. The amount of interest paid on some accounts is also dynamic. The interest rate is tied to the Federal Reserve Funds Rate (Fed Rate). The Fed Rate is the amount of interest that the Federal Reserve charges banks for loans. Because this interest rate changes nightly, the bank must perform a nightly recalculation of the interest on all of the active accounts in the Account Data Store.

Of course, customers need access to their account information. In this example, there are four ways for a customer to retrieve information: asking a bank teller; dialing a special telephone number, and then using the keypad to work through a series of menus; working through the prompts at an ATM; and lastly, using the Internet, filling out text fields in a secure online webpage.

Like the example of the automobile, this banking application can be viewed as one large system comprised of several subsystems (see Figure 1). The Customer Data Store is one subsystem, the Account Data Store is a distinct subsystem, the program which goes out to the Federal Reserve, retrieves the current Fed Rate, and then performs the requisite recalculation of interest on all the relevant customer accounts nightly is another subsystem. Taken as a whole, the various means of customer access can be viewed as a last subsystem with four modules.

The program that retrieves the Fed Rate and recalculates interest is completely orthogonal to the various means that customers have of retrieving account information. Call this program the Fed Rate Retriever and Recalculator (FRRR). To function properly, the FRRR needs the following things: a means of retrieving the Fed Rate, a means of retrieving account data from the Account Data Store, the logic to perform the necessary recalculation of interest on each account, and lastly, it needs a means to input the new account balances back into the Account Data Store. The FRRR operates independently of ATMs, web pages, telephone dial-ups, and tellers. The means of customer access are irrelevant to the FRRR's operation.

If orthogonality has been observed and properly applied in the design and implementation of the FRRR, then there is no reason for the FRRR to be modified if a change is made to any of the customer access modules. Further, this level of insulation from the customer access modules ensures that changes to those customer access modules will not cause the FRRR to fail if one of those access methods fails.

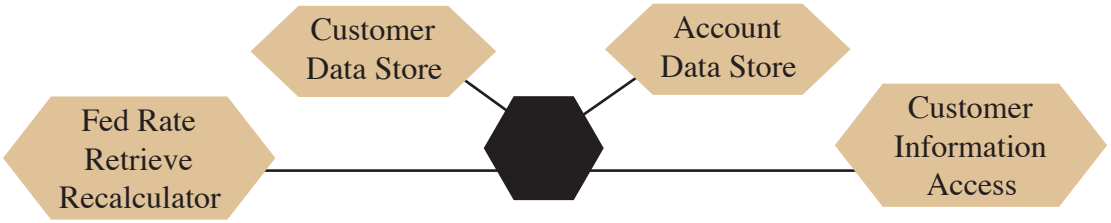


Figure 1: *The four subsystems of the Banking Application*

The proper application of orthogonality also guarantees that the reverse is true. If the FRRR should one night fail, its failure should not prevent tellers, or any other customer access module, from retrieving customer bank balances. True, those balances may be slightly out of date and an adjustment may have to be performed at a later date, but the failure of the one piece of software, the FRRR, will not cause the entire banking application to fail.

The proper application of orthogonality makes this system more reliable;
a failure in one part of the system does not negatively affect any other parts of the system.

Consider the Account Data Store. The information in the Account Data Store may be organized in any number of ways. The data could be stored in numerical order by account number. Alternatively, the data could also be separated first by type of account and then sorted numerically by account number. The number of valid organizing strategies for this data is almost unlimited. Strategy selection is a function of the amount of data and the time that is available for each operation.

The internal organization of the data in the Account Data Store is independent of the Customer Data Store, FRRR, and customer access modules. As long as the Account Data Store provides a constant unchanging means of accessing the data that it contains, there is no need to code

into the Customer Data Store, FRRR, or the customer access modules any knowledge of the internals of the Account Data Store. The Account Data Store has an orthogonal relationship to the other parts of the Banking Application.

A programmer or systems architect will recognize the orthogonal relationship of the Account Data Store to the other parts of the system. But how is this relationship preserved in the implementation? As a practical matter, the Customer Data Store, FRRR, ATMs, dial-ups, tellers, and the webpage all rely on the Account Data Store for the information needed to perform their functions. How can they retrieve the data without having to know the structure of the data store?

Well-defined Interfaces

The answer is that a layer of code is written to sit between the Account Data Store and all the other elements in the system. The Customer Data Store, FRRR, and customer access modules never interact directly with the data in the Account Data Store. In order to retrieve information on bank balances, etc., the other application modules call the same set of specific methods provided by this interface layer. The interface layer interacts directly with the Account Data Store and returns the requested information to the Customer Data Store, FRRR, and customer access modules.

Since the interface layer is the only thing that needs to understand the internal structure of the Account Data Store, we have effectively *decoupled* the Account Data Store from the other parts of the system (see Figure 2). As long as the interface continues to behave in the same way, the underlying structure of the Account Data Store can be changed at any time with no risk to any consumer of the information in the Account Data Store.

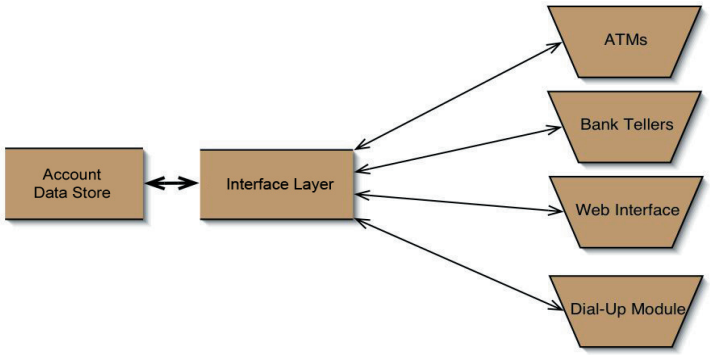


Figure 2: *Decoupling the Account Data Store from the other subsystems.*

In real life, data stores, such as the Account Data Store used in this example, are often proprietary applications. If the bank makes a more cost effective deal with a different database vendor, orthogonality shows its strength. Commercial databases often have proprietary interfaces. How much of the system has to be rewritten in order to utilize the new database? What is the cost of this modification in time and money? With the proper application of orthogonality, the answers to these questions are far more favorable than if each component of the application must be modified to interact with the new Account Data Store.

If the interface layer as described above does not exist, then each module must interact directly with the database (see Figure 3). This means that the knowledge of how to query the database must have been placed into every module and subsystem. In the bank application, code will have to be rewritten in the webpage, the dial-up module, at least once for the ATMs and then propagated throughout the ATM network, the computers that service the tellers, and the FRRR subsystem.

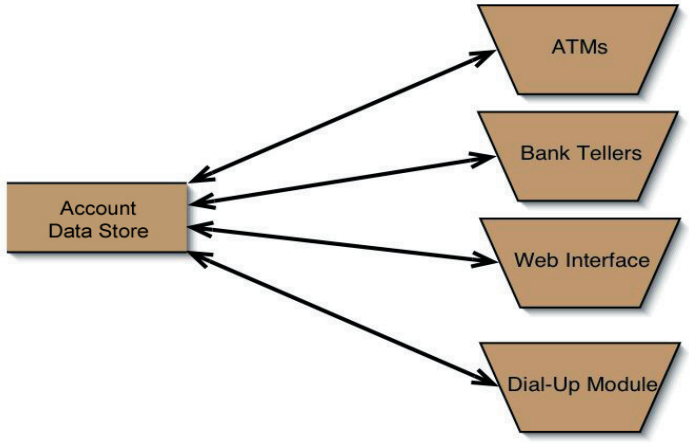


Figure 3: *System Design without Decoupling.*

On the other hand, if the programmers have properly applied orthogonality by creating and using an interface layer, then code only has to change in a single place: in the interface layer. The programmer charged with maintaining the system does not have to search the entire code base seeking out code that needs to be changed in order to accommodate the new database. Observance of orthogonality has made this change easier, less time consuming, less risky, and far less costly.

Adding the interface layer to the Account Data Store has also made the application more stable and less prone to failure. Changing deployed code is inherently risky. The code for the webpage, the dial-up module, the ATMs, the computers that service the tellers, and the FRRR is stable, tested, working code. Programmers make mistakes. Being forced to alter the code in many places as opposed to just one place – the interface layer – introduces unnecessary risk of programmer error. It is also possible that in searching the code base for required alterations, the programmer will miss a piece of code which needs to be modified. It is far riskier to make multiple changes throughout the application than it is to make a localized code change.

The addition of the interface layer also facilitates testing.

As stated previously, without the interface layer, knowledge of the database must reside directly within the Customer Data Store, the FRRR, and all the customer access modules. A change of database will necessitate modifying code in these modules. The tester will be tasked with not only testing the new database, but testing all the changes made to the Customer Data Store, FRRR and the four customer access modules as well. In short, it will be necessary to test the entire application again, because every module and subsystem is affected by the change of the database. This is time consuming, unnecessary, and wasteful.

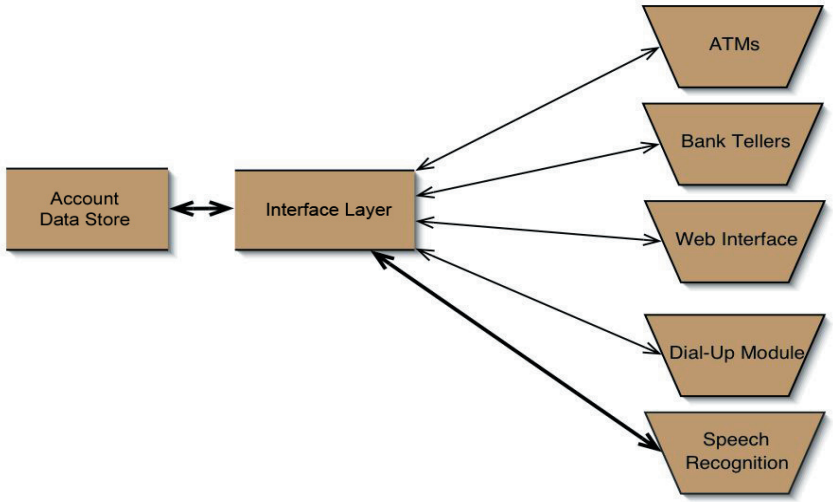


Figure 4: *Banking System extended to use Speech Recognition*

Testing is far simpler with the interface layer as part of the application. If the database serving as the Account Data Store is replaced, the test engineer will only have to test two things: the new database, and the changes to the interface layer. This dramatically reduces the scope of the required testing. Consequently, testing will require less time and effort, which leads directly to less cost.

Extending the system

The interface layer also makes it easier to extend the system's functionality.

The bank may want to provide their customers with an additional way to access their accounts using speech recognition. Customers would use speech with their telephones in the same way that they can presently use the keypad. Employing a series of prompts, the speech recognition module will collect the same information as the other customer access modules: the customer's name, account number, and a password. The speech recognition module will provide customers with the same information as the other four customer access modules: bank balances, etc.

The Account Data Store interface layer, as described above, has known good methods for requesting and receiving data from the Account Data Store. There is no need to write any additional code in the interface layer or in the Account Data Store itself. The speech recognition module will call the same methods that are used to provide information to the other four customer access modules (see Figure 4).

Because orthogonality has been properly observed, the speech recognition and Account Data Store are decoupled. The speech recognition module contains no code specific to the Account Data Store, and the Account Data Store knows nothing about the speech recognition module. Once again, this facilitates testing.
Because of this decoupling, testing of the speech module prior to integration with the larger application does not necessitate any linkage to the live Account Data Store. With the use of a test jig that simulates the interface layer, the system developers can test and debug the speech recognition module in isolation, mitigating the risk of testing the speech module in conjunction with the rest of the live banking application.

Summary

In summary, one or more elements are said to be orthogonal if they are separate and independent of each other and where changes to one thing need have no effect on the others. Recognition of orthogonality between elements, and preserving orthogonality in design and implementation, leads to a highly maintainable, testable, extendable, reliable, and portable system.

Reliability

In common usage, a thing is reliable if it works the first time and every time thereafter. This is often thought of as a binary operation: it works, or it doesn't work. After flicking a light switch, the lights either come on or they don't. After putting the key in the lock, the lock opens, or it doesn't. The notion of reliability implies an expectation carrying forward into the future. Flick the light switch and the lights come on; turn the key and the lock opens.

There is the common underlying assumption that things that work reliably always work exactly the same way. The light switch completes the circuit; electricity flows to the lamp and the light comes on. Turn the key in a lock, the core turns, and the lock opens.

This definition of reliability is applicable to mechanical systems; however, it is not sufficient for contemporary software systems. These systems do not necessarily execute instructions in the same order twice. The specific reasons for this are varied, but have to do with timing. Modern software systems are often asynchronous and event driven. They depend upon data from outside sources: users, networks, and peripheral devices. When and if this data arrives affects the control flow of the software.

Events

When the time flow of the events that drive an application cannot be predetermined, we say that these applications are asynchronous. Because of the unknown time order of events, the same input will not always yield the same result. This is not surprising because in real life the same property holds. For example, if you turn the key before you put it in the lock, the lock does not open. In software, one fundamental feature of reliability is that the result is appropriate to the state of the application at the time that an event occurs. The result may change over time; however, the result is always knowable if the state of the application is known.

To make matters more complex, a tool of great power has been added to the toolbox of the programmer: threads. The concept of a thread is simple; it is an independent execution stream within an application. The easiest way to think of threads in software is to think of threads as individual workers inside an office. To the outside, the office is producing results; inside the office the individual workers work together to produce those results. The control flow of the software, much like the internal operation of the office, will depend upon the order in which the various threads complete their execution; however, to the consumer of the final results, these internal ordering issues are unimportant.

Latency

Latency also plays an important role in any discussion of reliability. The results must be relevant; information loses value over time. An application must produce results in a timely fashion. A service that only updates stock prices every three hours is of little use to a day-trader. Radar that takes hours to make a single sweep is of limited value.

Robustness

Lastly, robustness is a key feature in a definition of reliability. A robust system is a system that is insulated from point failures and, where possible, identifies and handles systemic failures gracefully. In an office, work does not stop if a single person is out sick for the day. The people in the office identify the "point failure" and adjust the work structure of the organization accordingly. The same property should be true of software. There are many point failures which can occur in even the most mundane applications. Each of these point failures must be identified and handled so that the entire system does not fail due to a single point failure. Robustness can be built into applications at the architecture level, but must also be addressed at the code level.

Summary

In the context of modern software systems, reliability means that while the results are not always exactly the same, they are predictable or are at least quantifiable. A system is not reliable if the results are irrelevant because of excessive latency, if by the time the program reports out a piece of data, that data is no longer meaningful. Reliability means that an application is robust. If point, or even systemic failures occur, the application identifies the failure and gracefully handles it.

Maintainability

Maintainability is the aptitude of a system to undergo repair and evolution [Barbacci 97]. The two biggest issues affecting maintainability are the ease with which the maintainer of the system can understand the system's design and the extent to which the implementation lends itself to change.

The software engineer assigned to maintain the code base of an existing application assumes a number of responsibilities. He is expected to maintain and improve the quality of the application (fix bugs). He is expected to test after changes are made whether these changes are made in the pursuit of improving the quality of existent functionality, or due to the addition of new functionality. Over the lifecycle of an application he may have to assume the role of systems architect as well as developer; engaging in a refactoring effort wherein he must redesign and re-implement portions of the code.

The set of engineers responsible for maintaining the code base of an application are often not the software engineers who originally implemented the application. High-level developers or expert researchers may be able to field a system that is reliable, but the system is not maintainable unless it is immediately accessible to the programmers who come in after the development effort and who must assume responsibility for maintaining the system. The worst possible scenario is the one in which the only people who can easily understand a system are the people who wrote it. Under such circumstances, the engineers assigned to maintain the application must read through nearly every line of code in order to understand the application. This sort of learning curve is unnecessarily steep. Such a learning curve can also be unacceptable in terms of time and money.

Design Documents

Good design documents are critical to the effort to maintain software and to mitigate the steepness of the learning curve. The amount and type of documentation required for a system will vary depending on the size, type, and complexity of the system. At a minimum any system that is meant to be long-lived and maintainable must have good, up-to-date, design documentation, and a coding standard (see Appendix B) that is followed during development and maintenance.

Documentation comes in a number of forms. The high level design of the system is found in the system architecture documents. This documentation provides the maintainer with the knowledge of how the system was partitioned in the original design, and just as importantly, the rationale behind the original partitioning of the system. High-level design documentation lays out the overall design and architecture of the system.

Often these types of documents are able to preserve the "why" of a design, while the implementation preserves only the "how." In maintaining a system, the "why" can be critical to preserving the system's integrity. With high-level design documentation, the maintainer can validate each change that is to be made to the system against the original model. Without good design documentation to guide them, multiple maintainers will make changes that are mutually inconsistent, and inconsistent with the precepts of the original design [Bass 99].

That said, software architectures are not necessarily immutable. They are subject to change over time so long as those changes are made with all due deliberation. Again, reference to the original design documentation can serve as a guide to the original designer's intent, and give an indication as to how any profound changes made to the architecture may affect different parts of the system.

When such changes to the system are made, the documentation must be updated so that future maintainers of the system always have a reliable record of the system's current design.

In addition to high-level documentation there is low-level documentation. Low-level documentation concerns itself with the internals of the implementation. Prior to the coding of a system, a software functionality specification document is written spelling out in detail the different objects to be written. This specification includes such information as the function of the object, the inputs and outputs of the object's methods, and error conditions for each method.

Coding Practices

Code comments are another type of low-level documentation that is critical to the maintainer. As a general guideline, CMU recommends that no less thirty percent of the lines of codes in a source base should be comments. Typically comments appear just above the piece of code that they address.

There are two distinct types of comments in the code. The first type of code comment is directed at the developer whose code will interact with the module. Often the maintainer will find himself acting in this role as he makes changes. These comments are notes on the usage of the class and its methods. The test for good usage commenting is that the consumer of the module need not read the implementation code in order to use it. A number of relevant items should be addressed in this type of comment. Among them: error conditions, the expected return types of methods, and any side effects which may affect the variables passed in as arguments (more of these requirements can be found in Appendix B: CMU Coding Standard).

The second type of code comment is targeted at the maintainer who must fix bugs in the implementation details of an existing piece of code. These comments are typically very detailed and not meant to be seen by the consumer. If the module is properly encapsulated, the consumer should not need the knowledge of the implementation details in order to successfully use the module.

Abstractions

The quality of a system's design affects its implementation. *"In developing systems it is important not to lose sight of the value of good software abstractions. Abstraction leads to simpler interfaces, uniform architectures, and improved object models. It is the lack of effective abstractions that results in excessive system complexity. Commonality between components is often not recognized and sufficiently exploited. Without proper design abstractions, needless component differences are created, resulting in redundant software and multiple points of maintenance for fundamentally similar code." [Brown 98].*

Code designed with poor abstractions, and implemented with poor encapsulation makes it difficult for the maintainer to understand the design as embodied by implementation. Making changes to the code may be an inherently risky undertaking. As Brown notes above, in order to change a piece of functionality, the maintainer may have to search out and change code in several different places in the code base. Changing code in one part of the code base may have unforeseen side effects in other parts of the code – it may in fact introduce new bugs throughout the system.

Code that is implemented or maintained in an ad hoc manner is subject to any number of faults, some of which may eventually prove fatal and necessitate a complete rewrite of the system. While we do not agree with all of Yoder's and Foote's conclusions in their paper Big Ball of Mud, they enumerate a number of the real world problems that can occur if code is maintained poorly or allowed to drift with respect to the design:

"Data structures may be haphazardly constructed, or even next to non-existent. Variable and function names might be uninformative, or even misleading. Functions themselves may make extensive use of global variables, as well as long lists of poorly defined parameters. The functions themselves are lengthy and convoluted, and perform several unrelated tasks. Code is duplicated. The flow of control is hard to understand, and difficult to follow. The programmer's intent is next to impossible to discern. The code is simply unreadable, and borders on indecipherable. The code exhibits the unmistakable signs of patch after patch at the hands of multiple maintainers, each of whom barely understood the consequences of what he or she was doing." [Foote 97]

Summary

The use of strong encapsulation, the decoupling of components, good design documentation, good abstractions, and the adherence to a good coding standard by the original developers will result in a decent implementation – an implementation which is easy to understand and change. Moving forward, adherence to these practices by the maintainers will keep the code from becoming unwieldy and incomprehensible.

Extensibility

Extensibility is the ease with which a system can be extended to support new functionality.

"Adaptability is perhaps the most important quality of software. More than half of all software cost is due to changes in requirements or the need for system extensions." [Brown 98].

Extensibility can only be achieved by design. Making changes to extend systems which are poorly designed is very expensive. A properly crafted system architecture will enable the addition of new functionality without requiring changes across the entire system. Unnecessary changes to the system are time consuming, risky, and necessitate retesting of components which were previously tested and successfully integrated. [Batman 93]

Interfaces

Designing the right interfaces between components is the key to extensibility. Interfaces facilitate communication between components. A good interface permits the addition of new functionality without having to re-engineer the internals of the existing components.

"If properly specified, it is the set of interfaces that provide decoupling between components in the system . . . It is in the software interfaces where system stability and adaptability are realized." [Brown 98]

The extensibility of car sound systems is facilitated by a well thought out and well understood interface. An AM / FM cassette player can be replaced by a unit featuring a CD player with relative ease. This is made possible by the use of a simple interface which supplies a standardized power connector, and a standardized connector for audio output. As stated above, a good interface obviates the need to make changes to other parts of the system in order to extend the system. Because of the sound system interface, installing a new audio unit does not necessitate making a change to the speakers. Manufactures of audio units do not have to concern themselves with the speakers, or anything else in the car's system that lies beyond the interface.

Partitioning

Appropriate interfaces are the result of design decisions made when partitioning the functionality of the overall system. Good partitioning relies on finding the proper abstractions to reduce system complexity and arrive at proper divisions between components. *"Managing complexity is achieved through arriving at the right amount of abstraction for system architecture. Management of change is focused on the development of common interfaces. This is how services are accessed. Common interfaces allow for component replacement of applications and systems. Good architecture is about matching the right amount of abstraction with the appropriate set of common interfaces." [Brown 98]*

Relation to Reliability

Reliability does not equate to extensibility. The poor use of abstractions during the design phase can result in a system which is reliable but not extensible.

For example, snow tires extend the usefulness of a car, allowing the vehicle to travel more safely in winter conditions. Changing from normal to snow tires is a relatively simple operation. The tire is attached to the car by securing lug nuts on the threaded wheel studs protruding from the brake hub. The interface between the tire and the rest of the car is at the brake hub.

A poor abstraction would have been to view the system of wheel, brake hub, and the axle as one piece, locating the interface not at the brake hub, but where the axle meets the differential. Changing from normal tires to snow tires would necessitate changing out a very large piece of the car's infrastructure. This would be a considerably more awkward and expensive operation, requiring a trip to a mechanic. This does not imply that the car would be any the less reliable. This abstraction is just as valid, but it provides flexibility at an inappropriate place in the system.

Summary

When designing software systems, systems architects need to be cognizant of extensibility. They must look to the future and ask themselves: where in the system is flexibility and extensibility most likely going to be needed? In the example of a car, certain components wear out more quickly than others. The more often a component wears out, the easier it should be to replace. Automobile manufacturers address this issue. Automobile manufacturers find the right abstractions. Systems architects need to apply themselves to the question of how much time and expense is going to be required to add functionality. Like the tire changing example above, how much of the system is going to have to be touched in order to accommodate a change? Through the use of good abstractions that will yield the proper interfaces, systems architects can design for extensibility.

Portability

Producing an executable version of an application on a new platform, which is already working on another platform, is called *porting* [Mooney 94].

Typically, a platform is defined as a particular piece of hardware running a specific operating system. Changes to the system's hardware, operating system, or both, may necessitate a port. As computers get faster and cheaper it is often the case that the hardware component of a system will be replaced many times in the system's lifetime. Likewise, advances in operating systems will motivate a port either because of new features or because of diminished support for older versions.

The port of an application can be motivated not only by a wholesale change of operating system, but also by incremental changes within a family of operating systems. For instance, Microsoft CE, meant to work with PDAs and other limited resource devices, supports a reduced set of services relative to the other Microsoft operating systems. Additionally, there are sufficient differences between Microsoft NT, Microsoft XP, Microsoft 98, Microsoft 95, and Microsoft 2000 to sometimes necessitate rewriting portions of application code.

The term *portability* has to do with the amount of effort involved in the process of porting a working executable from one platform to another. The metric in determining whether or not a piece of software is portable is the cost of the effort required. Specifically, an application is deemed portable if the development cost to port the software is less than the cost of an entirely new build-it-from-the-ground-up development effort. [URL 01]

The problem with porting lies in specialization – the extent to which the initial software was developed with dependencies on a specific technology (an operating system or piece of hardware). At times there are reasons why external dependencies are unavoidable. However to the extent that these dependencies exist, they must be dealt with in the porting effort, and may represent a substantial portion of the effort and cost.

The complexity of a port is a function of two things: the structure of the software being ported, and the technical details of the new platform being supported. Of these issues, the structure of the software is controllable and must be managed carefully.

Effective partitioning – the proper division of the system into its functional components – in both the design and implementation of the original system is critical. In a good design, those parts of the program which have external dependencies will have been modularized; separated out from the rest of the program, and encapsulated behind strong interfaces. This use of encapsulation greatly enhances the portability of the software. Most of the development effort during porting will be directed at those modules encapsulating the external dependencies, focusing on changing the internals of these modules to support the new platform. The core of the program that contains the domain knowledge of the application should require little or no attention because it interacts with the external dependencies through strong interfaces.

Testability

"Testing is used to identify defects during construction and to assure that completed products possess the qualities specified for the products." [McGregor 01] McGregor describes testing as *"any activity that validates and verifies through the comparison of an actual result, produced by 'operating' the artifact, with the result the artifact is expected to produce, based on its specification."*

Testability is the extent to which a software system facilitates testing.

Testability hinges upon two things: the ease with which the tester can understand the system, and the extent to which the structure of the software lends itself to the above: to operating an artifact and comparing it to an expected result.

Except on rare occasions when the tester (or test team) is present at the inception of a software project, which is to say in the design phase, the tester comes by his knowledge of a system through documentation.

Good Practices

Documentation comes in several forms. Requirements documentation lays out the goals of the system. User documentation tells the tester how the consumers of the system are expected to operate and interact with the system. High-level architecture documentation gives the tester necessary insight into how the systems architect has partitioned the system. The same software functionality specification that details for the developers how each module is to be implemented, informs the tester as to the functionality of each module and details each module's interface. Good comments in the code are another critical source of information for the test engineer (for a more detailed discussion of Code Comments, see *Maintainability*).

Documentation – good, bad, or indifferent – is useless if the design and implementation of the system do not lend themselves to testing.

Systems can hinder testability in a number of ways. One is that the structure is so tangled and convoluted that the tester cannot find reasonable access points; there are no strong interfaces between parts of the system. A system which is structurally sound with regards to testability is one that has been decomposed into small pieces, each of which has clearly defined functions. This gives the ability to create a set of stimuli (in/outs) and quantify the results of program operation.

Poor design or coding practices in the system undermine the integrity of tests even if the tester has access to good interfaces. One such design practice is the use of 'global data' (or 'global variables').

Global data is unprotected by any interface. Global data is available to the entire system all the time. This might seem reasonable. If two or three different modules all need the same piece of data in order to perform their separate calculations, then making that piece of data global may seem like a good solution, but it's not.

How does the use of global data affect testability? How does the practice of using global data undermine the integrity of testing?

The best example is one where the answer to a simple question is desired. The question is: has anyone been in my house since I left it last?

The data needed to answer this question may be easy or hard to acquire. If we assert that the person trying to answer this question lives with a roommate, then the house in question is a global resource to

all occupants of the house. In this case, it is very hard, if not impossible, to answer the question. On the other hand, someone who lives alone and locks the door can easily answer the question. The house is not a global resource; it is a private resource for that single person. In this case, if the house has not been broken into, then the answer is clear.

The goal of the system designer is to create a system where each piece of data has it's own "house". In giving the data a "house" it is given the ability to control access to what goes on inside it. These pieces of data may actually be sets of information that are related to one another, and must be maintained jointly; however, they can be viewed as a single entity with a single home. If the system designer feels the need to introduce the concept of a roommate for a piece of data, then the system's abstractions must be carefully reconsidered since the addition of a roomate risks the ability to ask simple questions and be certain of the answer.

Summary

Lastly, testing is neither solely the province of the tester, nor is it an activity reserved for the final days of a development effort. Professional developers such as Hunt and Thomas [Hunt 99], Kernighan and Pike [Kernighan 99] call for testing throughout the development effort, starting with the developer. Before submitting the code to the common repository, the developer can and should test his own code for a number of common errors (boundary conditions, pre- and post-conditions, the validity of loop invariants, etc.). A great many bugs can be caught when developers exercise this sort of diligence.

Good Practices

Introduction

"The architectural design must provide an effective information protection strategy to encapsulate components, infrastructure, and external interfaces. This protection strategy supports a decoupling of components from the implementation details of other components and external systems. Within the constraints of performance requirements this is one of the most important properties enforced by architecture. It ensures conceptual integrity by managing proliferation of complexity and allows abstraction of stereotypical behaviors. It is a major contributor to the ability of the architecture to provide intellectual control over the system. It insures against side effects and hidden interfaces. An effective information protection strategy has a major impact on the integrability and maintainability of the system as well as on its reliability. We look for protection from the specific details of system components such as operating systems, file management systems, database management systems, hardware devices, inter-process communications facilities, and user interfaces."

Characteristics of an Organization with Mature Architecture Practices.
-- Joe Batman

Good quality does not arrive by accident. As per the above, it is achieved by the application of good practices, and attention to key concepts. What follows is an expanded discussion of those concepts and practices. Some, like abstraction and encapsulation, have been touched upon in the previous discussion of orthogonality and the desired attributes of quality software. Others like Model-View-Controller will be introduced here for the first time. All of what follows will aid the reader in understanding the analysis portion of this report.

Abstraction

An abstraction is a way of looking at things; it differentiates between one group of things in a larger group of things by isolating for certain important aspects. Webster's Dictionary [URL 02] defines abstraction as: "*The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others; analysis. Thus, when the mind considers the form of a tree by itself, or the color of the leaves as separate from their size or figure, the act is called abstraction. So, also, when it considers whiteness, softness, virtue, existence, as separate from any particular objects.*

"Note: Abstraction is necessary to classification, by which things are arranged in genera and species. We separate in idea the qualities of certain objects, which are of the same kind, from others which are different, in each, and arrange the objects having the same properties in a class, or collected body."

In software, the goal of abstraction is not to classify things for the sake of classification. The goal of abstraction is to find ways of looking at problems and problem domains which support the needs of the software architect, developer, tester, and user. These needs may seem different, but the architect, developer, tester, and user each need a model of the world, or an abstraction, that supports present day operations and also supports expansion paths into the future. In software engineering, we refine the definition of abstraction: "*Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate aspects that are important for some purpose and suppress those aspects that are unimportant*." [Rumbaugh 91].

Rumbaugh goes on to say that
an abstraction should have a purpose, and it is that purpose that determines what characteristics are important and which are not worthy of attention. [Rumbaugh 91]

The purpose is always to help solve a problem.

Abstractions in the domain of software engineering are not uniformly good or bad. Inherent in the definition of abstraction is the fact that there are almost limitless abstractions which can be found for a problem. Abstractions are judged good or bad depending on how well they help solve a problem.

Consider the automobile. There are a number of ways to think about this common item. One abstraction considers the car as one car among many on the road, moving from place to place. Another abstraction considers the car as a large collection of parts: an engine, tires, suspension, axles, etc. Both of these abstractions are completely valid. The first abstraction would be of little use to a car mechanic, and the second would be of little use to a highway engineer. Does this mean that these abstractions are bad? No. The abstractions are totally valid, but the appropriate abstraction must be found for each individual's task. If we flip the assignment of the above abstractions, then these abstractions are of high value: the abstraction which considers the car as a large collection of parts is useful to the mechanic, and the abstraction which considers the car as one car among many on the road is helpful to the highway engineer.

Abstractions are a way of looking at things. They are a useful way of looking at problems by sorting the units in a problem domain for a set of common and important characteristics, while ignoring other irrelevant characteristics. Abstractions are good or bad to the extent that they contribute to, or detract from, a solution to a problem.

Encapsulation

Encapsulation is the bundling together of data and the operations on that data into a single module. Encapsulation is used to hide and protect the data and operations of a module from the rest of the system.

The data items in the module are called "attributes" and the operations on the data are termed "methods." The internal data are not visible to the rest of the system. Other parts of the system can query the module through the methods provided by the module. These methods do not permit the other parts of the system to access the internals of the module. Since the consumers of the module are unable to directly access the data, they do not need to concern themselves with how the data is stored inside of the module in order to complete their operations. The set of methods that bridge the gap between the internal data and the outside world constitute the module's interface.

Encapsulation is an answer to the problems caused by the lack of data protection, particularly the use of global data in procedural programming. Data sharing results in data that is unreliable as it may be altered at any time by any part of the program. This problem is mitigated by encapsulation which facilitates data hiding.

In older programming paradigms, the code that acted upon the data was distributed throughout the system. There was no single place which described the system's expectations of, or interactions with, the data. This type of distributed access to data made it virtually impossible to understand the effect of changes made to the system. Small changes to the system often resulted in "side effects" – devastating bugs whose origins were difficult to ascertain. Encapsulation fixes the operations on data in a single place, the module. Encapsulation allows for changes to be made to the internals of an object without having to make changes to other parts of the system. Encapsulation protects the rest of the system from changes. *"Encapsulation prevents a system from becoming so interdependent that a small change has massive ripple effects"*. [Rumbaugh 93]

This is critical to extensibility, testability, and long-term maintainability.

Encapsulation supports extensibility by making it possible for the system to provide new functionality at minimal risk. A module's internal behavior can be changed so long as the new functionality supports the old interface. This modification of the internal behavior will be transparent to the rest of the system. This obviates the need for changes to other parts of the system, which is an inherently risky and expensive proposition.

By providing crisp definitions of modules and their interfaces, encapsulation also supports extensibility simply by making the design of the system easier to comprehend. The learning curve for the programmer is minimized.

Testability is also facilitated by strong encapsulation. Good unit tests rely upon simple, well understood interfaces and uncomplicated modules. Also, with strongly encapsulated modules, system level behavior can be viewed as the interactions between well known pieces. Thus, system behavior can be described at the module level. By testing just the module that was changed, system performance can be understood without having to run the entire system. The cost of retesting after changes are made to the system is greatly reduced, as wholesale retesting of the system is often unnecessary.

Object Oriented design is a paradigm that relies heavily upon strong encapsulation. Encapsulation, inheritance, and abstraction are the three pillars of Object Oriented design. [URL 03] Programming languages that support Object Oriented design through the use of classes such as C++ and Java support encapsulation by providing specific techniques for the protection of data and its operations.

Object Oriented Programming

The basic element in object oriented design and programming is the object. An object consists of data and the operations on that data. Thus an object is an encapsulation of data and operations. The data (called the 'attributes') and the operations (called 'methods') are hidden from the rest of the program. This approach differs from traditional programming that treats data and functionality separately. [Archer 95]

One of the most powerful aspects of object oriented design is encapsulation. Encapsulation supports data hiding, also called data protection. Data in more traditional programming paradigms – which separated procedures from the data – was often global, meaning that it was available to the entire system. This often resulted in data corruption. In object oriented programming, the only way that another part of the program can talk to an object is through a set of methods provided by the object. This set of methods is called the interface.

"Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation – different objects are free to implement requests differently." [Gamma 95]

"The abstractions that emerge during design are key to making a design flexible." [Gamma 95]

Object oriented design promotes looking at the design with an eye toward grouping things together that share common characteristics. Conversely, by ignoring unnecessary details, orthogonality surfaces. The abstractions that emerge from the partitioning of the system, and the grouping of data and operations into objects, result in better systems; systems that are reliable, extensible, maintainable, portable, and testable.

The usage of word *object* in the literature is often vague and ambiguous. [Rumbaugh 91] The word *class* is often used synonymously with object; for this document we adopt the usage of class as synonymous with object.

Inheritance

There are exceptions to the strict data protection provided by classes. One of these exceptions is the use of inheritance. Inheritance violates encapsulation – although in a principled way. Inheritance is a powerful tool and is one of the three concepts that motivates Object Oriented design and implementation: encapsulation, inheritance and abstraction. [URL 03]

Inheritance provides a number of benefits such as type compatibility and the provision of a common interface. These benefits flow from the fact that *"Object Oriented modeling and design promote better understanding of requirements, cleaner designs, and more maintainable systems."* [Rumbaugh 91] The use of inheritance also maximizes the reusability of segments of code thereby minimizing the cost of system construction. *"Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in the parent class."* [Gamma 95]

Simple inheritance between classes follows the real life model. There is a parent class and a child class. The parent is known as the superclass and the child the subclass. The subclass inherits the attributes and methods of the superclass. Subclasses not only inherit the attributes and methods of the superclass, but they also typically extend the functionality of the superclass by containing additional attributes and / or methods of their own. In this way, a child class can be said to be a specialized version of the parent class.

Consider a software system to track the inventory of parts for the service center of an automobile dealership. There is a parent class, *GenericPart*. *GenericPart* has four attributes, four pieces of data. They are *partNumber*, *name*, *cost*, and *numberOnHand*. All subclasses inherit these four attributes.

Tire and *SteeringWheel* are both subclasses of *GenericPart*. In real life they would contain additional attributes, the distinguishing characteristics of tires, and steering wheels; for simplicity those attributes are not shown in the diagrams. Figure 5 shows that the subclasses, *Tire* and *SteeringWheel* inherit from *GenericPart*.

A child class can have its own subclasses. *Tire* is a subclass of *GenericPart*. It inherits the four attributes of *GenericPart*. *Tire* is the parent of other classes: *Spare*, and *Regular*. These two classes have their own unique attributes; spares are smaller than regular tires, and are not designed for continued usage, etc. (See Figure 6).

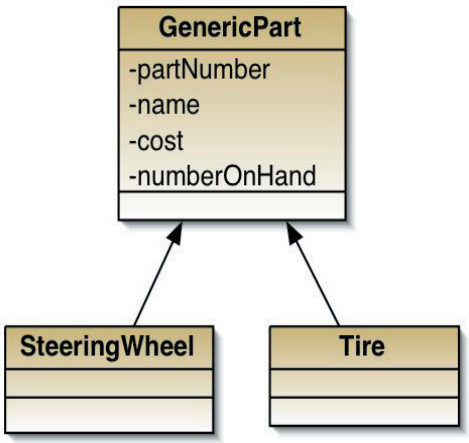


Figure 5

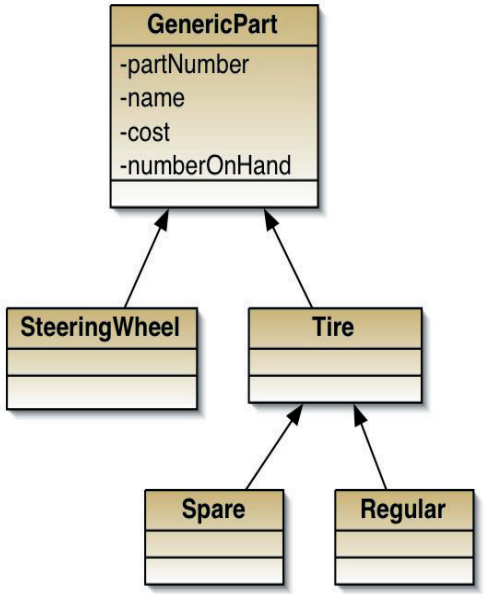


Figure 6

Subclasses may further specialize on their parent classes by overriding certain methods or attributes inherited from the parent. A parent class, *Shape*, may have a method called *computeCircumference* for computing the circumference of a closed curve. The subclass *Circle* can override this inherited method to provide the algorithm to compute the circumference of circle. Similarly, the subclass *Ellipse* will override *Shape*'s *computeCircumference* with its own implementation of *computeCircumference*.

There are ways to recover the encapsulation that inheritance violates. Built into programming languages, such as C++ and Java, is the facility to provide varying degrees of protection for attributes and methods. Attributes or methods may be designated as *private*, *protected*, or *public*. These designations restrict access to the labeled attribute or method. *Private* means that the method or attribute is restricted to the class in which it resides. *Protected* allows access to the subclasses of a superclass. *Public* methods or attributes are accessible by subclasses and any other part of the program.

A last way to specify access within a class is with the *friend* designation. Within a class other classes may be named as *friend*. These named classes are granted complete access to the methods and attributes of the naming class. In addition to the public methods and attributes, methods and attributes labeled as *private* and *protected*, are fully accessible to the *friend* class.

The correct relationship between classes is sometimes obscure. Confusion centers on whether a class is a child of a second class, or is contained in the second class. We can ask two "trick" questions that can help to determine the relationship between two classes: 'is a' and 'has a'. [Rumbaugh 91] The question 'is a', addresses the question of inheritance. The 'has a' question addresses the question of containment.Returning to the example above of a system for tracking car parts, we have a new class, *Car*. To understand the relationship between *Car* and *GenericPart*, we ask the two questions. Is *Car* a *GenericPart*? The answer is, no. Has *Car* a *GenericPart*? The answer is, yes. *Car* and *GenericPart* have a containment relationship.

Ask the same questions of *Tire* and *GenericPart*. Is *Tire* a *GenericPart*? The answer is, yes. So *Tire* is a subclass of *GenericPart*. For the sake of completeness in this example, we ask the second question: has *Tire* a *GenericPart*? The answer is, no. Figure 7 represents these relationships.

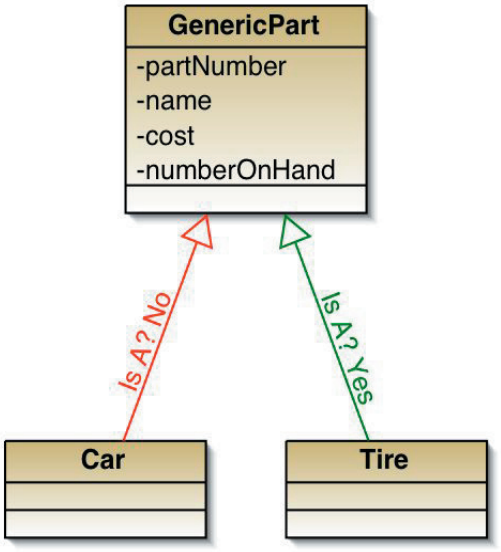


Figure 7

Model-View-Controller

"Several problems can arise when applications contain a mixture of data access code, business logic code, and presentation code. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. High coupling makes classes difficult or impossible to reuse because they depend on so many other classes. Adding new data views often requires reimplementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods."

"The Model-View-Controller design pattern solves these problems by decoupling data access, business logic, and data presentation and user interaction." [URL 04]

The Model-View-Controller design pattern was written in Smalltalk, a language developed in the 1970s at Xerox Parc. Smalltalk was, and remains, an object-oriented language. It supports the notion of classes, methods, messages and inheritance and was the first computer language based entirely on the notions of object and messages. [URL 05]

Model-View-Controller was developed at Xerox Parc as a way of dealing with one of the first computer systems to offer a view to the user, and to permit user feedback.

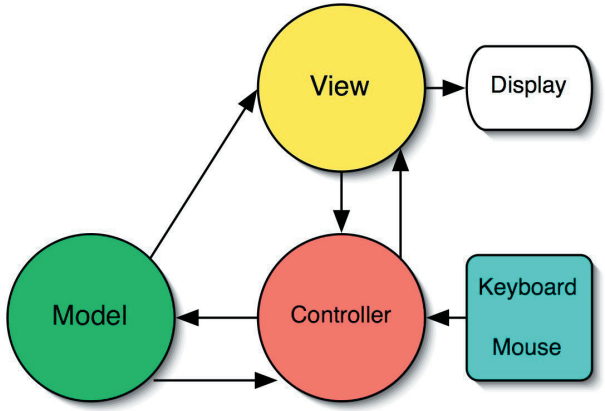


Figure 8: *Model View Controller Data Exchanges*

As stated above, MVC was one of the first examples of the use of objects and decoupling; the three parts of the system were decoupled; explicitly separating Model, View, and Controller. The View was the part of the system that dealt with the screen presented to the user. A keyboard and a mouse served as user input devices. Input from these devices was handled by the Controller code. The Model was the part of the program not directly concerned with interacting with the use. The Model contained the business logic; the core logic of what it was the application was actually designed to do.

The designers of MVC recognized that the Model, the View, and the Controller were orthogonal to one another. The designers of MVC went on to use objects as the decoupling mechanism. *"The formal separation of these three tasks is an important notion that is particularly suited to Smalltalk-80 where the basic behavior can be embodied in abstract objects: View, Controller, Model and Object. The MVC behavior is then inherited, added to, and modified as necessary to provide a flexible and powerful system." [URL 06]*

With the advent of personal computers, the MVC pattern has become one of the accepted patterns for designing applications that interact with the users through the use of a screen, keyboard, and/or a mouse.

Summary

Encapsulation, abstraction, and inheritance all come together in Object Oriented design and implementation. Object Oriented programming and design promotes orthogonality; decomposing systems into like parts and keeping parts that share no relationship separate. As a consequence, Object Oriented programming provides powerful support for Reliability, Maintainability, Portability, Extensibility, and Testability.

Maintainability, Portability, Extensibility, and Testability while sterling qualities in and of themselves, are also desirable because they drive down costs. Maintainability, Portability, Extensibility, and Testability save in human labor. Programmers do not have to spend an inordinate amount of time learning the system, because Object Oriented design results in clear, clean systems. They to not have to hunt through the code searching for functionality spread throughout the system in order to make a change. Additionally, programmers do not have to spend time in costly retest of the entire system when they do make a change. Changes to a proper Objected Oriented system will not result in mysterious failure cascades. Programmers do not have to rewrite or re-engineer large parts of the system to accommodate a port or to extend the system's functionality.

Lastly, a system that is unreliable is of little value. Object Oriented design supports Reliability by protecting data from the pernicious influence of other parts of the system. Reliability is enhanced because required changes to the system do not necessitate rewriting large portions of code. Programs are more robust because functionality is not vulnerable by being spread throughout the system but rather is localized in well defined objects. Reliability is enhanced because an Object Oriented system is simply so much easier to test.

Much of the analysis that follows is informed by the concepts and techniques of Object Oriented design and implementation. Please refer back to the preceding pages as often as is necessary. There is also a Glossary of terms at the back of this report which should prove helpful.

Analysis of THS(X) 3.2.0

Fatal Defect #1: Extensibility and Portability

Issue #1: The design of the core object model used in THS(X) is not extensible, portable, or maintainable.

Background:

Object Oriented Analysis and Design (OOAD) is one of the current favored techniques for system analysis and construction [Boggs 02]. While the technique contains Object Oriented (OO) in the name, and many discussions of the techniques use the word "Object", an Object is actually an instance of a Class. A Class is generally accepted as a type of Object. For instance, if we posit that there is a Class that represents a circle, there can be many instances of the circle Class, each with a different center and radius.

One of the reasons that OO design is very popular is because, when used properly, it strongly supports encapsulation. Consistent use of strong encapsulation maximizes the reusability of segments of code and minimizes the cost of system construction.

In OO software design, encapsulation is used to bind data and operations on that data into a strong unit called a Class. Generally the data contained inside a Class is not directly accessible outside of the Class. The access to the data is accomplished through the use of the operations on the data, which are also part of the Class and define the interfaces of the Class. A strongly encapsulated Class will only allow access through the member functions that define the Class's interface.

An OO design technique that interacts with encapsulation is inheritance [Rumbaugh 91]. The proper use of inheritance allows for Objects that are related to exploit the commonality between them while still safely allowing for the full expression of their differences. The reason that encapsulation and inheritance interact is that inheritance purposely violates encapsulation – though in a principled way. Because inheritance violates encapsulation, it can be quite powerful when used appropriately; however, as with most powerful tools, when it is used inappropriately, it can be destructive.

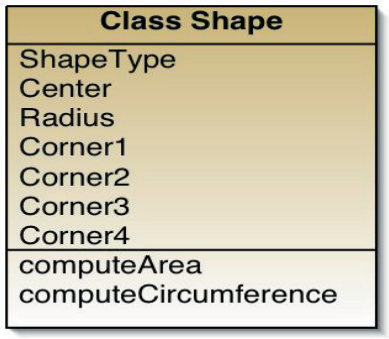


Figure 9: *Shape Design option #1*

The classic example of the appropriate use of inheritance is to define a Class named Shape. In our simplified example, the operations that Class Shape should support are: computeArea and computeCircumference. We posit that we want to support the area and circumference methods on the following types of shapes: circles, squares, and triangles. These methods are interesting because they are simple to compute for these shapes, but each is computed quite differently. A few designs will illuminate the appropriate and inappropriate use of these concepts.

The first design, which is proposed to address this set of requirements, appears to be the simplest. In this design, we have only a single class, Class Shape. This Class is detailed in Figure 9.* In this initial design of Class Shape there are 7 member variables and 2 member functions. Member functions are typically called methods. In this design all of the information to do the work of the methods is present.

There are a number of interesting points about this design. This proposed design uses a variable named ShapeType to know what type of shape it is. Depending on the value of ShapeType, the methods computeArea and computeCircumference will use different algorithms and different member variables to do their work. For example, if the ShapeType is a circle, the computeArea and computeCircumference methods will use the Center and Radius member variables. If the ShapeType is a triangle, then the methods will use Corner1, Corner2, and Corner3. Lastly, if ShapeType is a square, then the methods will use Corner1, Corner2, Corner3, and Corner 4. One of the interesting things about this design is that the proper function of the methods depends heavily on the ShapeType value. In order to do the correct calculation for each shape, each of the methods will have to make a set of decisions based on the ShapeType.

Another interesting point about this design is that all of the information that is required for each type of shape is actually in the class. Based on the enumeration of data dependencies in the previous paragraph, we can clearly state that only the circle ShapeType will ever use the Center or Radius member variables. This is not a problem per se; however, it does have some unfortunate consequences. One of the most unpleasant effects of this data dependency is that if the system must be expanded to include, say a pentagon, the Shape Class must be modified. At a minimum, a Point5 data member would have to be added to this class. As it has been described in this simplified example, the method interface to the Shape Class would not change; however, in the real world, the methods that are used to construct the Shape Object would need to change as well.

To summarize, this design is good because it is tightly encapsulated – only the methods on this class may interact with the data. This design is not perfect; it can easily be improved. But this design is not easily testable, extensible, or maintainable. This design is not easily testable because the behavior of the methods is dependent on a piece of state, the ShapeType, in the class. This design is not easily extensible because each time a new type of shape must be supported, the object must be modified and the entire object must be retested since it has changed. Lastly, this class is not maintainable because whenever a change must be made to the methods that compute area or circumference, all of the shape types are put at risk of being corrupted. In practice, for a simple object like this these are manageable problems; however, this type of design does not scale well. There are better designs to solve this type of problem.

The proposed second design looks more complex, but actually significantly improves on the first design. In the second design, inheritance is used. There is specific language that is used to describe inheritance relationships. The two most common terms are superclass and subclass. When two classes are related by inheritance, one is the subclass and one is the superclass. The superclass is the class that is inherited from. The subclass is then the class that is inheriting. This use of inheritance is quite similar to the common, English language definition. The superclass can elect to share certain pieces of data and functionality with its subclass that it does not make available to the "general public"; it can selectively pass on functionality and data to its descendants. This allows for a subclass to extend, or specialize, the functionality provided by the superclass.

* Throughout this document when Object diagrams are included they will be drawn in accordance with the Unified Modeling Language (UML). This diagrammatic language is commonly used within the technical community for Object designs. The way in which UML is used in this document is very limited. An attempt will be made to sufficiently support each diagram with text so as to make learning UML to understand this document unnecessary. If further information on UML is desired or required, please consult a UML reference book such as The Unified Modeling Language User Guide by Booch, Rumbaugh, and Jacobson.

The second design is shown in Figure 10. In this design, there are four classes instead of one. In this design each class has a more sharply defined responsibility in the system. The four classes are the following: Class Shape, Class Circle, Class Triangle, and Class Square.

In the second design Class Shape plays a smaller role than in the first design. In the first design Class Shape contained all of the data and also all of the processing methods on the data. In the second design Class Shape contains no data. One of the problems with the first design was that all of the data for each type of shape was stored in Class Shape. This forced the creation of a variable (actually an instance of an enumerated type, ShapeType) that defined which fields in the object were valid. In this second design, the appropriate data is stored in the appropriate subclass. With the introduction of the shape specific subclasses, there is no need for a ShapeType variable because only the data relevant to a shape is stored in the subclass.

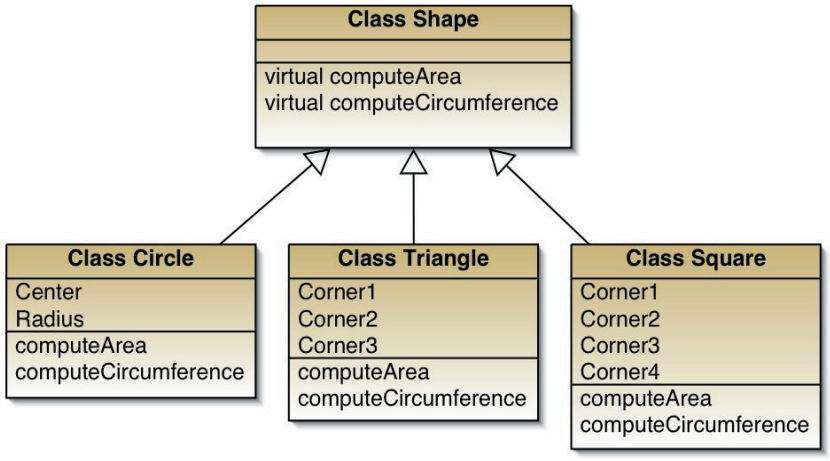


Figure 10: *Shape Design option #2*

The second difference in Class Shape is that the methods computeArea and computeCircumference are declared virtual. The meaning of the virtual keyword in this context is unique to OO design and implementation. In the OO context a virtual method is one that provides functionality that may be overridden in a subclass. The use of the virtual descriptor on the methods in this design allows for each subclass to override the definition of the computational methods found in the superclass, Class Shape. An extension to the concept of a virtual method is that of a pure virtual method. A pure virtual method is simply a placeholder containing no definition. It leaves the implementation to each subclass.

In this example, computeArea and computeCircumerence could be defined as pure virtual, because the techniques for computing area and circumference each shape are sufficiently distinct from one another that there is essentially no overlap in computation. The distinction between virtual and pure virtual is that a virtual method provides some implementation of the method that may be used, or may be overridden by each subclass. A pure virtual method provides no implementation for the method, and imposes the responsibility for implementation of the method upon each subclass.

With the reduced functionality in this second design of Class Shape, the real work of the system is pushed down into the subclasses: Class Circle, Class Triangle, and Class Square. One thing to notice is that even though there are more classes and a higher number of methods to write relative to the first design, each method now has a significantly simpler

job. The computeArea and computeCircumference methods in Class Circle need not do anything except apply the proper formulas to the Center and Radius member variables. Similarly, Class Triangle, and Class Square can each directly apply the proper formulas using the appropriate data. This separation of functionality simplifies the computational methods tremendously and also insulates them from defects. If a defect is found in Class Triangle, fixing it will not have any effect on the implementations in Class Square or Class Circle. In the original implementation, a fix to a defect in any of the shapes would force all shapes to be retested since the change to address the defect would change code that affected all shapes.

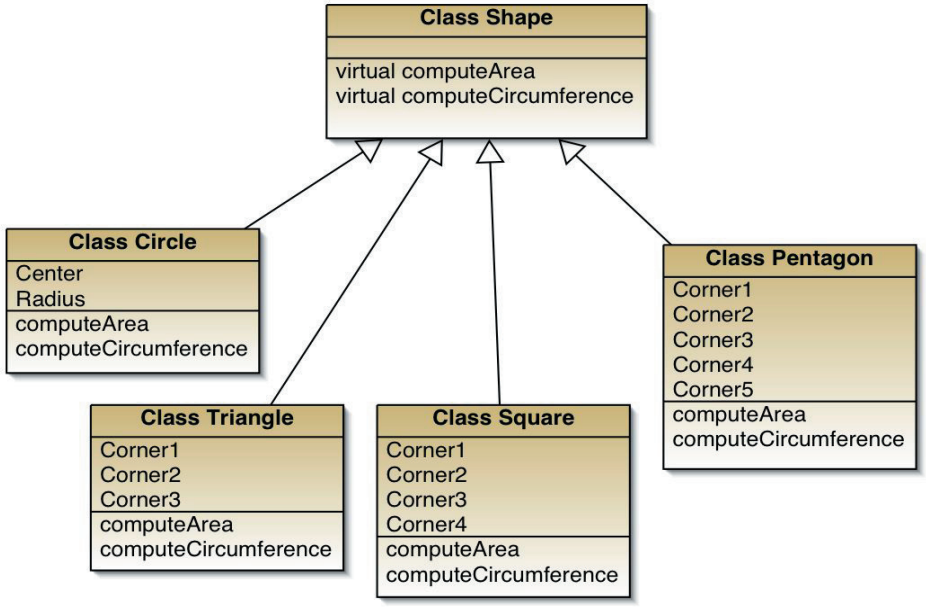


Figure 11: *Shape Design option #2 extended to support pentagons*

Now we again consider the addition of a pentagon to the system. Adding a pentagon necessitated a significant change to the first design. Not only did a data member have to be added to Class Shape, but also all of the computational methods had to be modified. This level of change in the original design forced a complete retest of the system since all of the functionality was bound up together in a single unit. In the second design, the addition of the pentagon is a lower risk, and lower cost, operation. In Figure 11, the system has been extended to support a pentagon.

Looking at Figure 11, one can see that adding support for a pentagon in this design is as simple as adding Class Pentagon to the system and having it inherit from Class Shape. This design supports ease of addition, and does not risk the pre-existing functionality of the other subclasses when pentagon support is added. The reason for this is that Class Pentagon's data members and its implementation of the methods computeArea and computeCircumference are completely independent of (one may say orthogonal to) the data members and the implementations of the methods in the pre-existing classes.

Ideally, this new design would be an unmitigated success and would be perfect. However, use of inheritance introduces some risk. As mentioned, inheritance purposely violates encapsulation. This violation is intentional and generally entails low risk. But the downside of inheritance is that each of the subclasses is now not fully insulated from the superclass, Class Shape.

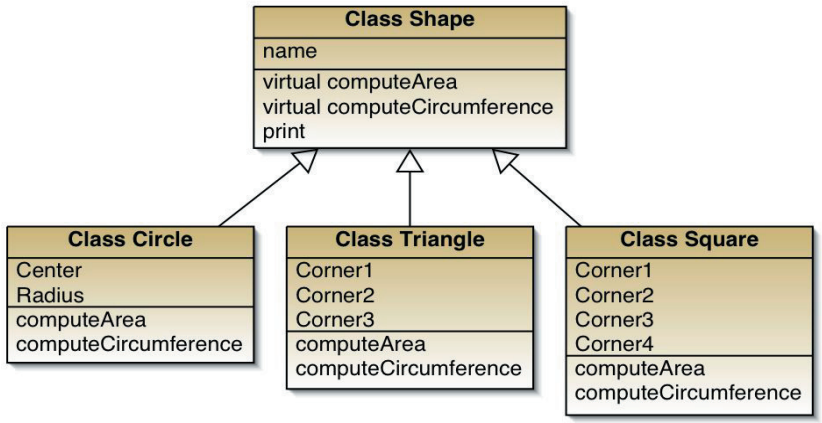


Figure 12*: Shape Design #2 with name*

To maximize understanding, this example has been distilled to eliminate complexities. To clarify the risk that inheritance entails, this example must be extended. The simplest extension that clarifies the risks introduced by inheritance is the addition of a name member variable and a print member method to Class Shape. This extended system is shown in Figure 12.

In the extended system, Classes Circle, Triangle, and Square inherit not only the pure virtual methods from the superclass, Class Shape, but also some functionality – the print method. Because these Classes inherit this functionality from the superclass and expose it as part of their respective interfaces, these classes are no longer fully insulated from the superclass – changes in the implementation of the print method in the superclass will affect the proper function of each subclass. This example is very simple and not controversial.

The exposure to the superclass changes through the inheritance of functionality introduces the subclasses to some risk. One case where the risks are clear is when the subclass is using the output of its superclass as an input to a method in the subclass. In this case, it is clear how a change in the implementation of the superclass can directly affect the correct function of the subclass. The key is that the data is being inherited from the superclass and used as an input to a method in the subclass. In these types of situations changes in a superclass can force retesting of each subclass to ensure the continued proper function of the system. In general, this is not a problem because the subclasses are crisply defined and can be tested with reasonable amounts of effort.

Analysis:

With the knowledge gleaned from the previous design examples, the design of THS(X) will be discussed and will use UML diagrams as an illustrative basis. Some of the classes in the THS(X) system which will be discussed, are large, and expose sizeable interfaces. The diagrams required to represent all of the system's detail are prohibitively large for inclusion in this document. Because of the

size and complexity of the system, distilled diagrams will be used as appropriate.
There are a number of interesting design structures in the THS(X) application. For the purposes of this analysis, selected domain model objects will be highlighted.

In the design process one of the most important things to do is to clearly and cleanly represent the real world. The objects used to model these real world constructs are the domain model objects. To begin the discussion of the design of THS(X) consider Figure 13.

In Figure 13 the basic layout of many of the domain model objects is present. From the top of the diagram, the highest-level class in this inheritance hierarchy is CObject. CObject is a Microsoft Foundation Class. There is a single subclass of CObject, GeoPoint.*+ From GeoPoint, StauderShape is the only subclass. Fourteen (14) subclasses are derived from StauderShape:

- Aircraft
- AttackCone
- AttackConePoint
- ControlPoint
- CoordinationMeasure
- EgressLine
- ExtendedGroundTrack
- FireFan
- LaserCone,
- LaserHandle
- Observer
- Target
- Tic
- Unit

This inheritance diagram raises a number of questions. The first question is why is CObject at the top of this inheritance hierarchy? CObject is a Microsoft Foundation Class. CObject is a very generic class in the MFC hierarchy and supports Microsoft specific extensions. In placing CObject at the top of the inheritance hierarchy for the many of THS(X)'s domain model objects, the portability of these platform independent objects is sacrificed. This loss of portability is offset by the addition of Microsoft-only extensions supported through CObject.

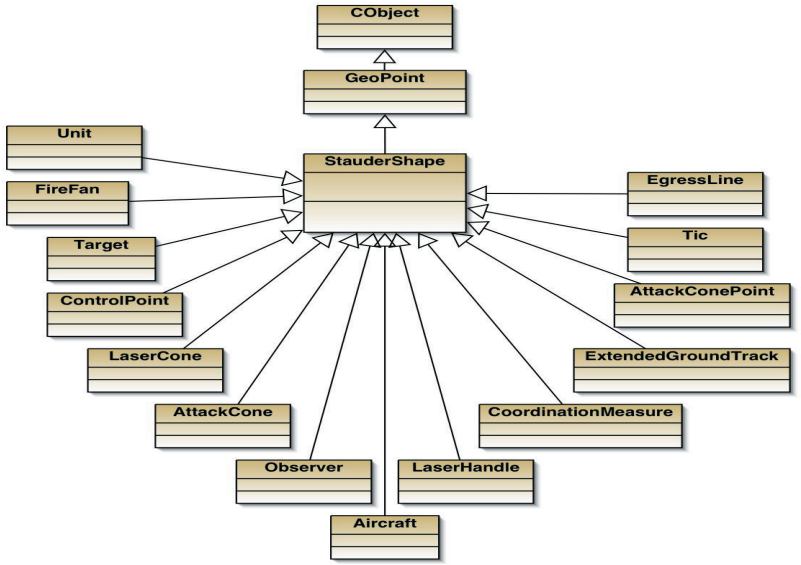


Figure 13*: Highly Distilled THS(X) Design*

One specific extension supported by CObject is a Microsoft specific runtime type identification system [URL 07]. Compilers for the C++ language inconsistently support runtime type identification of classes. CObject supports an implementation of runtime type identification through the use of the virtual method "GetRuntimeType." For this to function properly, the programmer must also use preprocessor definitions (IMPLEMENT_DYNCREATE, IMPLEMENT_DYNAMIC, or IMPLEMENT_SERIAL) in classes derived from CObject. The classes in this system use these preprocessor definitions and so support this Microsoft specific language extension.

* There are other Classes in the system that inherit from CObject; however, those classes are not relevant to this part of the system. The use of CObject is a potential problem, but is separate from the ones of inheritance discussed here.

+ The Stauder Code is relatively faithful in using the Microsoft conventions of all Class names starting with a "C" and Interface names starting with "I." These leading characters do not add anything to the discussion of the architecture and so have been removed when discussing Class and Interface names in this text.

While there are arguably advantages to placing CObject at the head of the inheritance hierarchy since Microsoft provides a number of data structures that store and operate on CObject data for "free," the price is high. This use of Microsoft structures at such a high level in the object design greatly limits the portability of the application. It is significant that CObject is so intrinsic to this design; it makes the application highly non-portable.

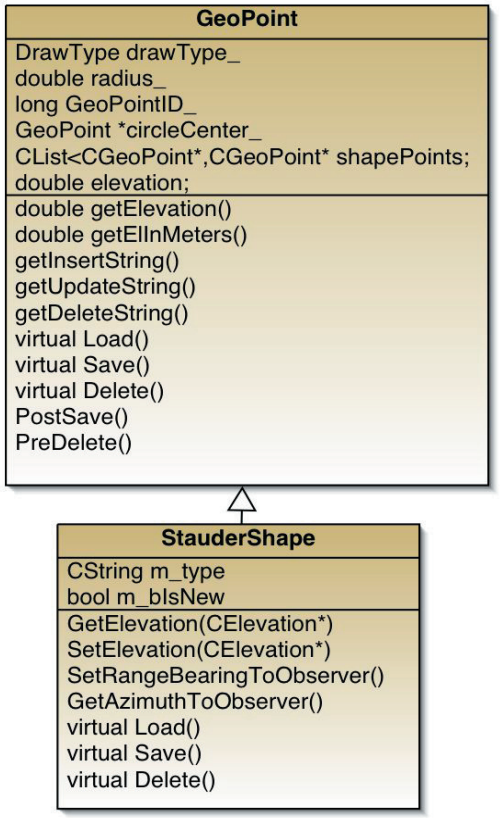


Figure 14: *GeoPoint and StraderShape detail*

Another question that this inheritance diagram raises is why do GeoPoint and StauderShape exist as distinct classes? One of the rules of good design is that there should never be an empty superclass in an inheritance hierarchy. If there is an empty superclass then a distinction is being made where there is no real difference. It is important to note that in this case, neither GeoPoint nor StauderShape are literally empty; in fact GeoPoint is a class that is full of variables and member functions that perform many jobs. The issue is that the inheritance hierarchy does not branch for the first three layers. As discussed above, CObject is a MFC class and cannot be modified, so to use it one must inherit from it (regardless of the cost/benefit tradeoff of doing so). The distinction between GeoPoint and StauderShape is more perplexing. Since there is only a single subclass of GeoPoint, it seems as though this layer of inheritance could be removed without removing any of the intrinsic flexibility in the design.

To further explore the issues with GeoPoint and StauderShape, consider Figure 14. In this diagram, more detail is added to GeoPoint and StauderShape, two central classes in the THS(X) design.

The first thing to do to expand our understanding of GeoPoint is to read a small segment of the class level comment that the developers put into the header file:

"Geographic Point data class that describes a point and also notifies registered listeners of any changes. This is a focal class for mapping applications. Please look at the following examples for suggested ways to use."

From this comment we know that GeoPoint is a class that describes a Geographic Point and handles notification of registered listeners. In addition to the documented functionality, reviewing the code reveals that GeoPoint also contains the following:

- **a definition of DrawTypes**
- **methods to save, load, and delete itself from a database**
- **a method to save itself into, and load itself out of, a serialized file**
- **a variety of accessor methods which can be used to convert between coordinate systems**
- **methods to do projections from this point to another point**
- **methods to compute range and heading to another GeoPoint**
- **methods to pop up a dialog describing this object**
- **methods to register and unregister listeners (implied by the notification, but not explicitly addressed)**

- **a method to set the DTED file name**
- **a method to get "all of the endpoints associated with this point"**
- **methods to support undo**
- **methods to set and get the GeoPointID, which is advertised as the database key value**
- **and methods to specify an arc (center point, radius, start angle, and end angle).**

GeoPoint is doing too many jobs.

GeoPoint should be true to its name and represent a single location. The above list of functionality includes things that are Model, things that are View, and things which are Controller; this interleaving of fundamentally orthogonal functionality into a single object greatly reduces maintainability.* One of the most dangerous things in the design of this class is that the View, or User Interface, is woven throughout the implementation. This interweaving makes the application non-portable in addition to less maintainable. Further, this interweaving also makes it risky to change the layout of the User Interface. **A simple User Interface change, such as changing the text in a dialog box, will affect the core functionality of the system.**

Another problem with GeoPoint is that it can save itself to either a file or to a database. This problem has a number of facets. The first facet is that the method to interact with a database and the method to interact with a file are completely independent – they share no code. This means that these methods must be maintained separately.

Another facet is that neither persistent storage mechanism has any versioning information. In other words, there is no way to expand the definition of GeoPoint without invalidating all data from the past. In addition to this, GeoPoint has methods named "GetInsertString," "GetDeleteString," and "GetUpdateString" which generate SQL strings to accomplish these tasks for the class. All three functions are necessary, but the method "GetInsertString" is never called. This is not maintainable since a naïve maintainer may well believe that all three of these methods are used and update them all. In truth, the functionality that "GetInsertString" purports to provide must actually exist somewhere else in the file. But the maintainer will not know that, thinking that the functionality is acutally proved by "GetInsertString".

Another problem is that the way that GeoPoint interacts with the database is unsafe. The save and load processes for GeoPoint are multi-step. There are methods named "PostSave" and "PreDelete." The reason that these methods exist is because a GeoPoint object can contain a list of other GeoPoint objects. Because the database may enforce referential integrity, the prevention of inconsistent data within a database, it is not possible to create, or destroy the object in one step. Unfortunately, the error handling mechanisms surrounding these multi-step processes are not up to the task of handling failures that may occur in the middle of the process. Database transactions are neither used, nor simulated by this object. This leaves open the possibility that the database will be left in an unknown state since a load, save, or update may not succeed or fail atomically – it may fail in the middle of one of these database operations.

Lastly, the problems that were introduced in the first design example are present in this design. The GeoPoint class contains an enumerated type which is used in methods contained in the class to determine how processing should be done. As described above, this strategy is not extensible and can cause serious testability problems for the system. The testability problems are amplified in this situation since this class is the superclass to other classes in the system. If this class is modified and needs to be retested, then all the subclassses will need to be retested.

* Please see the definition of the Model-View-Controller design pattern (MVC) earlier in this document for further explanation.

The next Class to consider is StauderShape. It too is included in Figure 14. StauderShape inherits from GeoPoint and so contains all of GeoPoint's public methods and adds its own public methods to its interface. StauderShape has a class level comment that sums up its role:

"This object extends the location and elevation data provided by the CGeoPoint class. It adds support for a drawing type that will link the object with a set of drawing properties, and provides a mechanism for the developer to limit the allowed drawing types.

"This class is intended to be subclassed to create business logic classes such as targets and control points"

StauderShape purports to extend GeoPoint by add drawing information at the subclass level. GeoPoint already contains drawing information. The added functionality here then is the limitation on the drawing type. It is interesting that the limitation on the drawing type is expressed as a string, while the drawing types in the superclass are expressed as an enumerated type; this disparity makes more sense when considered in conjunction with the subclasses described in the final paragraph of the preceding class comment.

Before moving on to discuss the next layer in this inheritance hierarchy, there are other significant problems. One problem that is immediately apparent when considering the StauderShape object is that it supports methods that set and get elevation data. This data is stored in the GeoPoint class. Since the elevation data is stored in the superclass, the accessors on that data should also be present in the superclass. The presence of these methods in the subclass is a violation of encapsulation.

In addition to the methods that StauderShape adds to the interface, it also modifies the behavior of the class by overriding some of the virtual methods defined in the superclass, GeoPoint. The astute reader will note that there are only three virtual methods shown in Figure 14; these three methods, Load, Save, and Delete, are the three which are overridden.

In general, overriding a virtual method is a good way for a subclass to expand upon the concept introduced by and (perhaps) implemented in the superclass. There are two ways to handle the overriding of superclass virtual methods. The first is to perform a wholesale replacement of the superclass method. In general this technique is appropriate to use when the method being overridden is computational in nature. For instance, if there had been a computeArea method defined in the Shape Class in our initial example, each class would have been well advised to override that method. The second way of overriding a superclass method is to extend it by invoking the superclass method and then running additional code that specializes the effect of the method for the situation.

StauderShape is extending the functionality of GeoPoint, yet it does not invoke the superclass versions of Load, Save, or Delete. In each case, it does a wholesale replacement of the method. Based on the reasoning as outlined above, this is wrong. The reason that this is wrong is that it introduces an unusual usage model that will be confusing for a maintainer.

This inappropriate overriding of the Load, Save, and Delete methods in GeoPoint by StauderShape explains the duplication of the Boolean that indicates if the object is "new" or not. This duplication is in and of itself a significant defect, indicating poor design. Additionally these Booleans are unnecessary and may cause runtime errors. The Booleans are used to determine if the object – be it GeoPoint, StauderShape, or another not yet discussed – should be inserted into the database table, or whether an existing entry should be updated. This use of the Boolean is an optimization that may cause problems. Instead of trying to track whether or not this is the first time this object has been saved, the system could do a select in the database and see if the record already

exists. If it already exists, then an update will work; otherwise an insert is required. The Boolean is optimizing this select out, but at the price of the developer attempting to track the state of the database. **Tracking the state of another piece of software is a tricky proposition at the best of times and leads to unpredictable defects at runtime.**

Further investigation shows that StauderShape and GeoPoint are stored in distinctly different ways. When a StauderShape is saved to, loaded or deleted from a database, it affects a database table named "Stauder_Shape". The data for GeoPoints is stored in a database table named "geopoint". StauderShape maintains a set of GeoPointIDs that are tracked separately from the GeoPointIDs in the "geopoint" table. **In other words it is possible to have two different GeoPoints in the system with the same unique identifier. This is a significant risk item in this system.**

Additionally, StauderShape interrogates the GeoPoint component of itself for the Latitude, Longitude, and Elevation, and then it saves this information into its own "Stauder_Shape" table. The subclass should not store the data of the superclass. **This is a violation of encapsulation.**

After considering GeoPoint and StauderShape alone, there is no apparent reason that these classes should be distinct, because so far there are no compelling differences. Certainly, StauderShape adds functionality to GeoPoint; however, since StauderShape has no siblings, this added functionality could simply be part of the superclass, GeoPoint. Before making a final judgment about whether StauderShape and GeoPoint are independently worthy classes, we will consider one of the fourteen terminal classes in this inheritance hierarchy and then make conclusions based on the whole structure.

For the purposes of this analysis Class Target, a subclass of StauderShape was selected.

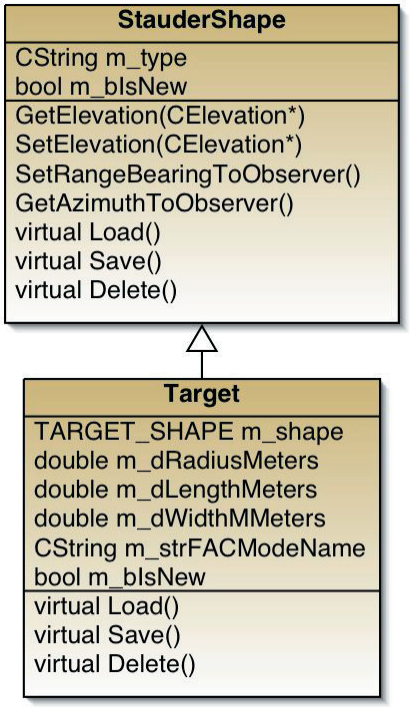


Figure 15: *StauderShape and Target detail*

In Figure 15 the interesting details of Classes StauderShape and Target are shown. There are a number of noteworthy things about the design of Target.

First of all, Class Target overrides the StauderShape definitions of the Load, Save, and Delete methods. Unlike the immediate superclass, StauderShape, Target does the right thing and invokes the superclass forms of these methods

since it is extending the functionality of its superclass. This class stores itself into a database table named "Target", which seems appropriate. It also uses the GeoPointID provided by StauderShape in its storage so that it can reassemble itself with the StauderShape record in the future.

Secondly, Class Target contains an enumerated type that lists a number of distinct shapes for a target, TARGET_SHAPES. There are two problems with Target containing an enumerated type for shapes.

Interestingly enough, the first problem with the enumerated type contained in Class Target is one of the specific issues addressed in the first design example in this section. This enumeration tells methods within the class what data fields to use at different times. In Figure 15, the data fields of Target are listed. The fields support a point, a circle, and a rectangle. This is a direct analog to the initial example and suffers from all of the same problems. To modify the definition of a Target is to have to completely retest the Target Class.

The second problem is that Class Target is defining shape types for the third time in this inheritance hierarchy: first in GeoPoint, then in StauderShape, and now also in Target. This is even more compelling when we consider that there are only four classes involved in the inheritance chain. It is not clear why Target overrides the definitions of both parent classes.

The conclusion to be drawn from this information is that shape types are ill defined in this system.

Lastly, Class Target has a method that has the following comment for the SetType method:

"Inherited method from CStauderShape. Do Not Use This one. To change the type of a Target, use SetTargetShape and SetMissionTarget. @see SetTargetShape, @see SetMissionTarget"

This comment is shocking and further supports the statement that shape types are ill defined in this system. This class, which inherits from StauderShape is unable to use the method in the superclass. Instead of making the method virtual and overriding it, a comment is made that it should not be used. It is not clear what will happen if it is used, but it is clear that it would not be good. Any developer familiar with StauderShape may not read this comment because they assume they know what this method does. This is a bug waiting to repeatedly happen. This is not maintainable.

Each class in the inheritance hierarchy has now been considered individually and in conjunction with its superclass. Certain design defects within the current structure have been enumerated, though by no means have all of the defects been listed. The unifying theme with all of the design defects discussed up until this point is that they assume that the inheritance structure is static and unchangeable. While it would be difficult to modify the inheritance structure, it is valid – and even imperative – to ask if the design could be improved through a restructuring of the inheritance hierarchy.

When stepping back from the inheritance structure of the system, the first thing to consider is whether or not inheritance is the correct relationship for each class involved. The classic question to ask which determines if an inheritance relationship is appropriate is the "is a" question [Rumbaugh 91]. In our earliest design examples in this section, we could clearly say that Circles, Triangles, Squares, and Pentagons were Shapes.

When the "is a" question is asked of the inheritance hierarchy discussed in this section, the answers are no.

StauderShape is a GeoPoint? No, it is not. It is not sensible for a shape to be a point. A point is defined as a circle with a zero radius. A point consumes no area and has no shape.

Target is a GeoPoint? No, it is not. As it is presently defined it is not sensible for a target to be a single point. The Target class contains a number of shape definitions within it to support Targets with a shape. Since a point has no shape, this is not sensible.

Target is a StauderShape? As StauderShape is defined in this system, it is not sensible. The same reasoning that prevents Target from being a GeoPoint prevents it from being a StauderShape – to be a StauderShape is to be a GeoPoint.

Target is only one of the fourteen classes that subclass from StauderShape. This same exercise could be done with all of the subclasses of StauderShape and the results would similar.

One of the major drawbacks of the present design is that it is not flexible in the places where it should be. With limited flexibility, the system becomes less extensible. For example, it is not clear how a user of the system could create a Target without knowing where it is since a Target is a GeoPoint. Since a Target is a GeoPoint, it is not possible to create a Target without one.

Another restriction imposed by this design is that if the requirement of representing moving targets was added to THS(X), it would cause a change to the existing system design at a very high level.

This means that the whole of the THS(X) application functionality would be at risk of changing. To clarify the risk, consider achieving the goal by enhancing the definition of a location to add in a time reference, a direction, and a speed. With these three pieces of data, the position of the Target could be computed at some time in the future. Using the present design, this approach is high risk for a number of reasons. First of all, it would expand the responsibilities of the already overloaded GeoPoint class. This would further reduce the maintainability of the GeoPoint class. Secondly, since GeoPoint is the parent of many classes in this system, changes to it would require retesting of many derived classes. For each change to GeoPoint, at least 15 classes must be retested – all of the derived classes of GeoPoint.

The conclusion is that inheritance is not the appropriate structural relationship for the classes in this discussion.

Fortunately, inheritance is not the only design tool available to a system architect. Another powerful tool for the architect is composition. Similar to inheritance, there is a test question that can be used to identify relationships appropriate to composition: "has a." The question largely explains the concept; a composition relationship is appropriate if one of the related classes in question is not a refined type of the other, but rather contains the other class.

For example, a car is not a tire; however, it does have tires; in the present design of THS(X), we would say that a Target is a location, instead of a target having a location.

Because of this we can say that a class representing a common automobile should not inherit from a tire class, but should instead contain a 4 instances of the tire class.

To continue in the problem domain of THS(X), consider Figure 16. This figure contains another way to design this system. In this design, the number of classes has increased from the present THS(X) design, and the functionality supported is a subset; however, this design is more extensible and maintainable than the present design.

In this design the basic concept of GeoPoint still exists, but its role in the system is dramatically clarified. This new embodiment of the GeoPoint concept is called GeoLocation. In this design GeoLocation is a location and a precision estimate for that location. The concept of a GeoLocation is a robust one. A location and the conversions between different coordinate systems is a valuable encapsulated unit of functionality. In this new design GeoLocation does not support the litany of functionality that is present in the present THS(X) GeoPoint design.

Class Shape is new to the discussion of THS(X), but brings in the design examples introduced early in this section. A geometric shape is a clearly defined concept and so a good abstraction. The shapes that THS(X) cares about are easily subclassed from the generic Shape class and are easily implemented and tested. This definition of the Shape inheritance tree is not complete; however, as previously discussed this design is robust to extension and so need not be fully defined initially.

The last class to discuss in this new design is Class Target. Target is another class that comes forward from the present THS(X) design. The new class is different in a number of important ways.

Class Target has no superclass. The concept of a Target is again robust and is something that is quite capable of standing alone. In the present THS(X) design the major systemic design objections were due to the fact that a Target was a shape and was a point. In this design, the definition of a Target is not intrinsically bound up in definitions of the other concepts.

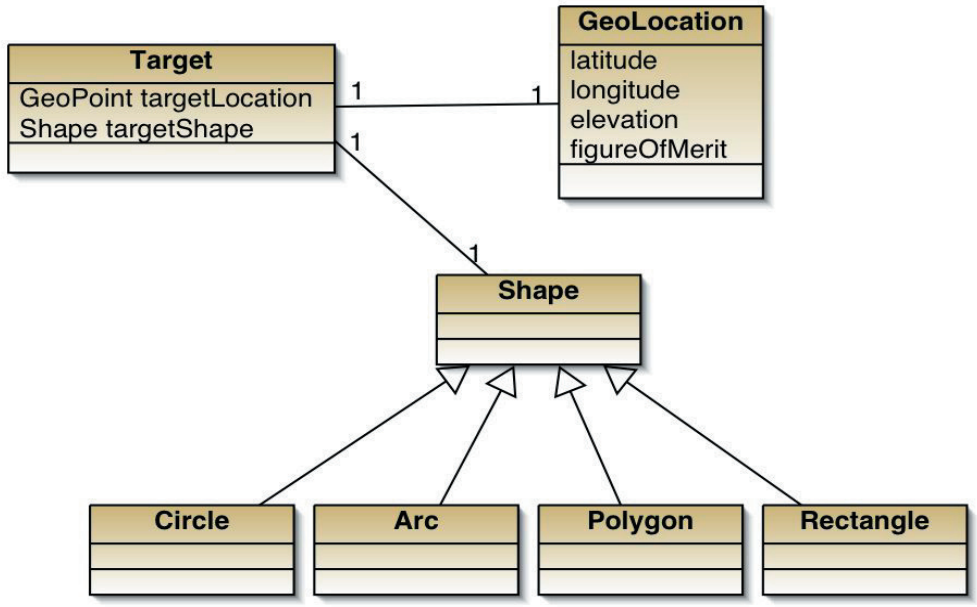


Figure 16: *New design concept for THS(X)*

Class Target, while no longer inheriting from GeoLocation and Shape, still supports that functionality. Looking in Figure 16, a new piece of diagrammatic notation is present. The lines between Target and GeoLocation, and Target and Shape are lines that represent composition, or containment. Each of these lines has a number at each end. These numbers indicate the multiplicity of the relationship. In this case Target contains one GeoLocation and Target contains one Shape. The multiplicity of these relationships can be considered separately. The main issue is that a Target no longer is defined as being a place and a shape – a target is now a target that is at a place and has a shape. This distinction may not seem initially important, but it makes the system far more extensible, testable, and maintainable. It is quite simple in this design to represent a target that presently does not have a known location.

With this new design for THS(X) proposed, consider again the extension of THS(X) to include moving targets. In this design, a target's definition is independent of the definition of a location; this allows this extension to be made in a number of different ways.

One option is to extend GeoLocation to add in a heading, speed, and time. The GeoLocation could then report out an expected position at any time, and could internally have a technique for estimating the positioning uncertainty. If the object that is moving was seen 1 hour ago, the set of possible locations for the object is an area rather than a specific point. This option is attractive because it allows the concept of a target to remain unmodified.

Another option is to add the knowledge of motion to the target class. This option is attractive because is keeps the purity of the GeoLocation intact – it remains a known location on the planet with a known positioning error. This also has the advantage of having the knowledge of the motion of a target be bound up within the target class that it represents. This is an excellent choice for encapsulation.

Good design is as much an art form as a science [Knuth 74] . There are virtually countless ways to extend the new THS(X) design to support moving targets. The design in Figure 17 is one is a good compromise between the two options discussed above, and is a strong design.

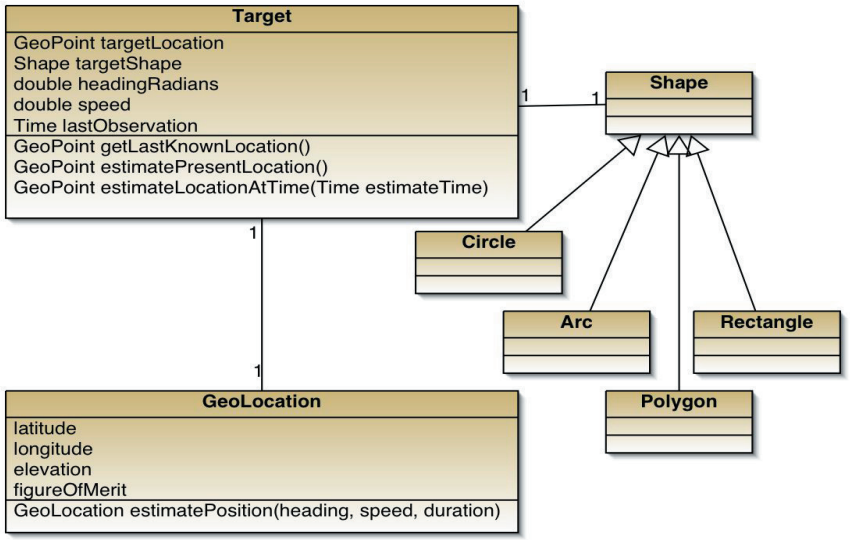


Figure 17: *New design concept for THS(X) with moving targets*

In this design a combination of the two initial strategies is used. Some of the functionality of motion belongs to GeoLocation. GeoLocation should be the object in the system that understands position and uncertainty. As such, this class has the estimatePosition method added to it. This method takes heading, speed and duration as arguments and returns an estimate of current position. This is the simplest form of this method and returns only a single point. This class could also easily add a method that returns a shape that would be the area that the target will be within. One could even add more advanced features to return probability regions so that the consumer can understand how likely the target is to be in any single place.

With GeoLocation supporting the methods to compute where the target is, it is left to Target to maintain knowledge about the target itself. This is appropriate since the heading, speed, and observation time is a property of the target, not of the location where the target was last seen. In addition to keeping the basic information about motion, Target adds three methods for the consumer of this class. These methods allow the user to find out where the target was last seen, where the target is likely to be now (using the last known heading, speed, and time), and where the target is likely to be at a specified time using GeoLocations estimatePosition method along with the

target's data. If GeoLocation is extended to be able to return more advanced types of position estimates, Target could elect to expose those to the consumers of this class as well. These properties show that this design is extensible.

This proposed system design for THS(X) is maintainable. Each class that has been defined in this design has a simple job to do. This packaging of concepts into simple pieces makes it easy for a maintainer to understand the system as a whole. If the maintainer can understand the system as a whole then the maintenance task will be better executed and the system will be viable in the long term.

This design is testable for much the same reason that it is maintainable. The classes in this system are decoupled are have simple definitions. This makes the system design accessible to the developer and the system tester. In addition to the simplicity of each class's job, the classes are also decoupled. By saying that the classes are decoupled, we are saying that they are not interdependent. The proof of this is that both Target and GeoLocation could be easily extended to support moving targets. Not only could they both be extended, the risks of doing so are local to the classes themselves. The change in Target cannot cause the implementation of GeoLocation fail, and vice versa.

Summary:

The present design of THS(X) is fatally flawed. The system is not extensible, portable, or maintainable. The architectural defects in this system are so inextricably intertwined in the core of the system that simple fixes are not an option. A new design concept, based on the use of composition instead of the exclusive use of inheritance, is required for this system.

Fatal Defect #2: Testability

Issue #2: The pervasive use of **friend** in the core object model of THS(X) shatters encapsulation and exponentially increases unit test cases.

Until now, we have not discussed the mechanics of implementation in a language. At this point a small amount of knowledge on the mechanics of implementation is required so that the issues of friend classes can be explored.

The point of a Class, in any language, is to tightly bind together data and the operations on that data. In C++ there are three levels of protection for data and methods in a class. Some parts of the implementation are exposed to entities outside of the class; these parts are considered "**public**." At the other end of the spectrum there is data that is not exposed to any entity outside of the class; it is considered "**private**." Private data is hidden from the outside world but is accessible to all of the methods in the class. This allows for the class to hide data inside itself and protect it by controlling access through use of it's methods. This data hiding is important because it allows for all of the knowledge about proper manipulation of data to be centrally located. Lastly, there is a type of data in the class that is not private but to which access is restricted to direct subclasses; this data is considered "protected." These levels of protection are used in C++ and provide the ability to define strong interfaces for classes.

Figure 18: *Sample of friend*

The friend construct in C++ is used to remove the access restriction put into place between classes. In Figure 18 there are two classes: Target and TargetManager. Between these classes is an association labeled **"friend."** The arrow in the association points from the left to the right. This diagram is showing that Class Target has declared TargetManager to be a friend class in its definition.

The risks introduced when declaring a class a **friend** are significant. In the Classes considered here, it is the case that TargetManager now has completely unrestricted access to the internal implementation of Target. This is bad for a number of reasons.

One reason that this is bad is that it removes the decoupling between these two classes. Now TargetManager has unrestricted access to the Target Class. Because it has this level of access, it is not possible to know what effect changes in Target will have on TargetManager.

Another reason that this is bad is that it will be very difficult to test the Target Class. The reason that it will be so difficult to test is that it is not possible to make statements about how Target should behave without considering the behavior of TargetManager

The inventor of the C++ language, Bjarne Stroustrup, says the following about the use of the friend declaration:

"When you define either a class that does not implement either a mathematical entity like a matrix or a complex number or a low-level type such as a linked list:

[a] Don't use global data (use members).
[b] Don't use global functions.
[c] Don't use public data members.
[d] Don't use friends, except to avoid [a] or [c]." **[Stroustrup 97]**

The author is saying that the use of friend is bad except in certain limited cases. We have not yet encountered a design problem that requires violating encapsulation with the use of friend; there is always another way.

Because of the use of the friend construct, Target and TargetManager are no longer decoupled and must be considered as a single unit. If wholesale violation of encapsulation is built into the design nothing good will happen – the software will not be maintainable or testable.

In Figure 19, a piece of the network of friends in THS(X) is shown as a directed graph.* This graph forces us to ask the following question, can the friend of a friend affect the original class? Essentially, we need to know if the friend declaration has the property of being transitive.+ To consider a specific example, we must find out if a change in MapHandlerObject can affect GeoPoint through GeoPointList.

The answer to the question is yes, MapHandlerObject can affect GeoPoint through GeoPointList. Worse, this property chains indefinitely. MapHandler can also affect GeoPoint through MapHandlerObject through GeoPointList.

This use of *friend* is extremely ill advised, is extremely poor design, and greatly increases the application's complexity.

This use of *friend* is devastating to the testability of this system.

Earlier we discovered that any change to GeoPoint would require the test of at least fifteen derived subclasses. With the knowledge of the implications of the use of friend, we now find that changes to any of eight classes (including but certainly not limited to GeoPoint) will force retesting of the fifteen subclasses.

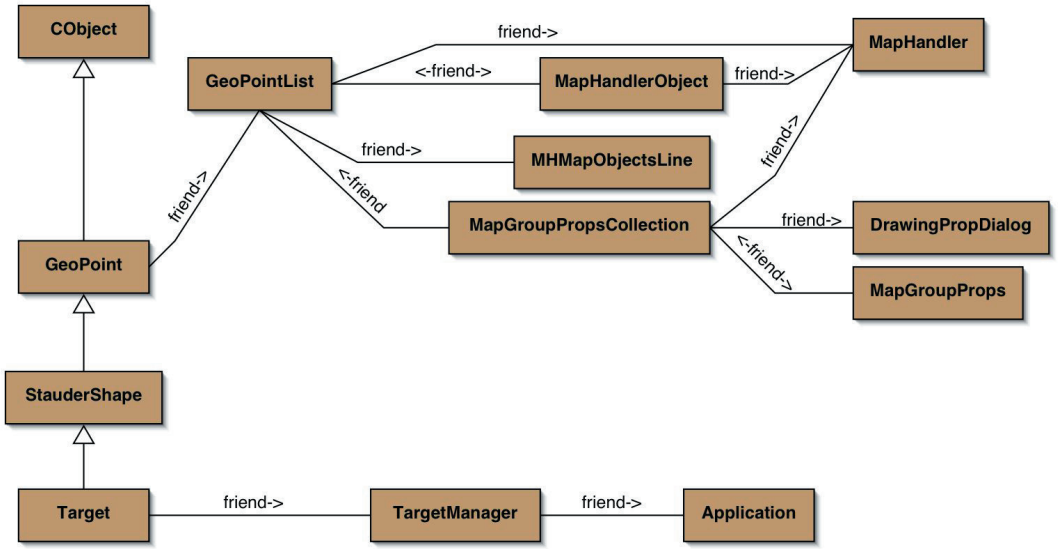


Figure 19*: Network of friend declarations in Target Inheritance hierarchy*

Additionally, we find that to completely test Target we must discover the interfaces to all of its parent classes, all of its friend classes, and at least all of its parent's friend classes. In the case of Target this means that to effectively test 13 classes must be considered. The exact number of tests that must be performed will depend on the topology of the graph of inheritance and friend relationships; however, we can state that the number of tests that are required will grow in a highly non-linear way.
This type of structure is so large as to not be testable.

Summary:

The present design of THS(X) is fatally flawed. It is not testable because of time and cost issues. It is not extensible because of poorly structured inheritance. It is not maintainable because the design of the system depends on heavy coupling between classes and poor inheritance hierarchies. This renders the system so complex that the maintainer would have to make a heavy investment of time to come to a reasonable level of understanding.

* The directed graph of friend relationships in Figure 19 is a subset of the total set of friend relationships in the application. We included only the set of relationships relevant to the THS(X) classes explored in this document in this figure.

+ The answer to this question in the most literal sense is no, friend declarations are not transitive in the C++ language. This lack of transitivity means that in this example MapHandler does not have direct access to GeoPoint; however, it can affect GeoPoint through use of methods and modifications of attributes in GeoPointList.

Fatal Defect #3: Reliability

Issue #3: The use of dynamic linking in THS(X) results in limitations on portability and reliability.

Systems are designed on many levels. In the previous sections, we considered the highest level of design: the THS(X) Object Model. At the level of the Object Model, the selection of Classes and the relationships between those classes are explored, discovered, and documented. Once the Object Model is created, the work of implementing the design into a concrete system can begin.

A critical step in the implementation process is deciding which techniques and technologies will be most beneficial to the project. Different projects have different needs and no single technology is the correct answer to every problem. The correct selection of technique and technology will greatly reduce the complexity and increase the reliability of an application.

The selection of techniques and technologies for use in the construction of THS(X) yielded decisions to use a technique, and the three technologies that the technique implies, which we believe was a poor choice given the requirements of this system. For example, the requirements for successful operations (SRS 3.7.1), graceful degradation (SRS 3.7.6) and security

(SRS 3.5.1) are all put at risk. The technique which we do not believe wise is dynamic linking – the technologies which are used in THS(X) to support this choice are DLLs, Microsoft's Component Object Model (COM), and the Windows Registry.

DLLs:Libraries:

It is not uncommon to measure software source code by the hundreds of thousands of lines. These large bodies of source code are not all intrinsically related in a good software design. To manage these large source bases, they will often be broken up into related groups of functionality (sometimes using an OO design as a guide). The groups of functionality are then placed into units called libraries. The benefit of a software library is that each one is compiled separately and then linked together.

A library must have an interface through which the executable program can access needed functionality.* The simplest type of interface for a library is a re-use of class interfaces. To use an object in the library, the executable code needs to include the interface from that class, just as it would for any other class. This interface gives the executable the correct syntax for accessing the library, but the library must also be linked with the rest of the program during the linking process to provide its binary implementation.

The following diagram demonstrates a linker combining objects from libraries with an executable [Levine 00].

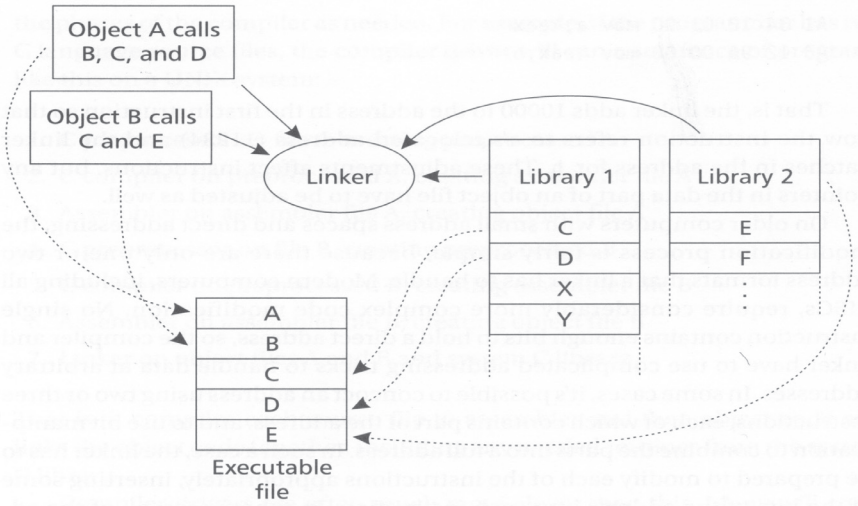


Figure 20: *Linking of Object Libraries into Executables*

* For the purposes of this discussion, we will assume that libraries contain classes; this is not always the case, but what is actually contained in the library is not critical to the discussion.

Libraries can be linked into an executable statically or dynamically, which defines how the functionality in the library is accessed by the executable. Libraries which are statically linked are directly included in the executable at compile time. Any discrepancies found in the linking process produce an error or warning, and the executable is not created. Libraries that are dynamically linked are not included in the executable at compile time. Instead these libraries are linked (the term is loaded) at runtime; this allows the library to be independent of the executable using that library. Thus, the success or failure of the compilation of the executable project when linking a library dynamically is not dependent on the actual structure of the compiled library. As long as the compilation system has the correct header files that show what the interface to the needed dynamically linked library classes look like, the compiler will assume that the binary form of the dynamically linked library will be available, and will exactly match the format given in the header file.

Extensibility

One advantage to the use of dynamic linking in a large project is that changes can be made to the library without re-linking the executable. As long as the interfaces remain the same, the executable project does not care about changes to the implementation within the dynamic library. A new version of a library can be

put onto the system, and if it has the same interface no other changes are necessary. This separation of implementation from interface is especially helpful when updates need to be distributed to end users or to other developers using the library, so that only a single file needs to be replaced, without recompiling and redistributing an entire program.

Dynamic linking is not a panacea. There can be advantages to designing systems that make heavy use of dynamic linking; however, there are also risks that come with its use.

Testability:

Dynamic linking reduces the testability of a software system.

Static linking of executables allows the compiler to check the use of a library object, and produce an error if linking fails. If dynamic linking is used, however, instead of producing an error for the developer, the program will fail for the user, in unpredictable and potentially disastrous ways. When a program tries to access an object that does not link correctly, it may end up accessing some other part of memory. The end result of the failure depends not only on what data the program was attempting to access, but also on what data was accessed instead. Often the behavior of the application will be nondeterministic and hence, unpredictable.

Other data could be affected which was not associated with the library at all. One of the major drawbacks of dynamic linking is that problems will not be discovered until the desired object is accessed at runtime, and the reason for the failure will not be clear. For a statically linked library, this sort of failure would be caught by the compiler and will allow the developer to fix the problem before the user ever sees the system.

Testing an executable that statically links all of its libraries can be viewed as a well-defined exercise. A statically linked executable carries all of the information and functionality that it needs to work within it. This makes the application more testable because there are fewer outside dependencies to consider. An executable that makes use of dynamic linking is much harder to test. Not only does the executable file have to be tracked, but all of the dynamically linked libraries must also be tracked. This makes the testing of a dynamically linked program much more complex.

Reliability:

Dynamic linking makes systems less reliable.

One way in which dynamic linking reduces reliability is that a new version of a library can be put into the system with a different interface. When this happens the executable will fail when it tries to access one of the members of the original interface. It may try to access a method that no longer exists. It may allocate less memory than the new version of an object needs. No matter the details of the failure, some form of access failure will occur. This causes the program to crash, possibly after corrupting unrelated data.

A second way that dynamic linking reduces reliability is that while the interface to a library may not change; the implementation of the functionality within the library may change. This introduces a second type of defect at runtime which may be more devastating than the application exiting unexpectedly – the application behaving in an unexpected way. A simple example of this would be a developer changing the units of a function's return value from feet to meters. If a developer changes the units from feet to meters and the method in question is used to send adjustments to fires then the fires will move far more than expected and safety will be compromised. To mitigate these kinds of risks, the amount of additional testing that is required is significant. With static linking there is no way that an application can use an unexpected version of a library function because it is embedded in the application.

A last problem with dynamic linking is that a library file can be deleted, moved, renamed, or corrupted, and the program will still expect to find it in the same place with the same name. Trying to access an object in a library that doesn't exist results in an access failure and an executable crash for the user. In this way, the end user is exposed to risks that are easily eliminated by statically linking libraries into executables.

Microsoft's Component Object Model (COM):

If the decision is made to use dynamic linking, one way to minimize the problems of dynamic library access on a Microsoft Windows system is to use COM (Component Object Model)[Knox 98]. An executable which uses COM to access a piece of dynamically linked functionality is called a COM client, and a library which exposes a COM interface to be used by an executable is called a COM server. If access to the dynamically linked library is done through COM interfaces, then when the client program tries to access a server library component it can use COM methods to check for the proper location of the library file and to check that the version of the library it is trying to use is the right one. However, this does not solve all of the aforementioned problems, and introduces some new ones.

COM ensures that the client executable is accessing the correct dynamically linked library by using universally unique identifiers to refer to the class that provides the functionality desired. Therefore, the COM client does not have to worry about where a dynamically linked library is installed, or what versions of the library are available on the system. This approach supports extensibility, by assigning new identifiers to newer versions of the library components, so the COM client can get the version it wants.
It supports some amount of maintainability by allowing support for multiple versions of libraries without catastrophic system crashes.

COM adds another level of encapsulation to an object, so that the interface it exposes is a COM interface. This prevents the problem of an executable assuming that a library still supports a specific interface, and trying to access a non-existent method. Instead, the client will ask COM for an interface, and COM can warn the client that the correct interface does not exist, reducing the chance for access violations and hence non-deterministic behavior. This does not eliminate the risk of failure caused by a change to a required library's implementation.

As a technology, COM is capable of doing much more than dynamic library management. It allows binaries developed by different vendors, using different compilers and different languages, to interact in the same process, across processes, or even on different machines across networks.

One example of proper use of COM is when you open up MS Word and want to import data from an MS Excel spreadsheet to a Word document. Word must get data from Excel, in a format which both can understand. With COM, Word can access the components of Excel necessary to interpret the Excel's data by opening Excel with COM. Word is the COM client, and Excel is the COM server. This is a good use of COM as a communication tool between two separate programs, with distinct roles supporting encapsulation.

The following statement and diagram (Figure 21) from the Microsoft developer's webpage [URL 08] explain the basic purpose of COM.

"The Component Object Model (COM) is a component software architecture that allows applications and systems to be built from components supplied by different software vendors."

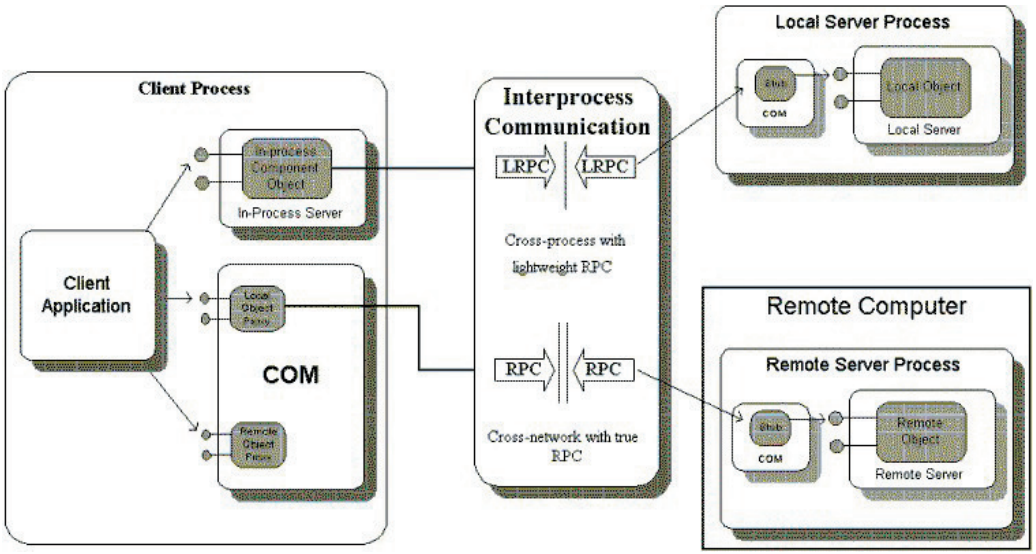


Figure 21: *A single COM client working with in-process, out-of-process, and remote servers*

Extensibility:

COM cannot protect an application from seeking an interface that does not exist. If the expected interface no longer exists, the program still cannot use the object. The most this allows the program is the chance to fail gracefully – which is no small thing. If the object is carefully developed to always support earlier versions, the desired interface will still exist and the object can be used. This form of development leads to an accumulation of old code that is maintained solely so that programs relying on the old versions of a library do not fail. This means that the amount of code that must be maintained steadily rises through the life cycle of the application. As these antiquated versions accumulate, the system becomes less and less maintainable.

If COM returns an interface for the program to use, in theory that interface will match what the program expects. Though an interface should not change, it can. If it does, it must be renamed or assigned a unique identifier such that a request for the old interface will fail, but this is not always enforced. Moreover, COM can be misused in multiple ways that will violate the separation of interface and implementation. If that happens, and the implementation changes, using COM has not protected the program in any way.

Portability:

COM is a Microsoft Windows specific solution to managing dynamically linked functionality. While it is portable across many Microsoft platforms, it is not supported elsewhere.

Designing an entire project around the use of COM means that the project will have to be completely restructured if it ever needs to be ported to another platform, or if Microsoft ever stops supporting COM.

In THS(X), the use of COM is pervasive. We have been told that COM is an intrinsic part of an injector's interface with C2PC. As such, COM is unavoidable in THS(X); however, COM is used in THS(X) to do far more than interface with C2PC.

THS(X) is packaged in a number of different dynamically linked libraries. Inside of THS(X), COM is used as the fundamental way in which many, though not all, objects interact. Therefore, at a relatively high level of design, this system is relying on a non-portable technology. A project designed in this manner cannot be made portable by changing a few key COM based interfaces; the choice of technology renders the entire THS(X) system non-portable.

Reliability:

As outlined previously, COM addresses some of the problems of reliability introduced by the use of dynamic linking; however, it does not solve these problems – it just exposes them in a different way. In addition to not solving the problems introduced by dynamic linking, COM introduces additional dependencies on the Windows Registry – which introduces other risks.

There are certain classes of applications where COM is a valuable resource to the systems designer. For a piece of software like Microsoft Word, which wants to be universally utilized, COM can be quite useful, but it does not lend itself well to critical applications.

The pervasive use of COM within THS(X) is not appropriate.

COM is used heavily in the parts of the system that send and receive messages. Each message that can be sent or received is a COM object. Each field in each message is a COM object. The risks introduced by the use of COM at this critical place in the application make COM an inappropriate choice.

The Registry:

Microsoft COM and dynamically linked libraries make use of the Windows registry. The registry stores information on the location of each dynamically linked library, DLL, and its contents. The registry is accessible to all applications running on a single machine.

The Windows Registry can be thought of as a personal address book. Each computer running Windows has a single address book. All of the applications on the computer share the address book. There are two interesting features of the Windows Registry. The first feature of the registry is that, as an address book, it gives applications the ability to look up the locations of resources (dynamically linked libraries, COM objects, and similar objects) through the use of a "name." The second feature is that since all applications on the computer share a single personal address book, if two applications want to use the same "name" to access a resource, whoever writes it last "wins." Winning in this case is getting the expected results when the Registry is used to look up the value associated with the "name." There is nothing in this scenario that limits the number of applications in contention for a "name" resource to two.

Reliability:

As one might imagine, using the registry to store data can be a risky proposition. If a registry entry is overwritten, which someone will likely view as the entry being corrupted, any number of outcomes are possible. At best, the resource being sought, a DLL or COM object for example, will be lost. Worse, the application may be given information which will result in the executable getting a different version of the resource it desires. At worst, the registry will give the caller the name of an unrelated or non-existent file. The program may attempt to access anything on the system, with potentially disastrous results.

The registry information is set when the program is installed. Thus, the installation program must have all the correct data on the name, location and contents of each resource registered with the Registry. If any of this information changes after installation, for example if a file is moved or renamed, the program can fail as discussed in the earlier sections on Dynamic Linking, and COM. Additionally, the existence of information on the contents of the file allows for the structure of the program to be exposed to the outside world. Thus, the use of the registry violates reliability and encapsulation, and risks exposure of a significant amount of information about the application. While the registry can support some amount of security, the registry key which is used in ATL COM (HKEY_CLASSES_ROOT) is one which is shared between all users in Windows NT. [URL 09]

Portability:

The use of the Windows registry is inherently not portable. THS(X)'s use of the registry in support of COM and dynamic linking is required. Unlike the use of the registry for these functions, THS(X) appears to be capable of storing data in the registry.

In the CommCenter, simulation network, and network code, THS(X) makes use of an ATL object called CRegKey to access the registry. ATL, the Active Template Language, is an extension of COM. This Class contains a method which allows the program to obtain a registry entry, and to query the entry for a string value. In the case of CommCenter, the program is using the ATL object to get the path to a database. This is not traditional COM, but a registry access based on a combination of data that was registered by ATL for COM, and data that was registered elsewhere. The code which uses ATL to query the registry for this information follows:

```
CRegKey objRegKey;

if(objRegKey.Open(  HKEY_CLASSES_
ROOT,
"\\DigitalComms.CommCenter",
KEY_READ) == ERROR_SUCCESS)

{LPTSTR lpDatabaseHost =
csDatabaseHost.GetBuffer(50);

ULONG lChars = 50;
if(objRegKey.QueryStringValue(
"DbHostName",lpDatabaseHost,&lChars) !=
ERROR_SUCCESS)
```

In addition to the fact that application data is being stored in the registry, the Microsoft Developer Network website has the following information about the ATL method being used to access the registry:

*"**Security Note** This method allows the caller to specify any registry location, potentially reading data which cannot be trusted."* [URL 10]

Summary:

A high level decision was made in THS(X) to use dynamic linking. In this section the technique of dynamic linking and the technologies that were used to accomplish the implementation were reviewed. To support dynamic linking three Microsoft specific technologies were employed: Dynamically Linked Libraries (DLLs), Microsoft's Component Object Model, and the Windows Registry.

There are times and places where dynamic linking is unavoidable: the interface to C2PC, the interaction of applications with the Operating System core services. Whenever possible dynamic linking should be avoided, especially in applications involving fires because it introduces unacceptable risk of failure to the system.

The use of COM in the messaging system is an extremely unwise decision.

THS(X) uses dynamic linking and COM in situations where they are not appropriate. With this implementation choice, it is possible that the forward observer, forward air controller, or naval gun spotter will take a system into the field and not find out that the system is inoperable until they try to send the first message. These types of problems are completely avoidable through the use of static linking.

Fatal Defect #4: Maintainability

The documentation for the THS(X) system is inadequate and lacks fidelity.

One critical issue with the documentation is that of relevance. When asked to deliver the documentation that goes with the source code and executables that CMU evaluated, Stauder provided the following documents:

Document Name	Version	Date
THS(X) Software Requirements Specification (SRS)	Spiral 1	30-Apr-03
THS(X) Software Requirements Document (SRD)	Spiral 1	30-Apr-03
THS(X) Software Design Document (SDD)	Spiral 1	7-Sep-03
THS(X) Software Architecture Design Document (SADD)	Spiral 1	30-Apr-03

There was some initial confusion regarding the documentation for two reasons. The documents are all labeled "Spiral 1" and all of the other materials delivered for review were labeled "Spiral 2." We sought clarification to make sure that we had received the most recent documentation for the system. On 25Feb2004, Stauder confirmed that the documents that we have are the most recent. [Muizers, 25March2004] In addition to the documentation being labeled "Spiral 1," the documents were as much as 10 months out of date with the implementation at the time that our analysis began.

The risk of letting documents age is that they become less relevant. As the documents become less relevant, they are less likely to be consulted. If they are not consulted they will not be updated. This is a self-perpetuating downward spiral for documentation. This is unacceptable in a production environment – documentation is one of the keys to making a system maintainable and testable. From maintainability and testability comes reliability.

Another issue with the delivered documentation is that the SRD, or Software Requirements Document, addresses more than just requirements. In this document certain design decisions are documented. From the SRD:

> ***3.5.3.1. User Accountability (SRS 3.5.3.1)***
>
> The application will log entries in a database table of all security relevant actions ***performed by THS(X) system. All incoming and outgoing messages and their content will be tracked. The administrator will be able to view the messages by user id. THS(X) will use C2PCs accountability tools to track user accountability.***

This segment of text from the SRD discusses the implementation of the solution – the use of a database, and the use of C2PC. This is not a requirement; this is a design decision. As a requirement it would be more useful if it clearly defined "*all security relevant actions.*" The question becomes, does the SRS, Software Requirements Specification, mandate that these techniques be used? If so, then this is a requirement. Otherwise it is a design decision. Section 3.5.3.1 of the SRD refers back to section 3.5.3.1 of the SRS. From the SRS:

> ***3.5.3.1. User Accountability (7)***
>
> *An audit mechanism shall be implemented to ensure that all security relevant actions performed on THS(X) are traced to the user or process performing the action.*

The SRS does not specify what technologies are to be used to accomplish the goals of auditing for the purposes of User Accountability. As such, either the SRS is badly restating the requirement

from the source document that it references, "TLDHS Security Requirements Traceability Matrix," or a design detail is recorded in a requirements document. In either case, this is a significant problem that will affect maintainability and testability because the designers, implementers, testers, and maintainers do not have good requirements documentation to work with.

In addition to design information being encoded in the SRD, there are requirements in the SRD that are not addressed in the SADD, Software Architecture Design Document. The SRD calls for a scriptable simulation system to be implemented for training purposes.

3.2.3.5. Simulation (SRS 3.2.22)

A simulation mode will be available for training purposes. While simulation mode is active, THS(X) will simulate digital communications. A script will be used to define the behavior of the simulation.

The requirement is for the simulation to be scriptable. The documentation suite does not provide nearly enough information on this feature. The SDD, Software Design Document, discusses the ability to simulate networks, but does not even discuss the system to the level of saying that the system will be scriptable. This is addressed in section 5.8 of the SDD.

5.8 Simulation Networks

For every network class defined in the Communication Component that works with a specific message type, there exists a corresponding simulation network class. The simulation networks are designed to work without requiring the system to be connected to the radio network, or even have one of the normally required modems on the system. As the name implies, the simulation networks simulate the communication with other systems. The design goal is to make the client think communication is actually happening on a real network, when it fact the system is not connected to any communication network. When the client asks the network to send a message, the network will send back the appropriate responses to the client to make it appear that the message was successfully sent. It may also generate and send to the client additional messages to make it appear that the fictitious entity the client has sent a message to is responding back with additional (and possibly different) messages.

The SADD discusses simulation in two places:

For each network, there is a corresponding simulation version of that network. This simulation network does not send any actual messages to any system. When it receives a message to be sent, instead of trying to send the message over the modem, it reads a script that specifies one or more messages to send back to the client, as if another system had just sent those messages. This simulation version of the network is used for testing purposes, as well as for demonstrations.

And

3.7.4. Communication Component Testing

A separate application will be developed that uses the communications component. This application emulates communication with other platforms. It allows the user to create and modify all valid types of networks, and send and receive all the supported messages for each network. Messages can be created and then saved for sending later. A script will also be created that will send a list of saved messages.

This application will be used for testing the communications component. Two systems will be set up, both running the same emulator application. Message will be sent from one system to the other, and the data on each will be compared.

In the requirements and design documentation that was delivered with the source, we were unable to find a meaningful disussion of the required simulation system. The documentation does not even discuss the format and location of the scripts which the system is to run. Without knowing how it is implemented, where the input files are, and how the system is to behave, it will be incredibly difficult for a developer to implement this system, a tester to test the system, or a maintainer to maintain this system.

While a large amount of time was not invested in finding this type of defect, it is our belief that this is not the only instance of requirements not being sufficiently addressed (or addressed at all), in the design documentation. There are a number of reasons as to why this may be. The most likely are that the requirements are not addressed, or that they are, but there is not design documentation to support that feature. In any case, these types of defects reduce maintainability and testability. A reduction in maintainability will reduce reliability and dramatically increase the life-cycle cost of the system since the maintainer will be unable to make changes to the system without undertaking a research project to determine the present layout of responsibilities in the system.

There are also cases where the contents of the documents contradict one another. One example of this is the discussion of the structure of the MissionManager hierarchy.

In the SDD, the MissionManagers are addressed as follows in section 4.4.1:

4.4.1.1 Hierarchy Chart

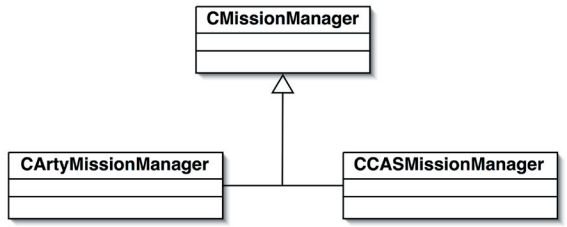


Figure 4-9 Hierarchy Chart for the CMissionManager shows how the specific mission managers relate to the generic mission manager. The generic mission manager provides the base class for the Close Air Support (CAS) Mission Manager and the Artillery (Arty) Mission Manager.

"The mission manager object will be a generic base class for all other missionmanagers to inherit from. Primary responsibilities of the generic mission manager are to create, retrieve, and delete missions, activate and deactivate missions, provide a list of all the currently active missions, provide the number of missions currently under the control of the mission manager, and provide persistence to the database for the individual missions. The current mission managers that will inherit from the generic mission manager are the CAS Mission Manager and the Artillery Mission Manager.
These mission managers will have functionality specific to their needs and will be described in their own sections.

"A generic mission manager object was created as a need to break up the individual missions and give them all the same base class. All of the functionality that is common to the individual mission managers will be defined in the mission manager object and then implemented in the specific mission managers."

Each of the CASMissionManager and ArtyMissionManager classes gets a section that clarifies the role it plays. In section 4.4.3, ArtyMissionManager's definition is expanded and the following text discusses Naval Fires:

"The types of missions the artillery mission manager can manage include the final protective fire missions, artillery registration missions, NSFS missions, and regular artillery missions."

In the SADD, the following text addresses MissionManagers:

"Mission Managers
The THS(X) system will contain a mission manager for the Close Air Support (CAS) missions, Suppression of Enemy Air Defense (SEAD) missions, Artillery missions, and the Naval Gunfire (NGF) missions. These mission managers will all be derived from a base mission manager. Basically, they will be containers which hold the missions and manage them. The management duties will include tasks such as database access, mission retrieval, and mission activation/ deactivation. The managers will be responsible for the business logic needed to support multiple and simultaneous missions. The application object will create exactly one manager object for each type of supported missions.

"CAS Missions
The CAS missions will be managed by their appropriate mission manager and will contain the information pertinent to CAS missions. That information will include an initial point, up to 3 egress points, a mission target, a flight, a laser cone and an attack
cone. Only one CAS mission will be active at a time however, many CAS missions may be planned.

"Artillery Missions
An artillery mission supports a mission carried out using artillery weaponry. Included
in an artillery mission will be a fire plan, round, fuze, method of control, method of engagement, and a fire mission type. The selected round and fuze will determine the fragmentation pattern drawn around the artillery mission's target. An artillery mission
will be managed by the artillery mission manager.

"SEAD Missions
A SEAD mission will contain one CAS mission, as well as one or more artillery missions. The SEAD missions will be managed by the SEAD mission manager.

"Naval Gunfire Missions
A naval gun is an artillery weapon mounted on a ship. Due to that fact, a naval gunfire
mission will be treated in a similar fashion to the artillery mission. The naval gunfire
mission will be managed by a naval gunfire mission manager."

The SADD does not provide a UML diagram of the system; however, the way that the text reads, the expectation is that the system would be structured as follows:

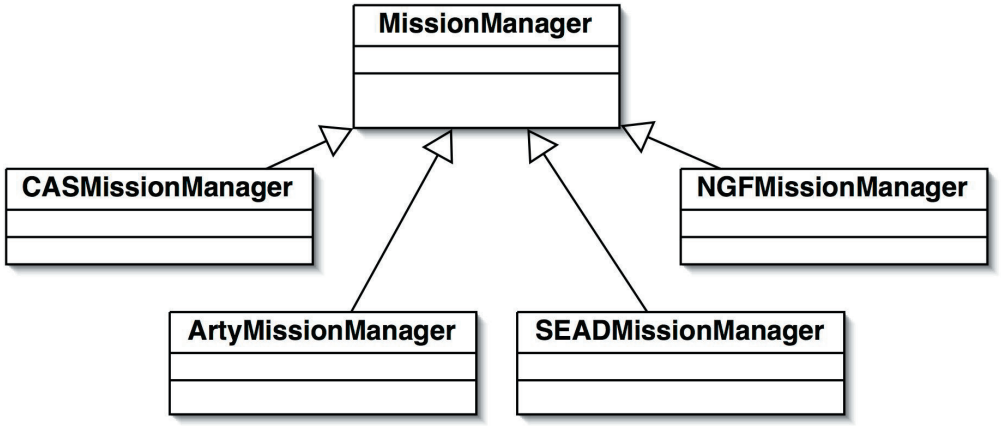


Figure 22: *UML Diagram of the system*

The definitions from the SDD and the SADD do not agree. To resolve this discrepancy, the code must be consulted. When the code is consulted, the design documented in the SDD is shown to be the correct one – the one implied by the SADD is not in evidence. There are no files supporting the existence of an NGFMissionManager or a SEADMissionManager.

In consulting the implementation, it became clear that the documentation in the SDD is far superior to the documentation in the code itself. While it is not the case that every word from the SDD, or SADD should be in the implementation, the class comment for the ArtyMissionManager class should discuss that it was managing NGF missions as well as standard Artillery missions.

The state of the documentation describing and supporting THS(X) is unacceptable. The documents are out of date; they contradict one another, place the wrong information in the wrong files (design decisions are encoded in requirements documents), and are not sufficiently detailed. This documentation will make the jobs of testing, extending, and maintaining THS(X) far more difficult and expensive than they would be with good documentation.

Conclusions & Recommendations

We strongly recommend that no effort be expended attempting to convert the existing THS(X) implementation to Block 3 THS(X). The present design and implementation of THS(X) is not reliable, extensible, maintainable, testable, or portable.

This recommendation is the result of a principled analysis of THS(X) version 3.2.0. The basis for the analysis is built upon industry standards as described in Section 1 of this document, *Foundations of Good Software***.**

Section 2 applies the framework built in Section 1, *Foundations of Good Software***, to THS(X) version 3.2.0 (executables, source code, and supporting documentation) as produced by Stauder Technologies, Inc. In Section 2, Analysis of THS(X) 3.2.0, we detail four fatal pervasive issues relating in THS(X).**

➦ Issue #1: The design of the core object model used in THS(X) is not extensible, portable, or maintainable.
The architectural defects in this system are so inextricably intertwined in the core of the system that simple fixes are not an option. A new design concept, based on the selective use of composition instead of the exclusive use of inheritance, is required for this system.

➦ Issue #2: The pervasive use of **friend** in the core object model of THS(X) shatters encapsulation and exponentially increases unit testing. It is not testable because of the combinatorial expansion of time required to do testing. It is not extensible because of poorly structured inheritance. It is not maintainable because the design of the system depends on heavy coupling between classes and poor inheritance hierarchies. This renders the system so complex that the maintainer would have to make an unreasonable investment of time to come to a necessary level of understanding.

➦ Issue #3: The use of dynamic linking in THS(X) results in reduced portability and reliability.
A high level decision was made in THS(X) to use dynamic linking. There are times and places where dynamic linking is unavoidable: the interface to C2PC, the interaction of applications with the Operating System core services. Whenever possible dynamic linking should be avoided, especially in applications involving fires, because it introduces unacceptable risk of failure to the system. A runtime failure of a dynamically linked system will manifest itself as unexpected application death, or unexpected application behavior. Both of these things are unacceptable in a deployed safety critical system. The problems that dynamic linking introduces are completely avoidable through the use of static linking.

➦ Issue #4: The documentation for the THS(X) system is inadequate and lacks fidelity.
The documentation for THS(X) is internally inconsistent and lacks rigor. The requirements documents encode design and the design documents contradict each other. Timely, relevant, clear, and accurate documentation is required when designing, implementing, testing, and supporting a software system. The documentation for THS(X) is none of these things.

Point Defects:

In addition to the four pervasive defects detailed in Section 2, a large number of point defects were identified during the evaluation. These point defects are cataloged in Appendix C. Each of these defects was evaluated for its impact to the THS(X) system. While the implications of these point defects are quite severe, many of the point defects can be fixed. It is not possible to fix the systemic defects of THS(X) by addressing any number of individual point defects; the problems with THS(X) are pervasive and flow from poor design decisions.

Installation Experiences:

Another part of the evaluation process was an investigation of the installation materials and processes for THS(X). We found the THS(X) 3.2.0 materials insufficient to successfully install the software on either a COTS computer or a Tacter-31, RHC. No amount of support from Stauder could resolve the issues that were encountered.

After unsuccessfully working with the delivered THS(X) 3.2.0 materials with the support of Stauder, we took an RHC to the Stauder office in St. Peters, MO. While the RHC was at Stauder, THS(X) was successfully installed. This installation did not follow the original installation materials provided with the 3.2.0 release. Additionally, the installation required the direct support of engineers on the Stauder staff. This successful installation required a complete reinstallation of the RHC's software including, but not limited, to the system BIOS.

After getting THS(X) installed on one RHC at Stauder, we were able to install THS(X) on a second RHC at CMU. In the process of this installation, we found that even the enhanced installation instructions provided by Stauder while at their facility were incomplete.

The installation process for THS(X) is poorly documented and immature.

Runtime Experiences:

Once THS(X) was successfully installed, a limited investigation of the runtime performance of the system was undertaken. Our initial goal was to create and execute a single CAS mission; we never successfully completed a simulated CAS mission due to THS(X) instabilities.

In the process of trying to create a CAS mission a number of disturbing problems surfaced.

The User Interface for THS(X) is very fragile. It was our experience that the User Interface would not react well to certain inputs. For example, if a user clicked on the "Exit" button on the "Department of Defense Warning Statement" dialog box, THS(X) and C2PC became unresponsive. To remedy the situation, the user had to forcefully terminate the C2PC application using the Windows NT4 Task Manager. Even this was not always sufficient as the user was sometimes unable to access the Task Manager. The remedy in this instance was a hard reset of the RHC.

The THS(X) system consumes excessive amounts of computing resources. At times, the system enters into a mode where it consumes more than 98% of the processor cycles. To recover, the user must exit C2PC and then restart C2PC and THS(X).

Sometimes, THS(X) consumed system memory at the rate of approximately 500 kilobytes per hour. The only solution was to exit C2PC and restart C2PC and THS(X).

In Summary:

After review of THS(X) and its associated documentation, we conclude that the design and implementation of THS(X) are fatally flawed. The design and implementation of THS(X) are not reliable, testable, extensible, maintainable, or portable. If deployed today, THS(X) will fail in operational environments. We strongly recommend a wholesale replacement of THS(X).

Glossary

Abstraction. The essential characteristics of an object that distinguish it from all other kinds of objects, and thus provide, from the viewer's perspective, crisply defined conceptual boundaries; the process of focusing upon the essential characteristics of an object.

Attribute. Data that is encapsulated into an object.

Class. A set of objects that share a common structure and behavior manifested by a set of methods; the set serves as a template from which objects can be created.

Data Protection / Data Hiding. A manifestation of Encapsulation. Data in an object which is not accessible to the rest of the program.

Encapsulation. The process of bundling together attributes and methods into a class or object.

Extensibility. The degree of ease with which a system can be extended to support new functionality.

Decoupling. The separation of unlike elements or functionality.

Friend. A designation which allows the named class to access all the data and methods of the naming class.

Information hiding. The process of hiding the structure of an object and the implementation details of its methods. An object has a public interface and a private representation; these two elements are kept distinct.

Inheritance. A relationship among classes, wherein one class shares the structure or methods defined in one other class (for single inheritance) or in more than one other class (for multiple inheritance).

Instance. An object with specific structure, specific methods, and an identity.

Instantiation. The process of filling in the template of a class to produce a class from which one can create instances.

Interface. The set of methods through which an object communicates with the rest of the system.

Maintainability. The aptitude of a system to undergo repair and evolution

Method. An operation upon a class, defined as part of the declaration of a class.

Model-View-Controller. One of the first Object Oriented programs; now a standard pattern for programs which interact with a user.

Object. The basic element in Object Oriented design. Objects have attributes, methods, and a name.

Object Oriented Design. A design methodology employing encapsulation, abstraction, and inheritance.

Orthogonality. In software engineering, when two things are independent of one another; when changes to one thing, have no effect on the other.

Portability. The degree of ease in producing an executable version of an application on a new platform from an extant executable on another platform.

Private. A designation which assigns a level of protection to an attribute or a method. Private data cannot be accessed by any object except the one in which it resides.

Protected. A designation which assigns a level of protection to an attribute or a method. Protected data can only be accessed by the object in which it resides, and any of that object's subclasses.

Public. A designation which assigns a level of protection to an attribute or a method. Public data is accessible to all parts of a software system.

Reliability. The aptitude of a system to repeatedly produce relevant, quantifiable, and desirable results.

Smalltalk. The first object oriented programming language.

Superclass. The class from which a subclass inherits its attributes and methods.

Subclass. A class that inherits the attributes and methods of a parent class.

Testability. The degree to which the design and implementation of a software system facilitates testing.

References

[Archer 95] Archer, C., & Stinson, M. Object-Oriented Software Measures, Technical Report CMU/SEI-95-TR-002, ESC-TR-95-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: April 1995

[Barbacci 97] Barbacci, M. R., Klein, M. H., & Weinstock, C. B. Principles for Evaluating the Quality Attributes of a Software Architecture, Technical Report CMU/SEI-96-TR-036, ESC-TR-96-136, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: May 1997

[Bass 99] Bass, L., & Kazamn, R. Architecture-Based Development, Technical Report CMU/SEI-99-TR-007, ESC-TR-99-007, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: April 1999

[Batman 93] Batman, Joe "Characteristics of an Organization with Mature Architecture Practices." Essays on Software Architecture, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: December 1993
Beizer, B. Software Testing Techniques Second Edition, Boston, MA: International Thomason Computer Press, 1990

[Boggs 02] Boggs, W., & Boggs, M. UML with Rational Rose 2002, Alameda, CA: SYBEX, Inc., 2002

Box, D. Essential COM, Reading, MA: Addison-Wesley, 1998

[Brown 98] Brown, W.J.; Malveau, R.C.; McCormick, H.W.; & Mowbray, T.J. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, New York: John Wiley & Sons, 1998

Dart, S., Christie, A M., & Brown, A. W. A Case Study in Software Maintenance, Technical Report CMU/SEI-93-TR-8, ESC-TR-93-185, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: June 1993

[Foote 97] Foote, B., & Yoder, J. Big Ball of Mud, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, August 1997

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; & Roberts, D. Refactoring: Improving the Design of Existing Code, Boston, MA: Addison-Wesley, 1999

[Gamma 95] Gamma , E; Helm, R.; Johnson, R.; & Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995

[Hunt 99] Hunt, A.; & Thomas, D. The Pragmatic Programmer: From Journeyman to Master, Boston, MA: Addison-Wesley, 2000

[Kernighan 99] Kernighan, B.; & Pike, R. The Practice of Programming, Reading, MA: Addison-Wesley, 1999

[Knuth 74] Knuth, D. E. "Computer Programming as an Art" Communications of the ACM 17, 12 (December 1974): 667-673

[Levine 00] Levine, J, Linkers & Loaders, San Francisco, CA: Morgan Kaufmann Publishers, 2000
Marick, B. The Craft of Software Testing, Englewood Cliffs, NJ: Prentice Hall, 1997

[McGregor 01] McGregor, J. Testing a Software Product Line, Technical Report CMU/SEI-2001-TR-022, ESC-TR-2001-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA: December 2001

Mooney, J.D. Bringing Portability to the Software Process, Technical Report TR 97-1, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1997.

[Mooney 94] Mooney, J.D. Portability and Reusability: Common Issues and Differences, Technical Report TR 94-2, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1994.

[Nelson 92] Nelson,M. Serial Communications: A C++ Developer's Guide, San Mateo, CA: M&T Publishing, Inc., 1992

Quatrani, T. Visual Modeling with Rational Rose 2002 and UML, Boston, MA: Addison-Wesley, 2003

[Rector 99] Rector, B., & Sells, C. ATL Internals, Reading, MA: Addison-Wesley, 1999

Rogerson, D. Inside Com: Microsoft's Component Object Model, Redmond, WA: Microsoft Press, 1997

[Rumbaugh 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Loernsen, W. Object Oriented Modeling and Design, Englewood Cliffs, NJ: Prentice Hall, 1991

[Stroustrup 00] Stroustrup, B. The C++ Programming Language (Special 3rd Edition), Boston, MA: Addison-Wesley, 2000

[URL 01] Software Portability Home Page
-> http://www.csee.wvu.edu/~jdm/research/portability/home.html
This page is maintained by Jim Mooney, PhD. Department of Computer Science and Electrical Engineering at West Virginia University.

[URL 02] Whatis.com
-> http://whatis.techtarget.com/definition/0,,sid9_gci343038,00.html

Definition of Abstraction

[URL 03] Design Patterns: Model-View-Controller, Sun Microsystems, Inc
-> http://java.sun.com/blueprints/patterns/MVC.html
An explanation of Model-View-Controller

[URL 04] Basic Aspects of Squeak and the Smalltalk-80 Programming Language
-> http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/smalltalk1.html
Course Materials. A history of Smalltalk and the Model-View-Controller paradigm. Associate Professor Wolfgang Kruetzer
Department of Computer Science, University of Canterbury, New Zealand

[URL 05] Applications Programming in Smalltalk-80
-> http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html
Course Materials. Ian Chai, Ph.D
Faculty of Engineering, Multimedia University, Cyberjaya, Malaysia

[URL 06] Dictionary.com
-> http://dictionary.reference.com/search?q=abstraction
Definition of Abstraction, Webster's Revised Unabridged Dictionary, © 1996, 1998 MICRA, Inc

[URL 07] MFC Library Reference
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_cobject.asp
CObject Class: class AFX_NOVTABLE CObject

[URL 08] MFC Library Reference / The Component Object Model: A Technical Overview
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_basicpmd.asp
COM introduction, Microsoft white paper

[URL 09] The Microsoft Developer Network
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/hkey_classes_root_key.asp
About the Registry, HKEY_CLASSES_ROOT key

[URL 10] MFC Library Reference
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/vclrfcregkeyquerystringvalue.asp
ATL Library Reference : An explanation of CregKey

MFC Library Reference

-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/cmf_d2l_48fo.asp
DllGetClassObject
MFC Library Reference
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/cmf_a2c_6yb8.asp
CoGetClassObject

MFC Library Reference
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/cmf_a2c_95rt.asp
CoLoadLibrary

MFC Library Reference
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/cmf_a2c_5ry0.asp
CoCreateInstanceEx

MFC Library Reference / Essential COM quoted on MS web page
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnesscom/html/classesservers.asp
Classes and Servers

MFC Library Reference / Essential COM quoted on MS web page
-> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnesscom/html/optimizations.asp
Optimizations

Java Blueprints J2EE Patterns, Sun Microsystems, Inc
-> http://java.sun.com/blueprints/patterns/MVC-detailed.html
An explanation of Model-View-Controller

Stauder Documentation:

System Architecture Document for THS(X) 30 APR 2003

Software Architecture Design Document for THS(X) 30 APR 2003

Software Design Document 21 FEB 2003

Software Requirements Document 9 DEC 2002

Software Requirements Specification APR 2003

Appendix A: THS(X) 3.2.0 Installation

Index

Overview

As part of the analysis process of THS(X) 3.2.0 we considered installation (e.g. BIOS issues), portability, and maintenance issues relating to the software.

There were two goals for this task. The first was to test the documented installation procedure for THS(X) as provided by Stauder. Regardless of the results of this test, the second goal was to get THS(X) installed and functioning on a machine, preferably an RHC. In the process of executing this second task, some insight was gained into the current maintainability of the system outside of the development environment.

In support of this effort, CMU received equipment and support from Naval Surface Warfare Center – Crane Division, and Stauder Technologies, Inc. The initial delivery of materials for the installation effort follows.

NSWC Crane	two Tacter-31 RHCs,
NSWC Crane	C2PC 5.9.0.3, and C2PC 5.9.0.3 patch 4
Stauder	THS(X) 3.2.0 installation media and instructions
Stauder	a FastIO (HD26) to Ethernet cable to load software onto the RHC

In advance of CMU's attempt at installation, we were told that the THS(X) software had been installed and run on a variety of commodity, non-RHC platforms. We were also told that in the past there had been issues trying to install THS(X) on RHCs. Accordingly, we augmented the hardware set supplied by NSWC Crane with three Shuttle PCs (commodity PCs) so that we could isolate for any problems specific to installation on the RHC. We decided that the addition of these non-RHC platforms to the test would be a good risk mitigation strategy.

Our experiences with the software during the installation task were not favorable. Because the installation documentation as provided by Stauder was inadequate, CMU invested over 60 man-hours of labor in this task before successfully installing the software on an RHC. We have identified a number of issues with the installation instructions provided with the THS(X) 3.2.0 release.

The documentation "Installation Instructions for THS(X).doc" was insufficiently detailed regarding the specifics of the install. When going through the installation procedure as described by the installation instructions document, we found them to be insufficiently detailed regarding the setup of the machine onto which THS(X) was to be installed. The document did not provide any of the following information:

- the version of the operating system to be installed
- the patches required for the operating system
- the version of C2PC to be installed.

We experienced a problem with the lack of information regarding the version of C2PC to install. We installed the most recent version of C2PC, 5.9.0.3 patch 4 as received from NSWC Crane, and had problems. The first thing Stauder requested we do was to install C2PC without Patch 4 because Stauder had never used the patch.

We also discovered that the installation process always issued a dialog box reporting error number 1722 during the THS(X) installation process. We were told that this is totally normal and not of concern. [Gabler, 10Mar2004] This error was not documented so that the user could expect it, even though Stauder expected it in every install.

These are significant issues which complicated the installation process – we were forced us to seek direct assistance from Stauder. In spite of this support, CMU was unable to install the software on either the Shuttle PC or the RHC.

In summary, the first 40 hours of the installation task were spent working with the initial installation instructions, as provided by Stauder. These instructions were implemented as delivered, and also as modified by Stauder through phone and email support. Two distinct modifications to the initial process were tried. CMU made many attempts at installation using the three different versions of the installation processes. After the initial time investment, it was decided that the best course of action was for a CMU representative to travel to the Stauder Technologies office in St. Peters, MO.

Upon arrival at the Stauder office, the initial installation instructions given to CMU were replaced by a 5-page document that was significantly more detailed and relevant to the installation process of the software. The initial installation instructions were not consulted during this installation process. We note that even this expanded replacement set of instructions did not cover all of the situations that were encountered during the installation process at the Stauder office. To address these undocumented situations, engineers were brought in multiple times to debug and assist with the installation process.

Successful installation on the RHC was achieved. Installation was not successful on the generic PC (Shuttle PC).

The detailed documentation of the series of processes that were tried at the Stauder office is located immediately after this document. Please consult the following materials for the details of the installation process if so desired.

After the installation at the Stauder office was completed, the instructions that had been debugged and used on 15March2004 were updated and this corrected version was emailed to CMU. These updated instructions are in the file “16Mar2004 Installation Instructions for WinNT.doc” which is included in the materials that follow this document.

Even the updated version of the instructions sent to us on 16March2004 did not contain all of the steps that were required to successfully install THS(X) on the RHC at either Stauder or CMU. Specifically, the instructions do not indicate that the SanDisk (a brand name for compact flash memory) must be made bootable. Additionally, these updated instructions also still referenced the office servers located at Stauder Technologies. Referencing internal servers greatly diminishes the value of these directions to anyone who is outside of the Stauder office.

In the end we were able to install THS(X) without the direct assistance of Stauder Technologies; however, we were only able to do so using what can only be considered a "golden" process. A "golden" process is one that works, though no one is quite certain why. The main feature of a "golden" process is the lack of certainty as to what makes it distinct from other processes that are attempted. During the execution of this task, CMU attempted to independently recreate the installation environment that Stauder used and failed. At no time was a good installation reproduced without the use of a "golden" process from Stauder. This process included not only wholesale replacement of the operating system installation, but also wholesale replacement of the BIOS on the RHC. Further, when we tried to identify the differences between the environment that CMU generated and the one that Stauder used, Stauder did not discuss the set of packages that constitute the Operating System that they use on the RHC. The proposed solution for the problems encountered with Windows NT 4.0 was to not use it, but rather to use Windows 2000.

Also, the working installation that we created using the Stauder process cannot use the data modem internal to the RHC. While we were told that all it would take to make the system work with the internal modem was a driver update on the machine to recognize the other data modem, this change was not made to the "golden" operating system installation. This is not to say that it could not be done, only that it has neither been done by us, not witnessed by us.

The installation process is immature. Using the initial installation instructions and materials, we were not able to work through the problems we had even with the aid of Stauder phone and email support. To successfully install the software, we had to take machines to the Stauder Technologies office. Stauder used a different version of the installation media than was provided to CMU. Stauder's successful installation was also predicated on following a different process than the one that was detailed in their installation instructions to CMU. The fact that we had to reinstall not only the operating system, but also the system BIOS as part of the successful installation process is troubling. This type of issue speaks to low level of process maturity and a development and release environment that is producing prototypes..

Muizers, 02 Dec 2003

From: Muizers Charles M CONT CNIN
Sent: Tuesday, December 02, 2003 2:09 PM
To: Gardner Robin D (Dale) CNIN
Cc: 'jstauder@staudertech.com'
Subject: RE: Request for THS(X) Information in Support of the C2PC Trade Study

Dale,

I spoke with Jerry Stauder, Stauder Technologies today regarding the information required for the C2PC Trade Study. Jerry is very enthusiastic and wants to support this effort.

Stauder requires an NDA (Non-Disclosure Agreement) be in place with CMU in order to provide some of the information. This is pretty standard. Stauder will inititiate this by providing a Draft to you. Spoke with Dr. Thayer. Upon receipt of the NDA, he will provide it to the Provost for approval. If the Provost will approve, then we are cleared hot. In the event that the Provost wants changes, then CMU legal will need to get involved...but we'll cross that bridge when we get there.

As the government Technical POC for the CMU contract, I have asked Jerry Stauder to deal directly with you on this. He will provide the information to you and you can provide to CMU.

Please email Jerry with your contact information and we can get this started.

One thing that needs to be clarified is of the list of requiredinformation (pasted below), what elements can be provided now and what elements will fall under the NDA. At first glance, I would think that items 3-10 could be furnished without an NDA. Regarding items 11, this will require a code drop from Stauder. Item 12, I don't know if we can this from Stauder, or if we need to go through the SPAWAR. If the latter, then I already have the forms to be filled out for the MOA to get the SDK as well as the C2PC 6.0 beta. I would think that only items 1 & 2 would fall under the NDA. Please discuss with Jerry.

Here is Jerry's contact information:

Phone: 636-498-6658
Cell: 314-308-0006
Email: jstauder@staudertech.com

Here is the list:
1. Design Guidelines Document (formal or informal)

2. Coding Standard (Organizational Style Guide)

3. Bug Tracking Process -- process (formal or informal) followed when addressing bugs

4. Tool Manifest -- what products (commercial or open source) are used for revision control, bug tracking, requirements tracking

5. What is the Build Environment?

6. For installation purposes, what is the OS and patch level for the OS?

7. For installation purposes, what is the compiler and patch level for the compiler?

8. Complete tool manifest of additional tools that will need on a machine prior to install, e.g. cygwin, etc.

9. Build instructions (formal or informal) --

9.a. Is there a specific place that the sources want to be installed?

9.b. Is the C2PC embedded in theTHS(X)?

9.c. Does C2PC need to be located in a specific place?

9.d. How do you build the stand alone THS(X) vs. the C2PC injector "version".

10. Install Instructions (formal or informal) 10.a. Once the binaries have been built, how must the target platform be installed and configured to successfully run the software? 10.b. Are two separate machines needed for the stand alone and the C2PC injector "version", or can these run on the same machine?

11. A set of sources delivered from Stauder with identifying information (version, tag, or label) 12. What version of C2PC is being used? How can CMU get access to the identical version?

Thanks,
Semper Fidelis,

Charles

Charles Muizers, SAIC
Project Manager
Littoral Combat Systems

Project Officer
Networked Fires
Littoral Combat - Future Naval Capability
Office of Naval Research

Work-812-854-3506
Cell- 812-360-3550
"Communications without intelligence is noise...Intelligence without communications is irrelevant." - Gen Al Grey, USMC

<<cmu_staudertech_PIA.doc>>
thsxBuildProcs.zip ¨thsxInstallProc.zip

Muizers, 21 Jan 2004

From: Muizers Charles M CONT CNIN
[mailto:Muizers_c@crane.navy.mil]
Sent: Wednesday, January 21, 2004 10:01 AM
To: John Gabler; Jerry Stauder; Gardner Robin D (Dale) CNIN
Subject: RE: Request for THS(X) Information in Support of the C2PC Trade Study

ALCON,

As you may have noticed the CMU NDA is taking much longer than desired... There are organizational delays at CMU with regards to the routing of such an agreement that are driving these delays. Iam now in jeopardy of those delays driving my schedule too far to the right, and want to intervene...

Here is my proposed solution. Below is the list of CMU's information requests. Of these I would like to know which elements would need to be covered by an NDA and which could be provided by Stauder without an NDA.

My thoughts (and hopes) are as follows:

It seems to me that items 5-12 would not need to be covered by an NDA. I would ask that Stauder provide the answers to these questions, and provide the requested sources from items 5-12 below, and that items 1-4 be pushed back until CMU can get the NDA resolved.

Is this acceptable and supportable to Stauder? If not, please let me know which of the below could be released/ answered prior to an NDA.

Thanks,

Charles
812-360-3550

1. Design Guidelines Document (formal or informal)
2. Coding Standard (Organizational Style Guide)
3. Bug Tracking Process -- process (formal or informal) followed when addressing bugs
4. Tool Manifest -- what products (commercial or open source) are used for revision control, bug tracking, requirements tracking
5. What is the Build Environment?
6. For installation purposes, what is the OS and patch level for the OS?
7. For installation purposes, what is the compiler and patch level for the compiler?
8. Complete tool manifest of additional tools that will need on a machine prior to install, e.g. cygwin, etc.
9. Build instructions (formal or informal) --

9.a. Is there a specific place that the sources want to be installed?

9.b. Is the C2PC embedded in the THS(X)?

9.c. Does C2PC need to be located in a specific place?

9.d. How do you build the stand alone THS(X) vs. the C2PC injector "version".

10. Install Instructions (formal or informal)

10.a. Once the binaries have been built, how must the target platform be installed and configured to successfully run the software?

10.b. Are two separate machines needed for the stand alone and the C2PC injector "version", or can these run on the same machine?

11. A set of sources delivered from Stauder with identifying information (version, tag, or label)
12. What version of C2PC is being used? How can CMU get access to the identical version?

Gabler, 04 Jan 2004

From: John Gabler
To: Muizers Charles M CONT CNIN
Sent: 04/01/25 21:07
Subject: RE: Request for THS(X) Information in Support of the C2PC Trade Study

Charles,

Below are the some answers to your questions:

5. What is the Build Environment?

Answer: We develop several core components that comprise THS(X). For most of those components, we use Microsoft VC++ 7.0 (aka .Net) However, the user interface and final build is compiled using Microsoft VC++ 6.0. This is because C2PC is compiled using 6.0. Those C2PC .dlls will not link with a 7.0 project. Since you are evaluating just THS(X) as it works with C2PC, I suggest that we send you just the source for the modules that interact with C2PC. That should significantly simplify your efforts in building a duplicate development environment. For, instance, you will only need VC++ 6.0. Warning, VC++ 6.0 is very hard to aquire now.

- For configuration management, we use MicroSoft Visual SourceSafe 6.0

6. For installation purposes, what is the OS and patch level for the OS?

Answer: We currently install on Remote Handheld Computers with Windows NT 4.00.1381. However, I believe that NT is stripped down by Tadiran (RHC Manufacturer) to satisfy DOD needs. That said, we have installed on Windows 2000 and XP with no impact on most functionality. The only issue is with GPS drivers on the RHC. If you want to install and test on a computer or laptop (No GPS or Digital Communications), I would feel comfortable on running on any current version of Microsoft OS (NT, 2000, XP).

7. For installation purposes, what is the compiler and patch level for the compiler?

Answer: As stated above, you should use Visual C++ 6.0

8. Complete tool manifest of additional tools that will need on a machine prior to install, e.g. cygwin, etc.

Answer: None

9. Build instructions (formal or informal) --

9.a. Is there a specific place that the sources want to be installed?

Answer: I don?ft think so. But, I will double check.

9.b. Is the C2PC embedded in the THS(X)?

Answer: Kindof. Actually, THS(X) is an injector into the C2PC application. C2PC exists as an application through which new functionality, like THS(X) can be added. To answer this a little bit more, we have coded our software so that it can it be easily converted from an injector to an application using other mapping components such as C/JMTK.

9.c. Does C2PC need to be located in a specific place?

Answer: No

9.d. How do you build the stand alone THS(X) vs. the C2PC injector “version”.

Answer: I will work on getting this for you in the next couple of days.

10. Install Instructions (formal or informal)

Answer: I will work on getting this for you in the next couple of days.

10.a. Once the binaries have been built, how must the target platform be installed and configured to successfully run the software?

Answer: Our software is installed using a disk created with InstallShild.

10.b. Are two separate machines needed for the stand alone and the C2PC injector "version", or can these run on the same machine?

Answer: I think that they can run on the same machine. But, I will double check.

11. A set of sources delivered from Stauder with identifying information (version, tag, or label)

Answer: I will work on getting this for you. As discussed above, I think that you want to set up a quick environment to evaluate, compile, build and test. To make this simple, I suggest that we get a snapshot of the User Interface related source code, zip it up and send it with compiled .dll and .libs of related components. Again, you may have some trouble getting a license for VC++ 6.0.

12. What version of C2PC is being used? How can CMU get access to the identical version?

Answer: The current plan is to deliver on C2PC 5.9.0.3. However, we are feeding requirements into version 6.0. For a copy, I suggest that you contact the C2PC Project Officer, William Bush, 703 432-4288, bushwj@mcsc.usmc.mil)

I?fm sorry for the delay in getting this material. I will try to follow up in the next day or so with the remaining info. As we discussed last week, I?fm at Nellis AFB until Thursday afternoon. However, I will try to satisfy you needs as best I can through email.

v/r

John Gabler
Director of Architecture and Quality
Stauder Technologies
115 Mexico Ct. Suite D
St. Peters, MO 63376
phone: 636.498.6658

Gardner, 30 Jan 2004

From: Gardner_R@crane.navy.mil
Date: January 30, 2004 4:36:59 PM EST
To: bommert@cs.cmu.edu, sthayer@cmu.edu
Subject: FW: Request for THS(X) Information in Support of the C2PC Trade Study

-----Original Message-----
From: John Gabler [mailto:jgabler@staudertech.com]
Sent: Friday, January 30, 2004 4:02 PM
To: Muizers Charles M CONT CNIN
Cc: Gardner Robin D (Dale) CNIN
Subject: RE: Request for THS(X) Information in Support of the C2PC Trade Study

Chales, Robin,

Attached are 2 rough versions of our build and install procedures for THS(X). These should satisfy questions 9 and 10.

Regarding question 9d, we haven't really built a deliverable C/JMTK version yet. It exists and is tested only on the developer's workstation. All components are in SourceSafe though. I'm told that there is little difference between the C2PC procedures and what is needed for C/JMTK. The major difference is that C/JMTK must be installed, and that a 1 or 2 different .dlls must be included in the InstallShield build.

I am still waiting on your view about our delivery of source code. Do you agree with my suggestion to just send the source for the source code interacting with the map, and everything else as a compiled .dll? Please call me to discuss.

John Gabler
Director of Architecture and Quality
Stauder Technologies
115 Mexico Ct. Suite D
St. Peters MO, 63376
phone: 636.498.6658
fax: 636.498.6659
jgabler@staudertech.com

Ommert, 23 Feb 2004

From: Bill Ommert
To: Muizers Charles M CONT CNIN
Cc: Gardner Robin D (Dale) CNIN; Scott Thayer
Sent: 2/23/04 10:16 AM
Subject: Stauder software status

Bill Ommert
Research Programmer
Robotics Institute, Field Robotics Center
Carnegie Mellon University, NSH 2205
412.268.9729 / wro@cmu.edu

Gents,

I got the Stauder Software Drop on Thursday. I have had a bit of time to look at it and partially digest what I got.

Here are the interesting points:

1) They shipped us something labeled as Spiral 2.This is new and different. All of the previous materials were labeled Spiral 1.

2) There is no design documentation with this release. The means that the latest design documentation is from Spiral 1. I will continue using spiral 1 documentation assuming that there is no spiral 2 documentation, or that Stauder will not provide it in a timely way. I would very much like to have Spiral 2 documentation (which presumably matches the provided source code); however, if it is not available immediately it is of very little use since the amount of time I have to review all of the materials is rather small. Every day that passes diminishes its value to me.

3) Dack has attempted an install of THS(x) onto a 5.9.0.3 patch 4 machine running Windows NT 4.0. C2PC has been run and is verified to some basic level to be working. When THS(x) is fired up it immediately dies. THIS IS PRELIMINARY -- I HAVE NOT YET SAT DOWN WITH DACK TO VERIFY ALL OF THE STEPS; however, I believe that he was quite careful and followed the provided instructions to the best of his ability. I will review his work soon and we will submit the process and findings. At that point, Stauder may need to be contacted to provide assistance if we are to get it to run.

4) A C/JMTK version of the software was provided -- though not in an installable form. We are presently investigating the viability of getting a C/JMTK software distro. It is not clear that we will have time to investigate this version of the software in any meaningful way for a number of reasons (advertised as a developers personal version, no installable image provided, questionable availability of C/JMTK). I am trying to get C/JMTK materials so that is can be an option if we find that we have some time to look at it.

On other topics, have we made any progress on a cd drive for RHC? If not, is there some other way that we can get the software onto the RHC?

Muizers, 23 Feb 2004

From: Muizers Charles M CONT CNIN <Muizers_c@crane.navy.mil>
Date: February 23, 2004 3:19:29 PM EST
To: 'Bill Ommert ' <bommert@cs.cmu.edu>
Cc: "Gardner Robin D (Dale) CNIN" <Gardner_R@crane.navy.mil>, 'Scott Thayer ' <sthayer@ri.cmu.edu>
Subject: RE: Stauder software status

Bill,

Spoke with John Gabler from Stauder.

If you would please make a list of the documents that you have, John will verify that you have the most current documentation. He admits that he does not have "Spiral 2" documents, but feels that there is minimal deltas between 1 & 2. Anyway want to verify that you have the most current docs.

Regarding the CD ROM, they have had problems with it in the past. What he recommends is loading through a network. He has offered to send you a home made cable that connects to the Fast I/O POrt of the RHC and connects to ethernet on the other side....Do you want one of these???

Let me know

Chuck

Muizers, 25 Feb 2004

From: Muizers, Charles M. [mailto:CHARLES.M.MUIZERS@saic.com]
Sent: Wednesday, February 25, 2004 8:09 AM
To: John Gabler
Subject: Documents and Cable

John,

I have confirmed that CMU has the following documents. Per our conversation, please review the list and if there are more current documents they would appreciated. Regarding your offer to provide a cable, that would be wonderful. If it would not be a problem, please forward that cable to Dale Gardner.
Thanks,
Charles

Appendix A - Communication Component API.pdf (create date of 3Nov2003)
Appendix B - Communication Component Class Details.pdf (create date of 3Nov2003) C2PC CCB Proposed Enhancements-Update.doc (create date of 3Nov2003)
O&O Concept for TLDHS Change 5.PDF (create date of 3Nov2003)
tldhs o&o ch4.pdf (create date of 3Nov2003)
SDD.pdf (internally datestamped 7Sept2003)
THS(X) SADD (Spiral 1) merged.pdf (internally datestamped 30April2003)
THS(X) SRD (Spiral 1)-merged.pdf (internally datestamped 30April2003)
THS(X) SRS (Spiral 1)-merged.pdf (internally datestamped 30April2003)

Charles Muizers, SAIC
Networked Fires Project Officer
Expeditionary Fires - Enabling Capability
Littoral Combat - Future Naval Capability
Office of Naval Research
Office- 703-676-2792
Cell- 812-360-3550

"Communications without intelligence is noise...Intelligence without communications is irrelevant." - Gen Al Grey, USMC

Gabler, 25 Feb 2004

From: John Gabler [mailto:jgabler@staudertech.com]
Sent: Wednesday, February 25, 2004 9:51 AM
To: Muizers, Charles M.; Gardner_R@crane.navy.mil
Subject: RE: Documents and Cable

Charles,

1. You have the most recent copies of the documents.
2. I am shipping the cable out today to:
   Dale Gardner
   NSWC Crane
   Code 6064, Bld 3218
   300 Highway 361
   Crane, IN 47522-5001
   Phone: 812.854.5981
3. I still haven't received the SOW from CMU. Would you please help me with that.

v/r,

John Gabler
Director of Architecture and Quality
Stauder Technologies
115 Mexico Ct. Suite D
St. Peters MO, 63376
phone: 636.498.6658
fax: 636.498.6659
jgabler@staudertech.com

Muizers, 25 Feb 2004

From: “Muizers, Charles M.” <CHARLES.M.MUIZERS@saic.com>
Date: February 25, 2004 9:58:42 AM EST
To: “Bill Ommert (E-mail)” <bommert@cs.cmu.edu>
Subject: FW: Documents and Cable

Bill,

Confirming that your design documents are the most current versions.

Also, cable will ship to Dale, and he will forward to you next week.

Thanks,

Chuck

Installation Instructions for the THS(X) Injector with Release 3.2.0

1. Make sure Internet Explorer 6 is installed. IE6 must be installed before the THS(X) Injector can be run. The program can be downloaded at:http://.microsoft.com/windows/ie/default.asp. Follow the IE6 installation instructions.

2. Make sure C2PC is installed. C2PC must be installed before the THS(X) Injector can be run. If you already have C2PC installed move on to the instructions for installing the THS(X) Injector. If you do not have C2PC installed follow the following steps to install C2PC:
    a. Run the C2PC setup. If you have the C2PC CD the setup should be run automatically when you insert the CD into your computer. If it doesn't, find the Setup.exe file and double click it.
    b. Click "Next" at the welcome screen.
    c. Accept the license agreement by clicking "Yes".
    d. At the "C2PC Installation Information" screen click "Next".
    e. Select "Client" for setup type and click "Next".
    f. Choose a destination folder and click "Next".
    g. At the "Select Program Folder" click "Next".
    h. At the "C2PC Gateway HOST IP" screen accept the default value of XXX.XXX.XXX.XXX by clicking "Next".
    i. Click "OK" at the "Information" dialog that appears.
    j. At the "Client Sub-Net Mask" accept the default mask by clicking "Next".
    k. At the "GCCS Opnote / Ovly Directories" screen accept the default directories by clicking "Next".
    l. At the "C2PC GATEWAY tcp/udp and IP multi-cast Port Numbers" screen accept the default values by clicking "Next".
    m. At the "C2PC VMF tcp/udp and IP multi-cast Port Numbers" screen accept the default values by clicking "Next".
    n. At the "C2PC Message Router tcp/udp Port Number" screen accept the default value by clicking "Next".
    o. At the GCCS TDBM HOST IP" screen accept XXX.XXX.XXX.XXX as the default value by clicking "Next".
    p. Click "OK" at the "Information" dialog that appears.
    q. At the "Start Copying Files" screen click "Next".
    r. Wait while the files are copied.
    s. When you are prompted to restart Windows select the "Yes, I want to restart my computer now>" radio button and then click "Finish". When your computer starts back up the installation will continue.
    t. Wait again while files are copied to your computer.
    u. When you are prompted to restart Windows select the "Yes, I want to restart my computer now>" radio button and then click "Finish". This will complete the installation of C2PC.

3. Following the following steps to install the THS(X) Injector.
    a. Run the Setup.exe file for the THS(X) Injector.
    b. Click “Next” at the welcome screen.
    c. Accept the default values at the “Customer Information” screen by clicking “Next”.
    d. Select a destination folder for the injector and click “Next”.
    e. At the “Ready to Install the Program” screen click “Install”.
    f. Wait while files are copied.
    g. At the “InstallShield Wizard Completed” screen click “Finish”.
    h. Wait again while the databases are set up.
    i. When prompted to restart your computer click “Yes”. This will complete the installation of the THS(X) Injector.

4. Running the THS(X) Injector
    a. Run the C2PC Client.
    b. Click on “Tools” from the C2PC menu bar.
    c. Select THSX. This will start the THS(X) Injector. If the injector is not listed continue on to the next step do the following”
        i. Click “Tools” from the C2PC menu bar.
        ii. Select “Injector Manager”
        iii. Make sure THSX is in the “Available Injectors” list to the left of the dialog that appears and check it if it is unchecked.
        iv. If THSX is not in the list, the install was not successful and must be redone.

THS(X) 3.2.0 on Shuttle, with C2PC Running, 08 Mar 2004

Baseline System - Windows NT 4.0 Build 1381 Sp 6.0a
-401comupd issue fixed from service pack
-mdac 2.6 downloaded from microsoft.com

Start Install
1. Inserted CD
2. Ran \Baseline_5903\C2PC\setup.exe
3. Clicked "Next" at the installation Welcome Screen
4. Clicked "Yes" to agree to license Agreement
5. Clicked "Next" on the Version Setup Screen
6. Chose "Custom" as install type - Clicked "Next" - to enable installation of gateway (THS instructions called for client - we installed custom to get both a client and a gateway) * Note - Install did not ask to reset *\
7. Installed to \Program Files\USMC\ - Clicked "Next"
8. Installed All options - clicked "Next"
9. Files Copy
10. Created C2PC Menu Group - Clicked " next"
11. Set C2PC Gateway to XXX.XXX.XXX.XXX(local) - Clicked "Next"
12. Set Subnet Mask to 255.255.255.0 - Clicked "next"
13. Entered Fully Qualified GCCS unix path to install opnotes and overlay displays (defaults) Opnote: /h/data/global/UB/message/Input Opnotes Overlays: /h/data/local/UB/Overlays Clicked -- "Next "
14. Entered Fully Qualified GCCS unix path for Input Screen Kilo and Four Whiskey (defaults) S.Kilo: /h/data/local/UB/Screen/Kilos 4.Whiskey: /h/data/local/UB/FourWhiskeys Clicked - "next"

15. Entered Gateway and IP multicast ports (defaults) tcp/udp: 2701 ipmc:2703 Clicked “next”
16. C2pc MultiTiered Gateway tcp/udp Ports (defaults) tcp/udp: 2702 Clicked “next”
17. C2pC VMF tcp/udp IP multicast ports (defaults) tcp/udp:1581 ipmc: 1582 Clicked “next”
18. C2PC MsgRouter tcp/udp port (default) tcp/udp:1581 Clicked “next”
19. IP for JMCIS/TDBM Host (default) kept xxx.xxx.xxx.xxx as we have no tdbm host Clicked “next”
20. Tdbm tcp/udp port (default) tcp/udp:2000 Clicked “next” /*Info- Connect to UB srver will not be possible if UB host IP not correctly set - clicked “ok”*/
21. Selected default connection option for c2pc gateway - chose Tdbm/UB3.0.2.x (default) -Clicked Next
22. List of options to be installed -Clicked Next
23. Created Username and password

/* REBOOT*/

C2PC 5.9.0.3.4PATCH
/*Gateway was not running*/
1.stopped c2pc timeserver from services in control panel
2.shutdown ALL C2PC Processes
3.”clicked next through all windows” – all with no options, just a next button
4.Selected yes to update support - install 4.x component for TMS server v4.5.2.0.P2 or later
5.Recreated USER ACCNT
6. after reboot -terminal windows(title TMSSDK) appears and runs
7. manual reboot

THS Install
1. Navigated to CD:\Setup.exe
2. Installed to \THS
3. Watch Files Copy
4. Terminal Windows Pops up and runs - gives errors
5. Install Completion Error 1722
6. Software Reboot
7. Opened C2pC
8. Opened Tools - No THS listed

Install THS(X) 3.2.0 on Shuttle with C2PC Stopped, 08 Mar 2004

Baseline System - Windows NT 4.0 Build 1381 Sp 6.0a
-401comupd issue fixed from service pack
-mdac 2.6 downloaded from microsoft.com

Start Install
1. Inserted CD
2. Ran \Baseline_5903\C2PC\setup.exe
3. Clicked "Next" at the installation Welcome Screen
4. Clicked "Yes" to agree to license Agreement
5. Clicked "Next" on the Version Setup Screen
6. Chose "Custom" as install type - Clicked "Next" - to enable installation of gateway (THS instructions called for client - we installed custom to get both a client and a gateway) * Note - Install did not ask to reset *\
7. Installed to \Program Files\USMC\ - Clicked "Next"
8. Installed All options - clicked "Next"
9. Files Copy
10. Created C2PC Menu Group - Clicked " next"
11. Set C2PC Gateway to XXX.XXX.XXX.XXX(local) - Clicked "Next"
12. Set Subnet Mask to 255.255.255.0 - Clicked "next"
13. Entered Fully Qualified GCCS unix path to install opnotes and overlay displays (defaults) Opnote: /h/data/global/UB/message/Input Opnotes Overlays: /h/data/local/UB/Overlays Clicked -- "Next "
14. Entered Fully Qualified GCCS unix path for Input Screen Kilo and Four Whiskey (defaults) S.Kilo: /h/data/local/UB/Screen/Kilos 4.Whiskey: /h/data/local/UB/FourWhiskeys Clicked - "next"

15. Entered Gateway and IP multicast ports (defaults) tcp/udp: 2701 ipmc:2703 Clicked “next”
16. C2pc MultiTiered Gateway tcp/udp Ports (defaults) tcp/udp: 2702 Clicked “next”
17. C2pC VMF tcp/udp IP multicast ports (defaults) tcp/udp:1581 ipmc: 1582 Clicked “next”
18. C2PC MsgRouter tcp/udp port (default) tcp/udp:1581 Clicked “next”
19. IP for JMCIS/TDBM Host (default) kept xxx.xxx.xxx.xxx as we have no tdbm host Clicked “next”
20. Tdbm tcp/udp port (default) tcp/udp:2000 Clicked “next” /*Info- Connect to UB server will not be possible if UB host IP not correctly set - clicked “ok”*/
21. Selected default connection option for c2pc gateway - chose Tdbm/UB3.0.2.x (default) -Clicked Next
22. List of options to be installed -Clicked Next
23. Created Username and password

/* REBOOT*/

PATCH
/*Gateway not running on login*/
1. stopped c2pc timeserver from services in control panel
2. shutdown C2PC Processes in taskbar*/
3. accepted defaults through all windows”
4 .Selected yes to update support - install 4.x component for TMS server v4.5.2.0.P2 or later
5. Recreated USER ACCNT
6. after reboot -terminal windows(title TMSSDK) appears and runs
7 manual reboot

INTERIM

C2PC test
1.started C2pc gateway
2. opened client, played around to prove functionality
3. exited client, halted gateway

THS Install
1. Clicked Setup.exe
2. Installed to \THS
3. Files Copy
4. Terminal Windows Pops up and runs - gives errors
5. Install Complete Message/Error 1722(a program run as part of setup did not finish as expected. Contact your support personel)
6. Software Reboot
7.Opened C2pC
8.Opened Tools - No THS listed

Install THS(X) 3.2.0 on RHC with C2PC Running, 08 Mar 2004

Install Start
1. Using Hub - Connected RHC through stauder cable to uplink port on hub - connected PC with straight cat5 ethernet cable to port #2
2. RHC IP - 10.1.16.111 -Set PC IP to 10.1.16.112
3. On PC - set CD drive to be shared on the network as \\10.1.16.112\D
5. On PC -Inserted MDAC 2.6 CD
4. On RHC -Opened Start Menu -Clicked on "RUN" -Entered "\\10.1.16.112\D"
5. On RHC -Ran MDAC 2.6 Update from "\\10.1.16.112\D" -MDAC 2.6 Updated -Rebooted RHC
6. On PC - Inserted C2PC CD
7. On RHC -Opened Start Menu -Clicked on "RUN" -Entered "\\10.1.16.112\D" -Went to "\\10.1.16.112\D\Baseline_5903\C2pc" -Ran "\\10.1.16.112\D\Baseline_5903\C2pc\setup.exe"

8. On RHC8. On RHC
C2pC Setup

-Clicked “Next” at the installation Welcome Screen
-Clicked “Yes” to agree to license Agreement
-Clicked “ext” on the Version Setup Screen
- Chose “Client” as install type - Clicked “Next” * Note - Install did not ask to reset *\
-Installed to D:\\Program Files\USMC\ - Clicked “Next”
-Instaled All options - clicked “Next”
-Files Copy
-Created C2PC Menu Group - Clicked “ next”
-Set C2PC Gateway to XXX.XXX.XXX.XXX(local)
- Clicked “Next”
-Info- Connect to Gateway srver will not be possible if Gateway host IPnot correctly set - clicked “ok”*
-Set Subnet Mask to 255.255.255.0 - Clicked “next”
- Entered Fully Qualified GCCS unix path to install opnotes and overlay displays (defaults)

Opnote:/h/data/global/UB/message/InputOpnotes
Overlays: /h/data/local/UB/Overlays

Clicked - “Next “ Entered Fully Qualified GCCS unix path for Input Screen Kilo and Four Whiskey (defaults)
S.Kilo: /h/data/local/UB/Screen/Kilos
4.Whiskey:h/data/local/UB/FourWhiskeys Clicked - “next”
- Entered Gateway and IP multicast ports (defaults)
tcp/udp: 2701
ipmc:2703

Clicked “next”
-C2pc MultiTiered Gateway tcp/udp Ports (defaults)
tcp/udp: 2702

Clicked “next”
- C2pC VMF tcp/udp IP multicast ports (defaults)
tcp/udp:1581
ipmc: 1582

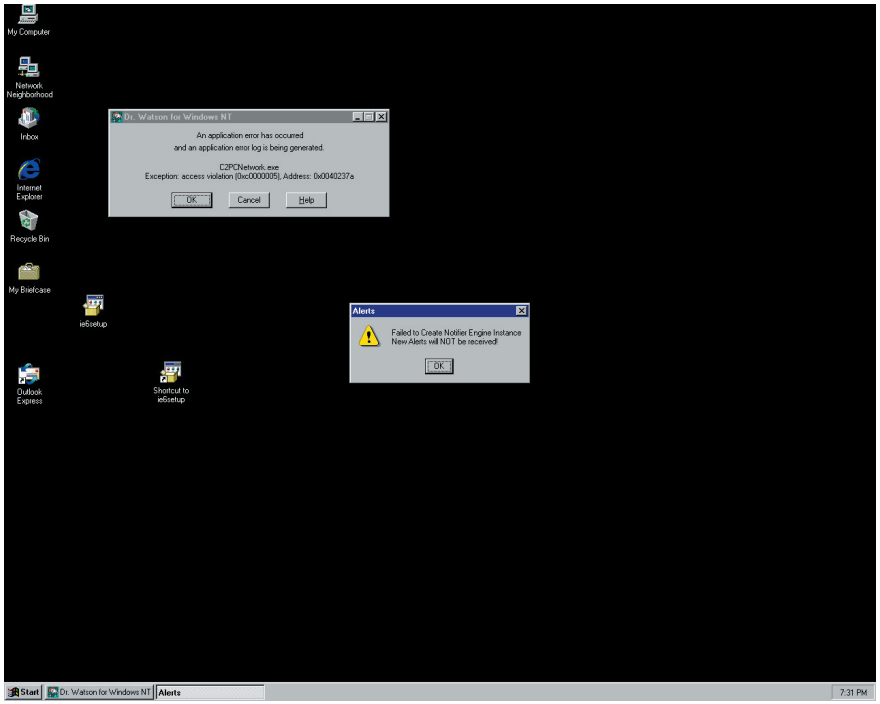
Clicked “next”
-C2PC MsgRouter tcp/udp port (default)
tcp/udp:1581-IP for JMCIS/TDBM Host (default)
kept xxx.xxx.xxx.xxx as we have no tdbm host

Step 9. On RHC On RHC -Opened Start Menu -Clicked on “RUN” -Entered “\\10.1.16.112\D” -Went to \10.1.16.112\D\C2pc_5903_patch4_31Jul03\C2pc” -Ran \\10.1.16.112\D\C2pc_5903_patch4_31Jul03\C2pc\setup.exe” -Clicked “next” at the welcome Screen -Clicked “yes” to License Agreement -C2PC installation info - clicked “next” -Start copying files - Clicked “next” -FILES COPY -Clicked “finish”
Step 10. On RHC -Manual Reboot
Step 11. OnRHC – **Installing THS**
/*C2PC Software Running - C2PC Time Server, Alert Server
Opened Start Menu
-Clicked on “RUN”
-Entered “\\10.1.16.112\D”
-Went to “\\10.1.16.112\D\”
-Ran “\\10.1.16.112\D\setup.exe”
-Clicked “next” at welcome screen
-Allow ths access for all RHC users- clicked “next”
-installed to E:\ths(x) (default) - Clicked “next”
-Clicked “Install”
-FILES COPY
-Popup “windows is setting up control panel so you can modify settings later”
-Installshield wizard popup - Click Finished
-MSDOS Script popup - Running SQL Scripts

/* MANUAL REBOOT*/

-Immediately upon Windows Login

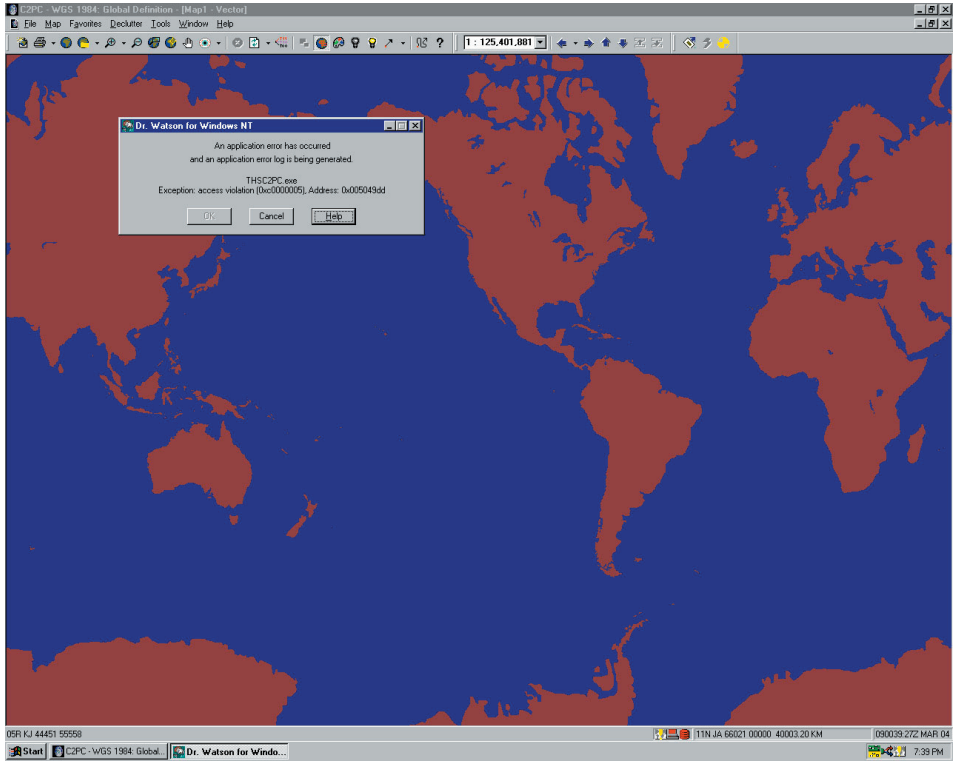
Windows Error - C2PCnetwork.exe Access Violation)



-Clicked “Ok”

-Started C2PC
-Opened “tools”
-Opened THS

Error - THSC2PC.exe Access Violation



-Clicked “ok”
-THS no longer appears under tools
Manual Reboot -Open C2PC -THS still does not appear

THX(X) 3.2.0 on Shuttle, Modified Install Process 09 Mar 2004

Note: Modifications due to conference call with Stauder and ONR on 9 Mar 2004.

1. Format HD 1 2048MB Partition(NTFS)
2.Installed to \WINNT Typical Install
3. Installed LAN and VGA drivers
/*REBOOT*/
4.Install NT SP3 High Encyrption
/*SOFTWARE REBOOT*/
5.Install NT SP4 High Encryption
/*Software REBOOT*/
6.Install NT SP5 High Encryption
/*Software Reboot*/
7.Install NT SP6 High Encryption
/*Software Reboot*/
8.Installed Partition Magic 7
9.Installed IE6
/*Reboot*/
10.Ran PM7 to redistribute free space
11. Ran \Baseline_5903\C2PC\setup.exe
12. Clicked “Next” at the installation Welcome Screen
13. Clicked “Yes” to agree to license Agreement
14. Clicked “Next” on the Version Setup Screen
15.Chose “Client” as install type - Clicked “Next”
/*REBOOT*/
16. Installed to \Program Files\USMC\ - Clicked “Next”
17. Installed All options - clicked “Next”
18. Files Copy
19. Created C2PC Menu Group - Clicked “ next”

20. Set C2PC Gateway to XXX.XXX.XXX.XXX - Clicked “Next”
21. Set Subnet Mask to 255.255.255.0 - Clicked “next”
22. Entered Fully Qualified GCCS unix path to install opnotes and overlay displays (defaults) Opnote: /h/data/global/UB/message/Input Opnotes Overlays: /h/data/local/UB/Overlays Clicked -- “Next “
23. Entered Fully Qualified GCCS unix path for Input Screen Kilo and Four Whiskey (defaults) S.Kilo: /h/data/local/UB/Screen/Kilos 4.Whiskey: /h/data/local/UB/FourWhiskeys Clicked - “next”
24. Entered Gateway and IP multicast ports (defaults) tcp/udp: 2701 ipmc:2703 Clicked “next”
25.C2pc MultiTiered Gateway tcp/udp Ports (defaults) tcp/udp: 2702 Clicked “next”
26.C2pC VMF tcp/udp IP multicast ports (defaults) tcp/udp:1581 ipmc: 1582 Clicked “next”
27.C2PC MsgRouter tcp/udp port (default) tcp/udp:1581 Clicked “next”
28.IP for JMCIS/TDBM Host (default) kept xxx.xxx.xxx.xxx as we have no tdbm host Clicked “next”
29.Tdbm tcp/udp port (default) tcp/udp:2000 Clicked “next” /*Info- Connect to UB srver will not be possible if UB host IP not correctly set - clicked “ok”*/
30. Selected default connection option for c2pc gateway - chose Tdbm/UB3.0.2.x (default) -Clicked Next
31. List of options to be installed -Clicked Next

32.Files COPY
33. /*not installing patch*/
/*reboot*/
Starting THS Install
34.Install to C:\THS(X)
35. Files Copy
36. Dos Command Windows Popups - Runs Sql - gives errors
37. Click “finish”
38. Error 1722. /* ThS reboots machine*/
39.install winzip to unzip stauder update files
40.Unzipped RegisterDigitalComs in THS directory
41.Opened C2pc
42.THS not in Tools menu
43.went to injector manager- THS resident - clicked to activate - deactivated all but trackplot and ths
44.clicked on “THS” in tool menu
45.THSc2pc - Error Connecting to the database - Error Loading Mapping Properties - Error Reading Op Area Names
46.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
47. Clicked “Switch to”
48. Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
49. Clicked “Switch to”
50. Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
51. Clicked “Retry:
52. Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
53. Clicked “X” to close the popup

54.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
55.Ended C2pC process
56.Warning Statement on US DoD Warning Statement
57.Clicked OK
58.Popup vanishes, no further activity
59.ReRan C2PC and Clicked on THS - SegFault Occurs
60.C2PC Still Running with THS menus - but does not respond
61.Opened Command Prompt - C:\Ths(x)
62.Ran RegisterDigitalComms.bat
63.runs several commandsin MSDOS - windows disappears too quickly
64.Opened C2PC
65.Clicked on tools - THS
66.THSc2pc -Error Connecting to the database - Error Loading Mapping Properties - Error Reading Op Area Names
67.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
68.Clicked “Switch to”
69. Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
70.Clicked “Switch to”
71.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
72.Clicked “Retry:
73.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
74.Clicked “X” to close the popup
75.Server Busy - Action cannot completed because the other program is busy. Choose “Switch to” to activates the busy program and correct the Problem
76. Errors Loop and won’t Exit

Ommert, 09 Mar 2004

From: Bill Ommert [mailto:bommert@cs.cmu.edu]
Sent: Tuesday, March 09, 2004 9:15 PM
To: John Gabler; Charles M. Muizers
Cc: David Kashmar; Scott Thayer
Subject: Updated status of THS install

John,
This afternoon Dave attempted the modified installation process that we discussed. Unfortunately, we did not have significantly more success with the modified process. I have attached Dave's blow by blow log of the steps he took. I'll let you review the attached log and referenced images.

To clarify, I believe that Dave has faithfully executed the plan that we discussed this afternoon. My understanding from our discussion of this afternoon is that we were to execute the following steps:

0) take the machine all the way down
1) install NT4, build 1381 and service pack 6
2) install IE6
3) install C2PC 5.9.0.3, client only
4) install THS 3.2.0
5) reboot
6) at the command prompt, go to the THS install directory, run the registerdigitalcomms.bat
Also, in the C2PC tools menu, open up the "injector manager" and turn off all of the injectors except for trackplot and THS.

The deltas in this process from what we were doing previously are the following:
1) We are no longer installing the C2PC 5.9.0.3 patch 4
2) We are no longer installing the C2PC server tools as described in the C2PC installation notes.
3) We are now running the registerdigitalcomms.bat file (enhanced version from your message of this afternoon) after we have completed the installation process as described in the installation instructions which came on the CD.

I have not yet opened up the scripts zip file that you sent today as well. To aid in problem isolation and identification, I want to get the basic system working with the minimum number of steps outside of the delivered documentation before trying to expand the functionality of the system.

The only remaining wild card that I can think of is the patch level of the operating system.
What service pack are you installing on the machines that you work with?

We are still running on the Shuttle PC at this point. I only have one RHC which we have not attempted an install on. Since there is registry key editing in the process I don't want to try to install on the RHC until we believe that we know what the issues are with our current process, or I have a way to reinstall the OS on the RHC. Clearly, if we view the Shuttle as the problem, we will attempt to work with the RHC. Since we have been getting equivalent behavior on each hardware suite I don't think that this is the problem, but am certainly willing to listen if you feel differently. If you are interested in looking at any shuttle pc documentation, the website for them is http://us.shuttle.com/

Any further thoughts on what the problem may be, or how we can get this installed and working, will be cheerfully accepted. Thanks for your support.

Cheers,
--wro

Bill Ommert
Research Programmer
Robotics Institute, Field Robotics Center
Carnegie Mellon University, NSH 2205

412.268.9729 / wro@cmu.edu

Gabler, 10 Mar 2004

From: "John Gabler" <jgabler@staudertech.com>
Date: March 10, 2004 10:18:23 AM EST
To: "Bill Ommert" <bommert@cs.cmu.edu>, "Charles M. Muizers" <CHARLES.M.MUIZERS@saic.com>
Cc: "David Kashmar" <dack@cs.cmu.edu>, "Scott Thayer" <sthayer@ri.cmu.edu>
Subject: RE: Updated status of THS install

Bill,

It looks like the database failed to build during the THS(X) install.

This should fix it:

- Remove THS(X)
- In the Program Files directory, remove the Microsoft SQL Server directory
- Install THS(X)

o Record any specific errors that occur while building the database

- Reboot
- at the command prompt, go to the THS install directory, run the registerdigitalcomms.bat

Note: The error "1722" always occurs during the installation process and is not an issue. We have been using Installshield express to create our disks and have found that it is a little underpowered. We will improve in later versions … I understand that this isn't much help to you. However, please understand that the 1722 is OK.

Let me know how this works.

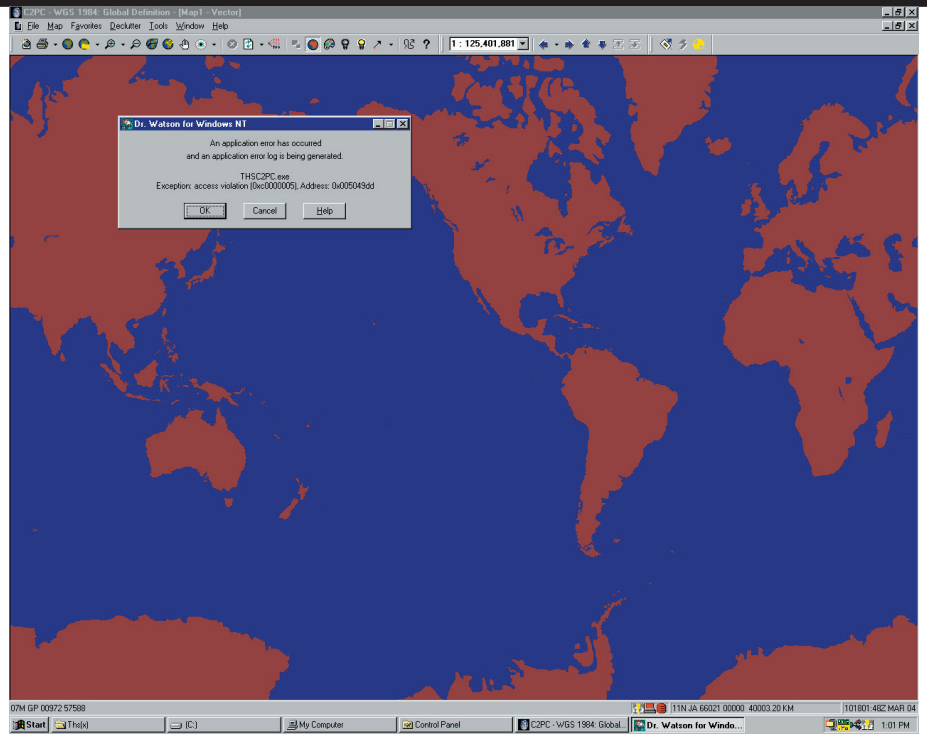
John

Reinstall of THS(X) 3.2.0 Database on 10 Mar 2004

This is the log of the steps outlined by John Gabler of Stauder to fix the failed THS install on 10Mar2004. The initial install is documented separately.

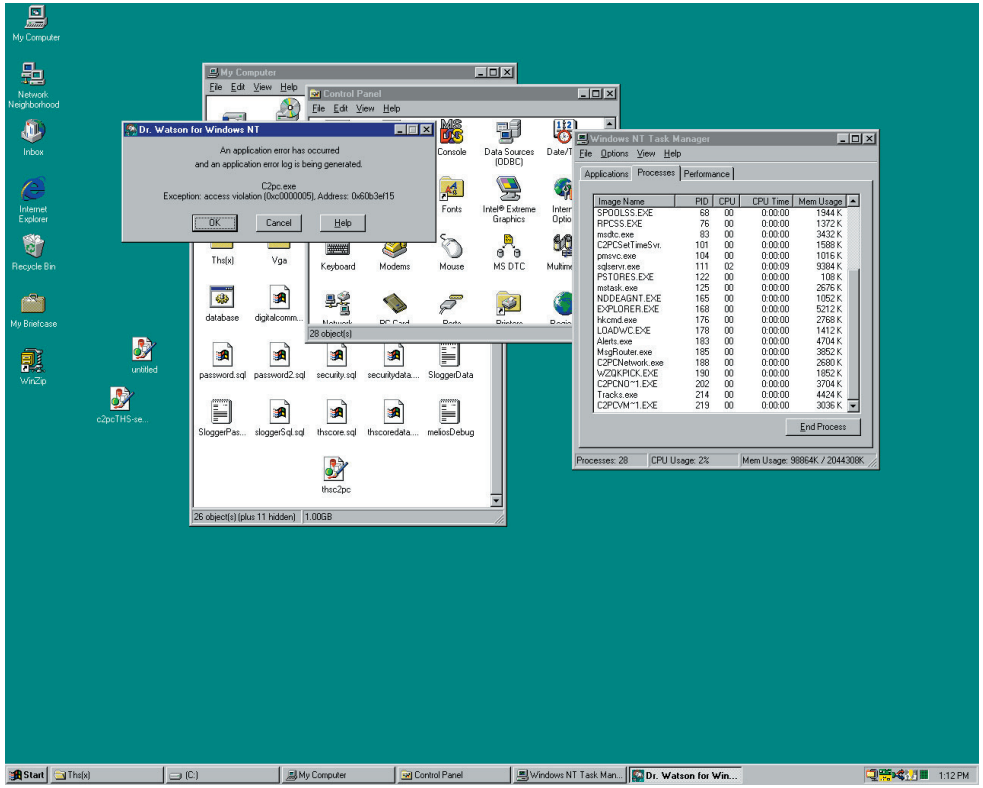
1.Shutdown All C2pC processes to make sure that nothing is using ths files
2.Clicked on “Add/Remove Programs”
3.Removed “THS Block III Spiral 1 Release”
4.Checked Task Manager for any running running instances of SQL: - found none
5. Looked for SQL directory - none resident
/*reboot*/
6.Reinstalled THS(x)
7.CryptoAPI Failed to initialize
8.Error1722
/*reboot*/
9.unzipped updated registerdigitalcomms
10.opened dos command line
11.ran registerdigitalcomms -runs several command lines -window disappears
/*reboot*/
12.open c2pc
13.ran ths
14.ERROR loading Database- Click OK
15.ERROR Loading Mapping Properties- Click OK
16.Error Reading Op Area Names
17.Error Reading Op Area Record
18.US DOD warning statement - click OK
19.Server Busy - Click Switch to

THSC2PC.exe Access Violation



20.C2pC still runs
21.tried running c2pc again
22.ERROR loading Database- Click OK
23.ERROR Loading Mapping Properties- Click OK
24.Error Reading Op Area Names
25.Error Reading Op Area Record
26.US DOD warning statement - click Exit
27.C2pc freezes
28.In task manager - End thsc2pc process
29.Error c2pc.exe – clicked ok

Error c2pc.ee



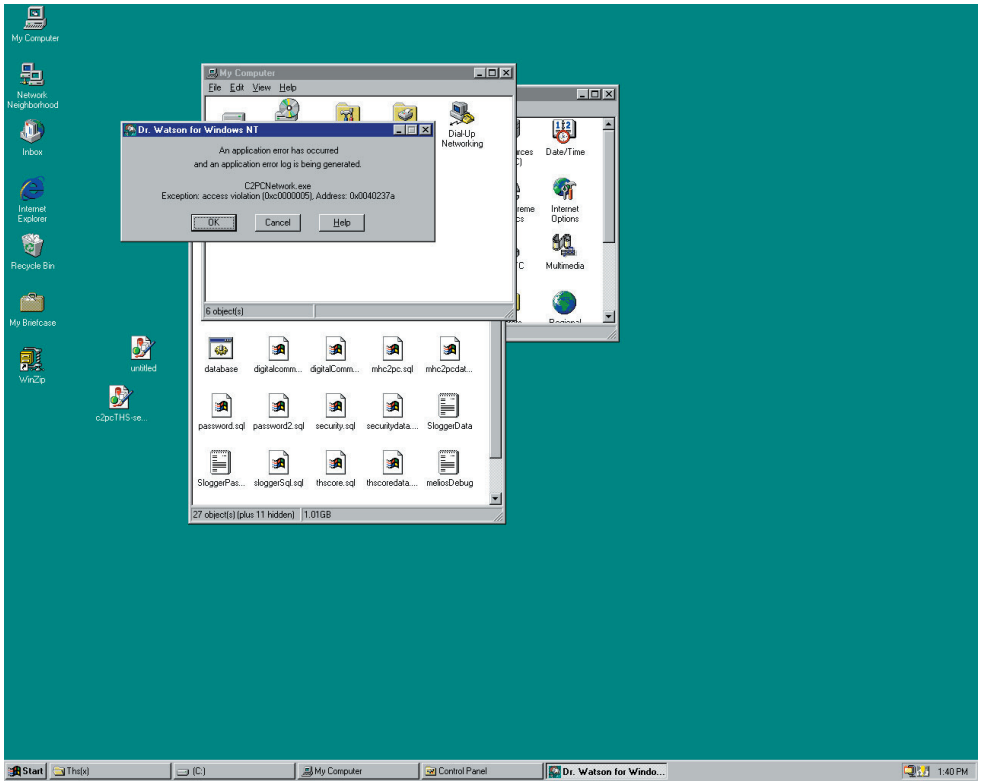
30./*rebooted*/

31.Upon Login into windows

-Error C2PCnetwork.exe (screenshot)

-Clicked Ok

Error C2PCNetwork.exe



32.Opened C2PC client

33.Clicked on Ths

-Error Connecting to Database - Ok

-Error Loading Mapping Properties - ok

-Error Reading Op Area Names - ok

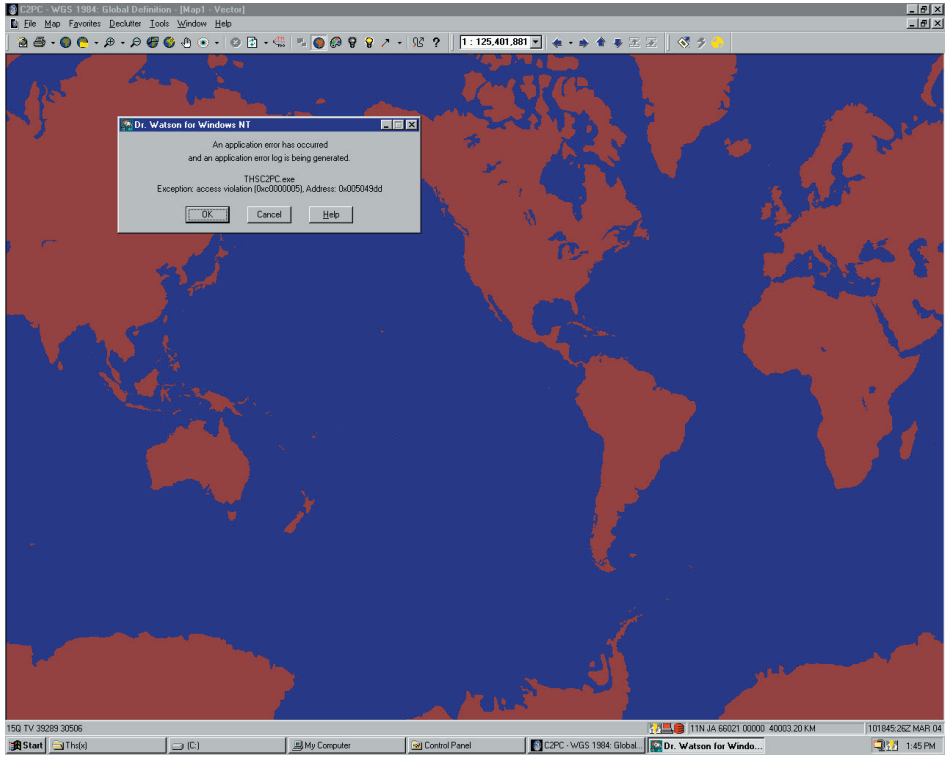
-Error Reading Op Area Record - Ok

-US DOD Warning Statement – OK

34.Server Busy- The action cannot be completed because the other program is busy. Choose “Switch To” to activate the busy program and correct the problem. “Switch To”

-THSC2PC Error

THSC2PC Error

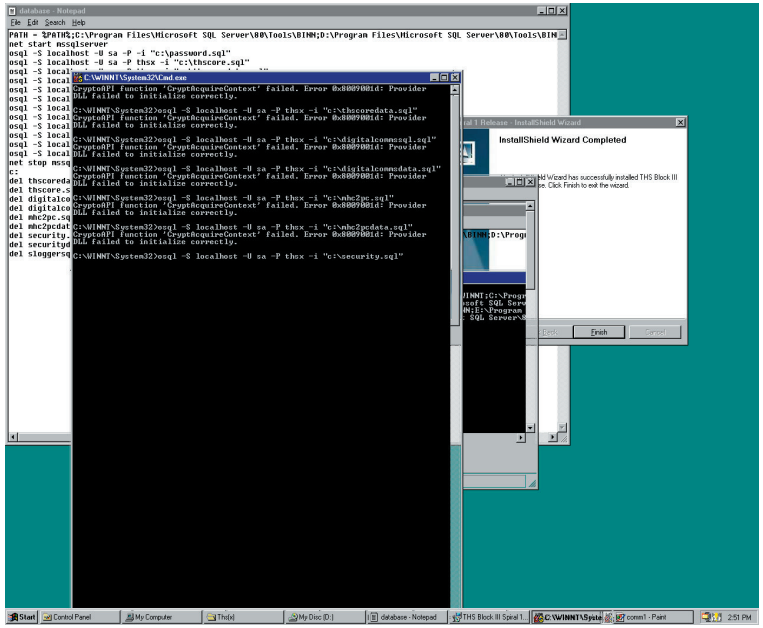


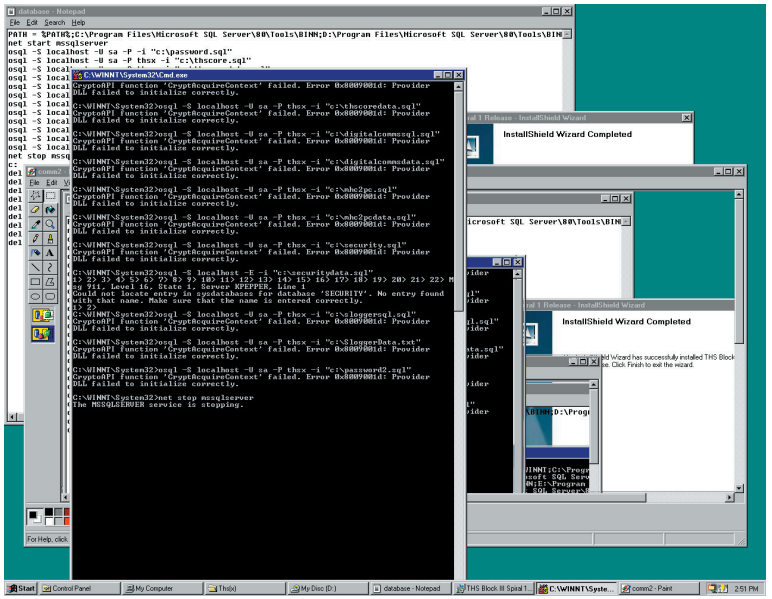
35.Uninstalled THS(to stauder instructions)

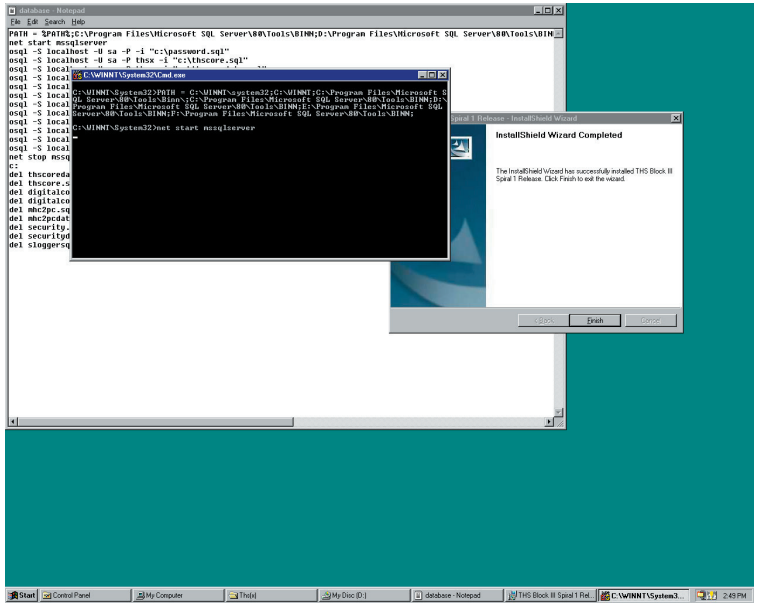
/*REBOOT*/

36.Installed THS

37.At end of installation, screenshots were taken of the sql command window that runs with errors (following screenshots)







38.uninstalled THS(to stauder instructions)
/*reboot*/
39.investigated database.bat - found hard coded directories (modified to THS directory)- reburned... reinstalled .... no effect
40. Uninstall THS from RHC -Uninstall Removes SQL Directory
/*reboot*/
41.Install THS
/*reboot*/
-Opened C2PC -Went to Injector Manager- No THS
42.Uninstall THS
/*reboot*/
43.Halt All C2PC Processes
44.Goto Add/Remove Programs - Remove C2PC
/*reboot*/
45. Install C2PC on RHC
46.Followed Stauder Directions
/*reboot*/
-Install THX -No apparent script running after install -Software wants to reboot
/*reboot*/
47.Startup C2PC -No THS Apparent in Tools Menu -No THS apparent in Injector manager

Ommert, 11 Mar 2004

From: Bill Ommert [mailto:bommert@cs.cmu.edu]
Sent: Thursday, March 11, 2004 7:02 AM
To: John Gabler
Cc: David Kashmar; Charles M. Muizers; Dale Gardner; Scott Thayer
Subject: THS install update (on RHC)

John,

Here is the promised update on the RHC install. I've attached Dave's log. This log is significantly more brief than the previous logs but it is a recapitulation of the processes documented and tried on the Shuttle PC. There are still problems -- THS is not showing up in C2PC at the end of the install. After the install completed, we reran the registry script and THS still did not appear. It is not appearing in the injector manager either.

Dave took the time to categorize the set of hot patches installed on the RHC in addition to ServicePack 6.
I have attached the set of hot patches that Dave identified.
What are the chances that the hot patches are making a the difference?
Do you have any experience with any of these hot patches?

Cheers,
--wro

Bill Ommertt
Robotics Institute, Field Robotics Center
Carnegie Mellon University, NSH 2205
412.268.9729 / wro@cmu.edu

Muizers, 11 Mar 2004

From: Muizers, Charles M. [mailto:CHARLES.M.MUIZERS@saic.com]
Sent: Thursday, March 11, 2004 7:24 AM
To: John Gabler; Bill Ommert; David Kashmar; Scott Thayer; Dale Gardner
Cc: Muizers, Charles M.; Stephens, Bradley S.
Subject: RE: THS install update (on RHC)

Gents,
If this is the solution, can we kill all of the birds and get
1 RHC- with C2PC and THS(X) injector
1 RHC- with THS(X) stand alone (C/JMTK)
1 RHC- with C2PC only??
John, let me know if the above is feasible.

Bill/Scott/Dale,
I will defer to you as far as this solution goes. I don't know of there are any technical issues with this approach.

Let me know.
SF,

Charles

Gabler, 11 Mar 2004

From: John Gabler [mailto:jgabler@staudertech.com]
Sent: Thursday, March 11, 2004 8:06 AM
To: Muizers, Charles M.; Bill Ommert; David Kashmar; Scott Thayer
Cc: Dale Gardner
Subject: RE: THS install update (on RHC)

Bill,

I think that this is getting to be too painful for everyone involved. To stop the bleeding, I propose that you send us the RHC, we load it and then return to you. With FedEx, I think we could complete the turnaround by Monday.

What do you think?

John

Gabler, 11 Mar 2004

From: “John Gabler” <jgabler@staudertech.com>
Date: March 11, 2004 8:42:54 AM EST
To: “Muizers, Charles M.” <CHARLES.M.MUIZERS@saic.com>, “Bill Ommert” <bommert@cs.cmu.edu>, “David Kashmar” <dack@cs.cmu.edu>, “Scott Thayer” <sthayer@ri.cmu.edu>, “Dale Gardner” <gardner_r@crane.navy.mil>
Cc: “Stephens, Bradley S.” <stephensbr@US-McLean.mail.saic.com>
Subject: RE: THS install update (on RHC)

Charles
I think that this will solve 2 out of 3 problems.
1. YES: C2PC + THS(X)
2. YES: C2PC alone - just to start the THS(X) injector
3. NO: C/JMTK - We are still working production and distribution issues including C/JMTK license keys.

As another/backup option, I can try to load our previous demonstration software (ACASS) which used MapObjects as the core mapping engine. MapObjects can be thought of as C/JMTK lite and is produced by the same mapping vendor ESRI. My only cautions are that it is not C/JMTK, and that it may not be compatible with C2PC. I am willing to try if that suits your needs though.

John

Ommert, 16 March 2004

From: Bill Ommert [mailto:bommert@cs.cmu.edu]
Sent: Tue 3/16/2004 9:28 PM
To: John Gabler
Cc: Charles M. Muizers; Dale Gardner; Scott Thayer
Subject: Thanks

John,

Thank you for you hospitality and support yesterday. I appreciate the time you invested in helping us to get THS(x) installed on the RHC.

I've taken a some time and written up my notes from yesterday's installation exercise. I've attached them to this message. If you have a chance and the inclination, please read them and let me know if I my notes and/or memory diverge from reality as you remember it.

Lastly, I tried to demo the system to my boss today when I got into the office. I managed to accidentally cause THS to "die" two times in a row. I wasn't trying to push at the edges of the system, though I suspect that I was. I wrote up those steps and have attached that file as well. If you have a chance to look at what I did and have any comments, please let me know.

Thanks again,
--wro

Bill Ommert
Research Programmer
Robotics Institute, Field Robotics Center
Carnegie Mellon University, NSH 2205
412.268.9729 / wro@cmu.edu

Gabler, 17 Mar 2004

From: "John Gabler" <jgabler@staudertech.com>
Date: March 17, 2004 8:14:56 AM EST
To: "Bill Ommert" <bommert@cs.cmu.edu>
Cc: "Charles M. Muizers" <CHARLES.M.MUIZERS@saic.com>, "Dale Gardner" <gardner_r@crane.navy.mil>, "Scott Thayer" <sthayer@ri.cmu.edu>
Subject: RE: Thanks

Bill,

On the errors you saw with THS(X):
1. On the C2PC Frozen when you hit the switch-to. - This is an issue that we are aware of and are working. It usually occurs when initializing a digital comms network. We have always gotten through this by clicking on the retry or switch-to buttons and then returning to C2PC. I don't recall seeing it hang under this scenario, but believe that you did see it. Please understand that we are working this.

2. For the problem with the CAS mission and duplicate names. Again you saw an issue that we are aware of and have fixed in our latest build. I'm sorry that this caused an issue for you. For now, with the build you have, please avoid duplicate names as you enter them.

Sorry that this is a quick message. But, I'm out of town and running to a meeting.

v/r

John

Successful Install

Initial attempts showed that CDROM was not in and of itself bootable, and appeared to be a simple copy and paste of the contents of a deployment disk. Thus, simply copying the files onto a Compact Flash (CF) disk was insufficient.

In seeking to turn the CF card into a bootable DOS disk, but we need access to a system running a Windows 9X Operating System to create a proper image.

A Win95 Startup Disk Image was downloaded off the web and booted onto desktop to a Win95 prompt. Unfortunately Windows 95 does not support USB or PCMCIA External Drives, making it impossible to read the disk conventionally. A solution was found using a device to make the computer see a CF card as a hard drive. Once the operating system could see the CF card, it became possible to format the card as a bootable drive.

At the command prompt, the commands that were entered were

format /S c: /*formats the CF disk to be a bootable partition*/
fdisk /MBR c: /*partitions the master boot record to be a primary boot drive*/

It was then possible to copy the files from the CDROM over to the CF disk. The CF disk was then tested and found to now be bootable.

1.Checked SanDisk for “OS-NT.txt” - present
2.Inserted PCMCIA FlashDisk into slot 2 of RHC
3.turned RHC on
4.Pressed F9 at startup
5.Chose Option 2 - boot from PCMCIA Slot
6.Win95 Loads to Dos Prompt
7.typed ghostpe
8.pressed enter to pass the startup dialog
9.went to \local\disk\From Image
10.navigated the the \Image directory
11.chose RHC301~.GHO file
12.enter license #FB87899670A8

13.Selected destination drive 2 (default)
14.pressed tab 3 times to navigate to the “ok” button - pressed ok button
15.selected “yes” to override the drive
16. Drive Images
17.Reboot
18.Press F9 @ bios
19. choose option 2
20.@ MSDOS prompt -cd bin -ran “updbios.exe”
21.ran “license” -chose 1. Update Unit Licenses -Unit # 10001 -updating action -chose 3 to exit
22.”pen” automatically runs
23.”lanaddr” auto runs
24.return to dos prompt
/***reboot***/
25.went to control panel/network
26.clicked on protocols/properties
27.changed Ip to 10.1.16.X
28.changed subnet to 255.255.255.0
29.closed network properties
/*reboot*/
30.Mapped CD drive from shuttle to RHC

31.on RHC E drive -Created “THS Install” Install Directory -copied over -C2PC install -IE60 -THS Spiral II directory
32.no TacLINK SOFTWARES - SKIPPING SECTION
33.Ran E:\THS INSTALL\IE60\IE6Setup -agree to license agreement -chose “install now” option - “next” -Installs Components -”Finish”
/*REBOOT*/
34.IE6.0 Installed

35.Ran E:\THS Install\C2PC Install\C2PC\Setup.exe
-Clicked “next” at welcome
-agreed to License agreement
-clicked “next” at C2PC install info
-Selected Client install - “next”
-install to E:\Program Files\USMC - “next”
-Created C2PC group- “next”
-left C2PC gateway IP to XXX.XXX.XXX.XXX- next
-Information
-The IP address does not appear to be valid Client will not have access to track Data If Gateway host is not correct.
-”ok”
-left subnet mask to 255.255.255.0
-Next
-left default

GCCS Opnote/Ovrly values
-Next
-Left Default Ports for C2PC gateway tcp/udp
-next
-left Default VMF Tcp/udp and IP multicast #
-next
-left C2PC Message router tcp/udp port #
-next
-left IP XXX.XXX.XXX.XXX for GCCS TDBM host IP
-Information
-The IP address does not appear to be valid Connections to UB Server will not be possible if UB Host IP is incorrectly set
-”ok”
-start copying files
-next
-Dialogue at ~10% “C2PC is installing MDAC”
- C2PC wants to REBOOT

/*REBOOT*/

36.C2PC Begins to copy files at login

37.c2PC wants to reboot

/*REBOOT*/
THS Setup
38.Ran E:\THS INSTALL\Spiral 3.2.1\Setup.exe
39.clicked “next” at welcome
40.took default values @custome info - next
41.Clicked “install
42.Files Copy
43.Clicked “finish”
44.Databases initialize
45.Error 1722
46.Software Reboot
/*REBOOT*/
47.Navigated to E:\THS(x)
48.Ran RegisterDigitalComms.bat
/*reboot*/
TESTING THE INJECTOR 49.opened c2pc client 50.No THS resident on Tools menu 51. THS in injector menu
52.went to command -went to E:\THS(x)
53.ran thsc2pc -regserver
54.opened C2pc -THS resident in injector manager

Appendix B: Coding Standard

Interfaces:

When the words “coding standard” enter into a conversation, it puts a chill in the air. No professional likes to have the total freedom of coding taken from them; however, it is critical in applications that are developed by multiple individuals and need to be maintained for long periods of time that some standards exist for the layout and commenting of code.

The goal of this document is to provide general guidelines for development. These rules are not intended to remove all creativity and flexibility from the development process. More than anything, this standard contains rules that should yield consistent style and reasonably tight code. The consistency of style removes some of the individuality from the code; this helps others to read, understand, maintain, and modify extant code. For instance, tight code results in the removal of many of the silly errors that often times occur when a case statement is just missed during development. Errors that are caused by missed cases are often the hardest to find because the code that is there looks right – because it is. It’s the missing code that’s the problem.

A theme that runs throughout this document is that code documentation is critical to the success of any development process. Code is an extremely abbreviated form of thought. Anyone who has written code professionally has had the experience of revisiting code that they wrote only to have no idea what it was doing, or why it worked. Documentation is the solution to the abbreviated thoughts encapsulated in code. The code gives you the what, the documentation should give you the why.

A coding standard is a living document. No two developers will agree on all aspects of software development style. In the best case, a coding standard will yield code that is consistent and easy to read. The adoption of a coding standard is a critical step of a mature software development process; however, it is only a single step. A coding standard that is not “enforced” through consistent code reviews will not be worth the paper on which it is printed. Code reviews will surface an amazing number of bugs, and will quickly bring all developers on to the same style page. These two things alone are quite positive and worth the “cost” of lost development time.

A few words on what this coding standard is not are in order. The adoption of a coding standard will not fix or even attempt to address design deficiencies in a system. This standard will make the code better; however, system design is an entirely different kind of review. There are some things in this standard that are somewhat language specific, or design concept specific (object oriented); however, in general, most of the rules enumerated below will make code in any language more robust and easier to read.

Rules:

1) Every "if" should have an "else". The thing about "if" statements is that they make a decision between two possible outcomes. Each outcome should be addressed – even if one possible outcome is only addressed with a comment.

Consider the following snippet of code:

```
#define THRESHOLD .0000001f
If ( floatValue > THRESHOLD )
floatValue = THRESHOLD;
```

As written, this snippet will clamp the floatValue to .0000001f. It is understood by the reader that if the value is greater than THRESHOLD, it will be clamped. What is missing, though, is that if the value is less than or equal to THRESHOLD, it is fine – this is the implicit **else**. This may seem like a trivial thing; however, it is a great kindness to the reader to make things like this explicit. A simple comment noting that the **else** to this **if** statement is to accept the current value would greatly increase the maintainability of this code. Future readers then can know why things are as they are and will not have to use intuition to figure them out. It is unnecessary to force indentation of the remaining code into an else block. The above code should be written as follows:

```
#define THRESHOLD .0000001f
If ( floatValue > THRESHOLD )
// floatValue is under the threshold value, clamp it to the THRESHOLD
floatValue = THRESHOLD;
// else, the floatValue is acceptable because…
```

2) Every command that can take a block should have a block. A block is defined as a pair of curly braces, {}. A block groups the things within it so that they are all considered to be at the same level. The places where blocks are commonly seen are in **if**, **while**, **for**, and **do** structures. Consider the following:

```
#define THRESHOLD .0000001f
If ( floatValue < THRESHOLD )
{
// floatValue is under the threshold value, clamp it to the THRESHOLD
floatValue = THRESHOLD;
}
// else, the floatValue is acceptable because…
```

The use of the braces in this context may seem unnecessary; however, subtle bugs can be introduced if braces are not used systematically. For example, the following 2 blocks of code do not do the same thing.

Block 1:

```
If( condition )
Printf( “foo”);
Printf( “bar”);
```

Block 2:

```
If( condition )
{
Printf( “foo”);
Printf( “bar”);
}
```

In Block 1, “bar” is always printed and foo is conditionally printed. In Block 2, both “foo” and “bar” are conditionally printed. This problem most often occurs in the debugging process when someone adds print statements to trace execution and the statements are not appropriately conditionally bound to the event that they are dependent upon. This problem can occur anywhere and can be hard to spot. As such, it is good defensive programming to always include blocks.

3) Comments
   a. No segment of code should have less than 30% comments. The 30% number is a good guideline, but more is often better. Code is an extremely abbreviated encapsulation of thought – especially if that thought contains any mathematical formulations.
   b. Every file needs to have an overview comment describing the function of the file.
   c. Every function or method should have a comment that is sufficient to describe the function without the reader having to resort to reading the implementation.

In general all code should have two types of comments. The first type is the end user documentation. This is the documentation that a consumer of a function, method, or class should read and then be able to use the item without misunderstanding the interface. This type of comment should include the usage model, any memory management issues, variable typing, unit information for each of the variables, return values, error codes, exceptions, and other items of interest.

The second type of documentation is internal to the function or method and is directed at the developer and maintainer. This documentation should address the concerns of an implementer. Specifically, these comments should include the following: algorithmic descriptions, mathematical derivations, and references to source materials where appropriate.

In general, the comments for the implementer will be significantly more detailed than comments for the consumer. The comments for the implementer should also be spread throughout the code – not lumped into one section as will be the case with the comments for the consumer.

4) White space is important. There is more to readability than comments. Good code is formatted in a way that breaks the processing up into logical, sensible chunks which are easy to read and understand.

```
For( int count=0;count<1000;count++){printf("The square of %d is %d\n");printf("The cube of %d is %d");}
```

While is seems unlikely that someone would format code as it appears above, it has been done. The above code is much simpler to parse and understand when reformatted as below.

```
for( int count=0; count<1000; count++)
{
printf("The square of %d is %d\n");
printf("The cube of %d is %d");
}
```

This is a very small and simple example. Typically these problems are easier to identify on such a small segment of code. The more devastating problems occur when many small pieces of code have been run together. One example of this is to consider the effects on the readability of a file when all of the blank lines between methods are removed. This standard recommend two blank lines between each method or function in an implementation file, and a single blank line between the major computational steps in a method.

5) Every file needs to have a copyright statement. This is more of a legal concern than a development concern; however, it is a good practice to mark all sources with a standard copyright so that they can be updated en masse at the beginning of a new year. Many places do not keep the copyright headers up to date because of the amount of time and hassle involved in updating all of the files by hand. Updating these copyrights by hand is unnecessary, because they are all completely standard. The standard recommends the use of the following stock copyright comment:

```
/*
 * BEGIN_COPYRIGHT
 *
 * Copyright 2004, Carnegie Mellon University.
 * All Rights Reserved.
 *
 * END_COPYRIGHT
 */
```

Clearly, "Carnegie Mellon University" should be replaced with the appropriate organization name. Other than that, this statement can occur anywhere in the file and can be found and updated by a simple regular expression parser. The standard recommends placing it as the first thing in the file.

6) Every file needs to have a versioning header. Version headers can save much time by clearly identifying the version of a file being used and the person who created it. In the best of all worlds, the versioning header also contains links to a bug database which indicates what defects were addressed by modifications to the file.

   In general, all revision control systems recognize and use the RCS header format and keywords. As such, the following version header is recommended:

```
/*
 * $Header$
 *
 * $Author$
 * $Date$
 * $Revision$
 */
```

This header includes the information about the file's location in the repository, the creator of the last revision, the create date for that revision, and the revision number. All of this information is helpful and will give a person insight into what has happened to this file and where to go look for more information.

The versioning header should immediately follow the Copyright text in each file.

7) Every variable should be initialized. Don't trust the compiler. Many subtle and "random" bugs have been caused by not initializing memory. There are some calls that return initialized memory to the caller, and there are many that do not. Variables allocated at runtime on the stack are not typically initialized by the compiler. Regardless of the source of the memory, always initialize memory before use. For blocks of memory, use **memset**. For single instances, set them equal to NULL, or some other invalid value so that random data does not enter into the system.

8) Do not optimize a piece of code unless the profiler identifies it as the bottleneck. Everyone has that one piece of code that they have optimized out to be super fast and efficient. Almost uniformly, the code that has been tweaked out has been of no consequence in the overall application performance. Ahmdahl's Law clearly shows that the net effect of an optimization is limited by the percentage of the execution time that the optimization affects. Specifically, making a single piece of code which accounts for 1% of a process 1000 times faster, makes the program run 1000 times faster for 1% of it's runtime. It is much more meaningful to make something which accounts for 60% of the runtime go 1.25 times faster.

There are good tools that profile applications. Modern optimizers do a good job. It is rare that a programmer will outguess these tools. There are times when the human will outperform the tool; however, this is usually accomplished though the insertion of more appropriate data structures and algorithms – not though unrolling loops or other such code tweaks.

9) Every deviation from the coding standard should be explained in comments. The coding standard is the gold standard for an organization's code. That said, it can not anticipate all circumstances. It is permissible to deviate from the standard; however, you should document what the deviation is, and why it was done. This is common politeness to those who come after you and are use the coding standard as a guide.

10) Variable naming
    a. First word lower, inner words upper. thisIsAGoodVariableName
    b. Units should be encoded in the variable name
    c. Naming should be unambiguous and clear Good variable naming can make a system far stronger.
       Bad variable naming can wreak havoc in a system for all time.
       There are many layers which should be considered: conciseness, clarity, and readability.

The basic format for a variable should follow the first word lower and inner words upper format. This format takes some getting used to. It looks odd for a short time, but then is quite readable and compact. For instance, consider the following variable names: metersPerSecond, framesPerSecond, angleToGround, accelerationOfGravity, distanceToGoal.

There has been talk of self-documenting code – this may not be as effective as has been reported. Code is too brief and distilled to be truly self-documenting; however, well chosen variable names can greatly aid in understanding.

The second thing about variables is that they should encode units in the name. In the examples given above, we can see names which encode the necessary units and names which do not. In the variable angleToGround, it is not clear if the angle is in radians, degrees, grads, or some other angular measure. On the other hand, given a value in the variable metersPerSecond, it is immediately clear what units of measure are being used.

Thirdly, the naming should be clear. The meaning of a variable named metersToGoal is not immediately clear. Is that a Cartesian distance to goal, Manhattan distance to goal, or some other algorithm specific value? In general, it is appropriate to simply use meters to goal internally in a method if there is supporting documentation; however, it deserves a slightly more descriptive name if used as part of an interface.

Lastly, variable name length is of secondary concern. Many times variable names have been abbreviated, truncated, or just badly constructed so that the developer can save a few keystrokes. This is a poor practice. It is well worth the extra time and effort during initial development in order to increase the readability and maintainability of the code.

11) Type and Enumerated Type naming Types should be first letter of each word Upper. ThisIsAGoodTypeName.

12) Symbolic constants are useful and clear; numeric constants are an invitation to bugs. Constants are often used for thresholds, constant multiplers, and things of that nature. While it is true that the speed of light in a vacuum is not going to change this week, the maximum number of robots in the system may. Additionally, symbolic constants can greatly increase readability. 300000000 is a fine representation of the speed of light; however, SPEED_OF_LIGHT_METERS_PER_SECOND is completely unambiguous and clear.

    All of the naming issues of variables also affect constants. Generally, instead of a first word lower and inner word upper scheme, constants are all capitals with underscores to separate words. This makes them stand out as the special things that they are.

13) Scoping

    a. There should not be global variables. When a Global variable is used, by definition whatever is done to it is a side effect of the routine. Side effects should be avoided; they are inherently untestable. If something is to be modified, it should be an argument to, or a return value from, a function or method.

    b. Variables and constants should be scoped as tightly as possible to reduce the possibility of name space conflicts and confusion. It is very sensible to have a constant named MAX_HEIGHT in both a building class and also in an airplane class; however, they will not be the same value. If the constants are scoped as private enumerated values in the appropriate class then these two classes can each have a version of this "constant."

    c. Variables in classes should not be scoped as public. Variables should be scoped as private unless direct access to the variable is needed by a subclass. The question that must be asked of a protected variable is whether or not it would be better to have a private variable and a set of protected accessor methods.

14) Each class should be in a file named after the class. This seems an unnecessary thing to say to any professional software developer; however, it is a practice which is not always observed. Specifically, if there is a class in the system named Localizer, there should be a Localizer.h to contain the relevant prototypes and a Localizer.c or a Localizer.cpp file in the system to contain the relevant implementations and instances of static member variables.

15) There is no reason to have more than a .c and a .h file. To continue to use the Localizer example from above, there could be a Localizer.inl file that contained the inline functions for the Localizer class. This is adds unnecessary complexity which can lead to confusion and cause unfortunate problems.

    If, for some reason, you really do feel that this is necessary, then this is something that should be called out in comments in the header and implementation files.

16) The use of **goto** is not recommended. **goto** leads to reading, testing, and maintaining nightmares.

17) Enumerated types give meaning to explicit sets of values. The constants could encode many different things: states in a state machine, packet types, the complete set of keys for key value pairs, and countless others. Enumerated types are an extension of the power of symbolic constants. Symbolic constants are a nice way to handle singleton constant values; enumerated types are a nice way to group related values together into a self-checking group (compiler checked, actually). State machines are very powerful, but often require modifications to their structure during development. It is nice to be able to group like states together, and to be able to refer to them by name. Consider the following:

```
enum{ InitialState, WorkState, FinalState }
```

These 3 states compose a very simple state machine with only 3 states. The transitions for this machine are also simple: from InitialState to WorkState, and from WorkState to FinalState. One could probably get this to work with numbers: 1 goes to 2, and 2 goes to 3. If we consider a more complex state machine, things become rather more complicated:

```
enum{ InitialState, WaitForConnectionState, ServiceConnectionState, CloseConnectionState, FinalState }
```

The 5 states enumerated above can be used to compose many different state machines. The most common use for states such as these is to construct a simple server to service a single client which may connect and disconnect multiple times. A common state machine would be for the InitialState to go to the WaitForConnectionState, the WaitForConnectionState to transition to ServiceConnectionState when a client connects and remain there until the client disconnects, ServiceConnectionState transitions to CloseConnectionState when the client disconnects and then transitions back to the WaitForConnnectionState. The FinalState is reserved for handling errors, or for the application to forcibly shut down the system. The same sequence could be written with numbers instead of words for the states; however, it would be totally incomprehensible and therefore not maintainable or extensible.
The last trick to point out for enumerated types is the use of typedef and class level enumerations to make enumerated types more useful to an application at large. Not all implementations of a state machine will be within a single object or even a single file (indeed, see the State Pattern in Gamma). As such, some scoping and persistence in the enumerated type is necessary. The simplest way to get the type available in multiple places is to make it universally available through the use of typedef.

```
typedef enum{ InitialState, WaitForConnectionState, ServiceConnectionState, CloseConnectionState,
    FinalState } ServerState;
```

This typedef now allows variables of type ServerState to be created. This solves the first problem; however, if there are multiple valid servers running in the same process, scoping may become a problem. To address this, enumerations can be

made a public, private, or protected members of a class. This eliminates the typedef statement and the addition of the type to the global namespace. The use of class level enumerations is encouraged when it is appropriate.

18) Remember, all text strings will eventually be seen by the user. This sounds silly, but numerous times users have seen some developers debugging text (some rather unsavory text). Depending on the user in question, this can be something of a problem – their sense of humor does not usually line up with that of developers. In one classic case, the network police came hunting down a machine which was running a server that had gone into an error mode. In this mode, the server was spewing "Unix rules; DOS sucks" packets onto the network as close to line speed as it could manage. Needless to say, people did not find this particularly funny; the content of the text only made the situation worse.

19) Do not trust **malloc** or **free**. This concept can be extended to cover all system calls, or all calls that are ever made; however, the most egregious offenders are calls to **malloc**, **free**, **new**, and **delete**. **Malloc** can validly return NULL if it is unable to allocate the memory you request, or if the argument to it is invalid. Referencing a NULL is the most frequent cause of unintentional programmatic death. Equally disturbing is the effect of calling **free** or **delete** on NULL. This seems as though it should be harmless; however, the behavior of the program will vary based on the operating system on which the code is running. In WindowsCE 3.0, if **free** is called on NULL, the execution of the present stack frame is stopped, and control is returned to the calling stack frame immediately. This can be rather challenging to detect and correct. Always check the values returned from memory allocators, and the values given to memory deallocators. It is a very good idea to check the return codes of other methods as well.

Yes, always checking return codes will inflate the line count of code and make it seem to flow poorly; however, this is one of the major differences between prototype or demo code, and production code. One of the hardest things to do in software is to handle these errors well on a consistent basis.

20) Use meaningful text in error strings. Many of us have written code that looks like the following:

```
if( returnCode == FAILURE )
{
    fprintf( stderr, "Fatal Error! Error code: %d\n", returnCode);
    exit( returnCode );
}
```

Problem #1: this doesn't tell anyone where the problem actually occurred or what was being attempted at the time that the error occurred. This means that the application was running fine and then just died. One could argue that the user knows what they were doing and so debugging can begin from that point. The counter to that argument is that the application may be multi-threaded and what the user was doing may not have been the proximate cause of the problem.

Problem #2: this writes an error code to standard error. This is much better than writing it to standard out; however, it assumes that the user will be monitoring standard error, but this information is not helpful to the user. It may be helpful to a developer, but likely it is not.

Problem #3: this code exits on error. There are few things worse than just having a program die. As a user it is very disconcerting; if the program is a server which is supposed to be somewhat robust (running on an autonomous robot perhaps) just exiting has possibly killed the robot. Where possible, the software should be built to run in degraded modes when a failure is detected.

21) Judicious use of "assert" and "exit" can be helpful; however, inappropriate use can cause severe problems. "Assert" is a fantastic macro for developers when building code. It allows for conditions to be asserted and automatically tested with a single line of code. Consider the following 2 examples:

Sample 1:

```
if( condition == TRUE )
{
        fprintf( stderr, "Assert failed at %d in %s\n", __LINE__, __FILE__ );
        exit( 1 );
}
```

Sample 2:

```
assert( condition == TRUE );
```

Believe it or not, these 2 examples do the same thing. The user of "assert" is much more compact. There are problems, though. The "assert" is a preprocessor macro which is removed when DEBUG is not defined, or when NDEBUG is defined (this varies possibly between compilers, so be careful!). This means that you can not depend on "assert" to be there in production builds! Further, when it is available, a failure results in exiting the software, so even if a production build is done with DEBUG defined (not a good idea!) the behavior is still quite lacking.

The primary use of "assert" should be in the debugging of applications. It is not a substitute for good bounds checking.

22) Consistent tabbing is important. This is an extension of the "white space is important" rule. If used properly, line breaks in a file, class, or method can lend great clarity to what is going on in a file. Most of the time developers are looking at a few lines of code specifically. In such cases, consistent tabbing allows the developer to quickly and accurately discern the flow of the code that is being reviewed. The default Unix tab is 8 spaces. This is a huge indention and code quickly flows over the 80 column barrier. A far more sensible tab setting is 2 spaces per tab. 4 spaces per tab is also acceptable., In any case, a consistent number should be selected.

Often there is a source base which uses a specific tab setting and there is resistance to changing that setting, or even standardizing that setting. There are many utilities which help with this process: tabify, untabify, perl, sed, and awk are but a few. Do not let an extant source base prevent the right thing from happening.

23) Code and comments should comfortably fit inside of 72 columns. Often code is printed. Whether for the purposes of debugging, peer review, or full out code review, the printed page is quite helpful. Printers print 80 columns. that is the standard. As such, all code should fit neatly into 72 columns. If the goal is a 72 column break, a character or two extra on a line is not a problem. If the goal is 80 characters per line, there is no flexibility. Do not be afraid to break single statements up into multiple lines. Consistent indention will make things quite a bit easier to read.

24) Do not overload return values. Be careful to not have error return values that are also valid return values. This is another simple rule that will save great amounts of time and trouble in debugging. In general, error return values should be defined constants or enumerations. It is often simplest to pass in all of the return values as pointers to valid memory in the heap. This strategy reduces the chances of memory problems later. This is not always appropriate. Judgment is required.

25) Clearly document when a function or method returns newly allocated memory. Memory leaks can be tricky to detect. Some of the more insidious leaks come from functions or methods returning pointers to allocated memory and not informing the consumer of the routine that the memory needs to be deallocated when the user is done with it.

26) Do not trust input arguments. People lie, cheat, and steal. Programmers are people, too. They may not be bad people, but they make assumptions that sometimes are incorrect. The only way to write code that always works is to trust nothing – even if you are calling yourself. Bounds checking input values can add a large volume of code; however, it almost increases the quality and clarity of the code. Comments to the developer are very useful since it makes the valid ranges of values for each argument clear in the mind of the programmer – often not the case if the bounds checking is not done.

27) Code for testability. The unit test is an invaluable aid. If code is designed correctly, unit testing is simpler, and bugs can be found at the lowest level. Finding bugs at the unit level reduces the difficulty of integration and system testing. This does not mean that every class or function needs to have thousands of lines of test code. Tests are generally written in a lazy execution way – regression. If code is built for testability, many bugs will be identified before they have a chance to occur.

28) Libraries should never call “exit” (or anything comparable). When a user calls a function in a library they expect a valid return value, or an error. What they do not expect is for the program (or thread in the program) to exit. This is one of the most inconsiderate and aggravating things that happen. Systems must be held to a high standard; library code must be held to an even higher standard. Libraries are services. The consumer should not have to debug them to get work done. Similarly, the code commenting in libraries should be held to a higher standard.

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

Appendix C: Severity Key

	location			Affected Pillars				
de-fect#	**dir**	**file**	**description**	**port-ability**	**reli-ability**	**maintain-ability**	**exten-sibility**	**test-abil-ity**
	THSCore	GenericLookupList.{h,cpp}						
1.1			Class level comments expose implementation details to the class consumer					
1.2			Class contains Protected member variable, breaks encapsulation					
1.3			Class uses Carray to store internal data. Carray is a Microsoft Foundation Class.					
1.4			Cut and paste coding where factoring could reduce line count and reduce the potential for bugs. Specifically, the methods GetDescriptionFromEnum(long lEnumValue) and GetIdValue(long lEnumValue) use the same iteration code through the internal data structure (the CArray) and differ by 2 lines total. This problem is likely systemic throughout the set of lookup functions.					
1.5			There is an implicit and, as far as I can tell, unenforced assumption in the GenericLookupList relating to enumerations. The assumption is that only a single entry per enumeration will be made into the GenericLookupList. If there is more than a single value per enumeration, only the first will ever be returned. This failure to identify further matches is silent.					

Portability Reliability Maintain-ability Extens-ability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

1.6			The interface to the method GetListOfDescriptions(CStringArray* strArray) strongly encodes a MFC class. CStringArray is a MFC data structure and is part of the public signature of this method.						
1.7			Pervasive use of magic numbers. Variables throughout are set to 0, -1, and "" (the empty string). These values are not defined and managed centrally.						
	THSCore	LookupItem.{h.cpp}							
2.1			Use of Cstring, MFC specific string class, to store string data						
2.2			Methods do not enforce good usage: each item to be "looked up" can be looked up by name, id, or enumerated value; however, each of these things can be set independently of the others allowing for only one to be set.						
2.3			Unclear that the values which are returned are meaningful. The initial values for each field are 0. It is not clear that 0 is an invalid value -- indeed it likely is a valid value.						
2.4			Bounds checking not done on arguments in copy constructor. This may cause unlogged, and untrapped runtime failures. The copy constructor immediately dereferences the argument passed in. If it is NULL the thread executing the program will likely terminate.						
	THSCore	RaidoList.{h,cpp}							
3.1			Radio misspelled in filename, contained class spelled correctly						
3.2			Header file contains no documentation on class purpose, or for methods implemented therein.						

Portability Reliability Maintainability Extensability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintain-ability	Extens-ability	Testability
3.3			SQL query string hardcoded in .cpp file. Assumes a database schema, is undocumented.					
3.4			Creation of unnecessary temporary variables. strFormat and sqlStr are distinct, but contain the same information. Only one is ever really used.					
3.5			In LoadList, there is an if() without an else. It is not clear whether the if failing is a bad thing or not.					
3.6			There is no bounds checking on the data loaded from the database. This may allow for corrupted data to be loaded into the program and to cause a runtime fault.					
3.7			There is no discussion of what the valid sets of values are for the enumerations and Ids for ra-dios. This makes error trapping inside the class impossible.					
3.8			Exception handler only catches a single class of exceptions (_com_error) If that type of error occurs, then the fault will be logged. It is not clear what will happen if this method receives a different type of exception. This method does not advertise to the consumer that it may throw an exception.					
3.9			In the case of a caught exception, the object is left in an unknown state. The data which has been loaded will be availible, but there is not necessarily any indication that other data was lost during the loading process.					
	THSCore	ResponseType List.{h,cpp}						
4.1			Header file contains no documentation on class purpose, or for methods implemented therein.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
4.2			Class constructor does the work of LoadList. LoadList is specified to populate the internal list structure contained as a protected data member in the superclass, CGenericLookupList, in the superclass documentation. In this class LoadList is empty and the constructor does its work.						
4.3			Memory alloctions not checked in constructor. May cause random runtime failures.						
4.4			Constructor uses Carray.add() method which can throw CMemoryException. This exception is not caught and is not documented as potentially thrown. This causes doubt that it is handled cleanly.						
4.5			Construction of LookupItem class instances in the ResponseTypeList class constructor highlight a defect in the design of LookupItem. In the ResponseTypeList constructor a sequence of LookupItems are constructed and then each field within the constructed object is set. This indicates that perhaps LookupItem should have a constructor form which allows the setting of these fields directly. Further it raises the question of whether or not LookupItem is a mutable or immutable class. It looks immutable which makes the public exposure of set methods inappropriate.						
	THSCore	98145.451							
5.1			Class level comment is present, but is too vague to be helpful.						
5.2			LoadList method comment is cut and paste from the superclass (GenericLookupList). This is a literal "cut and paste" since the comment still states that the method is a pure virtual method, which is clearly not true. The LoadList method is implemented by this class.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

5.3			LoadList method is cut and paste from the RadioList class. Again, this is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are idenical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					
	THSCore	TargetDescriptionList.{cpp,h}						
6.1			Class level comment is present, but is too vague to be helpful.					
6.2			LoadList method comment is cut and paste from the superclass (GenericLookupList). This is a literal "cut and paste" since the comment still states that the method is a pure virtual method, which is clearly not true. The LoadList method is implemented by this class.					
6.3			LoadList method is cut and paste from the RadioList class. Again, this is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					

Portability Reliability Maintain-ability Extens-ability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

	THSCore	TargetMarkingList.{cpp,h}						
7.1			Class level comment is present, but is too vague to be helpful.					
7.2			LoadList method comment is cut and paste from the superclass (GenericLookupList). This is a literal "cut and paste" since the comment still states that the method is a pure virtual method, which is clearly not true. The LoadList method is implemented by this class.					
7.3			LoadList method is cut and paste from the RadioList class. Again, this is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					
	THSCore	TargetTypeList.{cpp,h}						
8.1			Systemic failure to get source code to format cleanly into an 80 column editor. Lines are wrapped as they are displayed to the user and will not print to a standard printer cleanly.					

Portability Reliability Maintainability Extensability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

			LoadList method has poor internal documentation. This poor documentation causes me to wonder if it contains a defect, or if there is something subtle which is not explined. When considering the process for adding a target subtype, whether or not the target type exists affects the construction of the subtype. Specifically, if a target type exists then the SetAfatdsCode method is not invoked when the subtype is created; if the target type does not exist, then the SetAfatdsCode method is invoked when the subtype is created. This is pretty subtle and is likely to cause bugs since there is no supporting documentation for this subtle difference.					
			Methods getTargetType and isTypeNew are almost identical. They both iterate through the known list of types. One can easily recreate the functionality if isTypeNew by invoking getTargetType and comparing the return value against NULL. If NULL is returned then the type is new, otherwise the type already exists.					
			The public method "GetListOfDescriptions" does not sufficiently document how the Carray passed in as an argument is treated within the routine. As it is, it appends data to whatever is in the array at the time it is passed in. It is not unreasonable to expect that it would clear the data before adding its own. This is an extensibility and maintainability problem.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
			The public method "GetTargetSubTypeListForAfatds" reproduces the functionality of the method "GetTargetType(LPCTSTR lpszDesc)" in the process of doing it's job. The code which reproduces the functionality is 16 of the 21 lines in this method. This duplication of code is wasteful and makes the code less maintainable.					
			The public method "GetTargetSubTypeListForJvmf" reproduces the functionality of the method "GetTargetType(LPCTSTR lpszDesc)" in the process of doing it's job. The code which reproduces the functionality is 16 of the 21 lines in this method. This duplication of code is wasteful and makes the code less maintainable.					
			It is not clear why this class is not related to the GenericLookupList object. These two classes present a quite similar interface and yet are unrelated. This may be appropriate, but is suspect. This is a concrete instance of poor naming. There are many Classes which subclass from the GenericLookupList, they all have List in their names. This class also contains "List" in its name, but is unrelated. This is quite confusing. The documentation does not address this issue.					
	THSCore	TargetType.{cpp,h}						
9.1			Class TargetType subclasses directly from Cobject. Cobject is a Microsoft Foundation Class.					
9.2			Class TargetType uses Microsoft specific runtime type identification techniques (DECLARE_DYNAMIC,IMPLEMENT_DYNAMIC). The implementation language does not support runtime type identification. The use of this extension is not portable.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

9.3			The TargetTypeList class is declared as a friend class to this class. This is a violation of encapsulation. Friend classes are very dangerous and risk maintainability and extensibility.					
9.4			The signature of the public methods "GetListOfJvmfSubTypeDescriptions" and "GetListOfAfatdsSubTypeDescriptions" uses CStringArray. CstringArray is a Microsoft Foundation Class; this limits portability.					
9.5			The comments for the public methods "GetListOfJvmfSubTypeDescriptions" and "GetListOfAfatdsSubTypeDescriptions" do not sufficiently document how the consumer is to handle the CStringArray when done with it. Is it the responsibility of the caller to deallocate memory, or if the user deallocates memory will it corrupt the state of the TargetType object? This lack of clarity leaves it to the consumer to guess what to do and will cause unexpected memory corruption, or memory leaks that will make the application unstable when run for a long time.					
9.6			Class Constructor uses hard wired constants instead of defines for initial values for fields instead of enumerated values or #defines. This is a maintainability problem					
9.7			Class Constructor does not initialize the contained CObArray. This places too much trust in the implementation of the CObArray to clear itself on construction. There is a clear method on this contained object and it should be used. With it not being initialized, the state of the object is not entirely clear and so it is a testability and reliability problem.					

Portability Reliability Maintain-ability Extens-ability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
9.8			The public methods "GetListOfJvmfSubTypeDescriptions" and "GetListOfAfatdsSubTypeDescriptions" do not sufficiently document how the Carrays passed in as arguments are treated within the routine. As it is, they append data to whatever is in the array at the time it is passed in. It is not unreasonable to expect that it would clear the data before adding its own. This is an extensibility and maintainability problem.					
	THSCore	AFAPDTarget TypeList. {h,cpp}							
10.1			Class level comment is present and specific enough to be helpful; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					
10.2			LoadList method comment is present; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					
10.3			LoadList method is cut and paste from the RadioList class. This is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

	THSCore	AFAPDTarget MarkingList. {h,cpp}						
11.1			Class level comment is present and specific enough to be helpful; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					
11.2			LoadList method comment is present; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					
11.3			LoadList method is cut and paste from the RadioList class. This is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLooku-pList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					
		MTSTarget MarkingList. {h,cpp}						
12.1			Class level comment is present and specific enough to be helpful; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					

Portability Reliability Maintain-ability Extens-ability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
12.2			LoadList method comment is present; however, it exposes the internal implementation detail to the user. This limits the flexibility of the class consumer to treat the class as a black box.					
12.3			LoadList method is cut and paste from the RadioList class. This is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.					
	THSCore	TargetSub Type.{h,cpp}						
13.1			Class TargetType subclasses directly from Cobject. Cobject is a Microsoft Foundation Class.					
13.2			Class TargetType uses Microsoft specific runtime type identification techniques (DECLARE_ DYNAMIC,IMPLEMENT_DYNAMIC). The implementation language does not support runtime type identification. The use of this extension is not portable.					
13.3			The TargetTypeList and TargetType classes are declared as friend classes to this class. This is a violation of encapsulation. Friend classes are very dangerous and risk maintainability and extensibility.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

13.4			This class seems to support a very odd composition of methods. First of all, the commenting for the class does not make the purpose of the class clear. Without clear purpose, the meaning of the methods and fields is lost. It is interesting that this class supports methods to support JVMF, AFATDS, and F16 aircraft. It seems that this class encodes some knowledge about the F16 aircraft within it which is inappropriate. The knowledge of F16 policy for mapping AFATDS or JVMF target codes to the appropriate F16 codes should be encapsulated within an F16 class. If the F16 policy changes, it is not at all clear that the maintainer would know that this class had to be modified. Further, even if the maintainer did know to look here, the entire class would need to be unit tested again since the class would be modified.						
13.5			The method GetCode is not clear. Neither the method name, nor the comments for the method make it clear what code this is. Is it an AFATDS code, a JVMF code, an F16 code, or some other internal identifier which is not useful to anything outside of this class?						
13.6			The following two lines of code are present in this class: "long m_lCode; //JVMF Code", and "long m_lAfatdsCode; //AFATDS code--this is the JVMF code, but may map to another subtype". These lines are confusing and no further guidance is given.						
	THSCore	Severe WeatherList. {h,cpp}						
14.1			Class level comment is present, but is too vague to be helpful.						

Portability Reliability Maintain-ability Extens-ability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
14.2			LoadList method comment is cut and paste from the superclass (GenericLookupList). This is a literal "cut and paste" since the comment still states that the method is a pure virtual method, which is clearly not true. The LoadList method is implemented by this class.						
14.3			LoadList method is cut and paste from the RadioList class. Again, this is a literal "cut and paste." There is a single difference in the 68 lines of implementation code -- the SQL string used to select rows from the Database. Outside of this single 2 line difference, the methods are identical. This indicates that there is a common function between sibling classes (children of the same superclass, GenericLookupList) which should be handled at a higher level. This is not being done and may become a source of maintainability and testability problems.						
	THSCore	SpotPage.{h,cpp}							
15.1			There is no class level comment, and there are no method, or member variable comments.						
15.2			Data members of this class are scoped as Public. This class provides no encapsulation.						
15.3			This class subclasses directly from CPropertyPage, a Microsoft Foundation Class.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
15.4			This class mixes application functionality with display. This class handles the Spot report. Instead of collecting the information and forwarding it to a piece of the application which specifically knows how to deal with network and messaging issues, it creates the message as part of processing the GUI elements. This is a violation of the accepted Model View Controller design pattern. This is also a problem for reliability, portability, testability, extensibility, and maintainability. Since the system can not be fully tested without running the GUI, expensive and unreliable GUI testing tools must be used to test core messaging functionality. This is very complex and unnecessary. Due to the high cost of testing, it is unlikely that it is a financially viable option for each small change and so the system will be fielded without a complete test cycle. This will make the application unreliable. Additionally, since the superclass of this class is a Microsoft GUI class, this segment of the application is not portable. If this segment of code contained no application logic, it is possible that this segment could be replaced without tremendous risk (see MVC for details); however, with the intertwining of display and application logic portability is lost. Further, the integration of application logic into the GUI means that the GUI can not be easily changed without risking the functionality of the application. This will require a system test for a change to display layout. Lastly, anyone who wants to modify this application will be forced to learn the entire application since the application logic is scattered in inappropriate locations. This makes the system unmaintainable.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
15.5			Handling code for GUI needs to be refactored to use helper functions where appropriate. In the methods "OnDialogInit" and "UpdateFromA-FATDS" there is a piece of common logic which makes the same set of decisions based on a set of latitude and longitude values. This repeating of code may cause odd problems in the dialog (it bahaves differently depending on the set of steps taken to get to a specific place in the application). This common code should be factored out into a private helper method. This lack of factoring is pervasive.					
	THSCore	SpotSalute Page.{h,cpp}						
16.1			There is no class level comment, and there are no method, or member variable comments.					
16.2			Data members of this class are scoped as Public. This class provides no encapsulation.					
16.3			This class subclasses directly from CDialog, a Microsoft Foundation Class.					
16.4			This information contained in this class is a direct extension of the information contained in the SpotPage. The SpotSalute message supports a superset of the information in the Spot message. These classes are not related by inheritance or composition; hence, the code needed to support the common functionality is present in both classes. The reason for this is not entirely clear; however, it is likely due to the fact that the Model and the View are bound up together. This duplication of code may cause bugs in the future as bugs are identified in one of these messages and the developer does not know, or remember to update the other class with the bugfix.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

16.5			This class subclasses directly from Cdialog, a Microsoft Foundation Class. This MFC superclass is a View class, this subclass supports actual functionality and application logic. This blend is unhealthy and not safe. Please see Defect 15.4 for further details.					
16.6			There are a number of interesting initializations and tests which are undocumented in this class. The values of a number of fields are initialized to the value 0. In the OnOK() method, these same values are tested using the test {value}>-1. First of all, all of these tests and initializations are done with constantly defined numbers. This means that the developer will have to search through the code to understand how the static contants work together. This is a maintenance problem. Further, it is not clear what happens if the value IS less than 0. This class does nothing leaving the behavior up to the message class which is being created. I suspect that the message class defaults the value to some reasonable value; however, this class's correct behavior is now tied to the behavior of another class. This is dangerous and a potential source for bugs. This may be a reasonable assumption; however, without documentation explaining the choices, the next developer will have to make assumptions which may or may not be true.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
16.7			In the OnOK method, due to the design of this class (mixture of View and Model, which is bad and covered earlier) the method must go and get the correct C2 Network. The code which does this executes an interesting test. It gets a Carray of network Ids. If the number of C2 networks is greater than 0 it takes the first returned C2 network and uses it. There is no documentation supporting this choice. It is not clear that there can ever be more than a single C2 network; however, with a return type of an array, this is a source of potential bugs which will be quite difficult to find.					
	THSC2PC	ChildFrm.{h,cpp}						
17.1			This Class has no documentation, and is essentially empty. It does contain a number of comments of the form "TODO:". How this class interacts with the rest of the system is completely unclear.					
17.2			This Class directly subclasses from a MFC class. This is suspicious, but may be appropriate since this directory is labeled THSC2PC. I have been informed, though I have no documentation supporting or contradicting, that C2PC requires COM to be used for integration. If that is the case, then the use of an MFC class here may be necessary.					
	THSC2PC	THSC2PCView.{h,cpp}						
18.1			This Class has no Class level documentation in the header or implementation file.					
18.2			This Class's methods are mostly, if not completely, undocumented.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

18.3			This Class has the following 4 methods in it: OnAircraft1(), OnAircraft2(), OnAircraft3(), OnAircraft4(). These methods have no method documentation. There is a single comment within each method which is identical even though the methods contain subtly different logic within. (I had to review these methods multiple times before seeing the subtle differences; initially I thought them to all be logically equivelent.) The logic in these methods is virtually identical; indeed, the code seems to have started out the same (cut and paste). These methods also each use a magical constant during processing. The constant in question occurs 4 times in each method. (Two occurences are off by one from the other two; however, the variance is undocumented and looks like array indexing issues. Undocumented.)						
18.4			This Class contains a method OnC2Ft(). This method has no comment and so is undocumented. When I initially read the method name I assumed that it had something to do with using Feet as a unit of measure. This method instead pops up a FreeText dialog box.						
18.5			This Class contains a method OnCasClrha(). It is undocumented. In reading the implementation, it seems that this method is sending a Clear Hot Abort message; however, that is intuition since the there is no documentation.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
18.6			Method OnGps() violates MVC design pattern. It does some work which is appropriate; however, it also does some work which is application logic. In this method the application object is queried for GPS information and then decides whether or not to set the observer location. The decision to more the observer is not a function of the GUI and yet is implemented in a GUI class (which has "view" in the name).					
18.7			In the OnInfo method some interesting things are done. It seems as though THS takes all mouse click event from the system and consumes them through the use of a MouseEventsHandler COM object. This is a custom COM object defined by Stauder in this same module (THSC2PC). It is not clear to me that this is, or is not appropriate. It should be somewhat safe since these object files will almost certainly be in the same dll; however, it seems suspect.					
18.8			In the method OnShowLabels(), the Capplication instance and CMapHandler instance are each given instructions. It does not seem appropriate that the GUI is doing more than notifying the application (which is the embodiment of the Control in MVC) that an event has occurred. To do more, as in talking directly to the mapping system is to implement application policy directly. This may be appropriate if this system is implementing Model-View-Presenter; however, even then this would have to be investigated closely. Lastly, this is not the only method where this type of thing occurs -- this occurs throughout this class.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
18.9			In method OnZoomTracking, logic which should be part of the Control is implemented in the View. This method finds all of the points which are relevant to a CAS mission and zooms in to show only them and a small amount of extra space. In addition to this design flaw, there are implementation issues: (1) There is a great deal of repeated code for finding the extents of the area to be covered. (2) The code does not check to make sure that it is not selecting to display a single point as the entire screen - as the code is written this is possible, though not clear that it would occur in an operational scenario.						
18.10			The implementation of the OnClick routine concerns me. It seems that the way that mouse clicks are handled in this system is that the MouseEventsHandler captures the clicks from C2PC and queues them into a separate message pump thread for THS to handle. This in and of itself is interesting. What is further interesting is that the MouseEventHandler does not record the location of the click at the time that it occurs. This means that the event handler in this class must query the mouse for its current location when invoked. If the application begins to bog down, the mouse may not be in the same place when this method is invoked as when the click occurred. This is something of a boundary case, but it could negatively impact usability.						
18.11			The methods OnUpdateAircraft1, OnUpdateAircraft2, OnUpdateAircraft3, and OnUpdateAircraft4 are all identical except for a single constant which occurs two times in each method. This functionality should be factored into a helper function.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

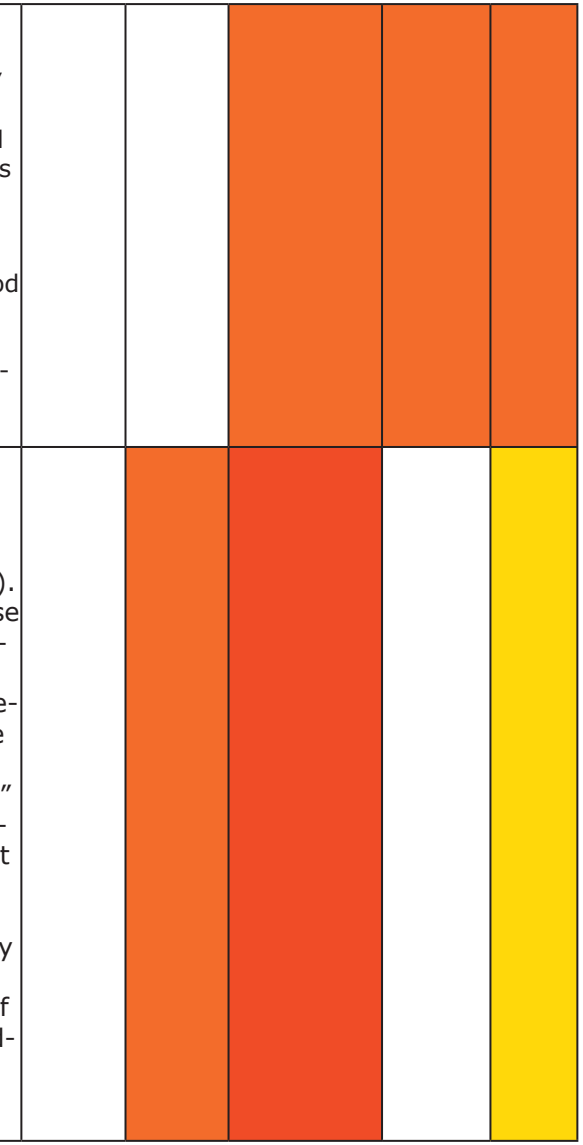
					Portability	Reliability	Maintainability	Extensability	Testability
18.12			There are a number of methods in this class (OnZoomIn, OnZoomOut, OnPan, to name a few) which are suspect in their interaction with COM. The set of methods which are at risk are the ones which deal with the CMouseEventsHandler. The standard operating procedure for these classes is to create an instance of the correct type pointer on the stack and then to call CreateInstance handing in that pointer. The result of this call is that the stack pointer now points at a valid instance of the CMouseEventsHandler. This new instance is configured and then installed in the containing object. I do not find any case in which the old object is destroyed or released. COM may be doing some garbage collection of interfaces, but will be unable to do so without calls to the Release method. I believe that the instances of the MouseEventsHandler class are being leaked. This may be a problem for they system if it runs for a long time.				
18.13			The code in the methods OnEgressPoint, OnEgressPoint2, and OnEgressPoint3 do not report out some possible, serious errors. Specifically, if there is not a single active CAS mission, the routines will ignore the point. This is unfortunate due to the fact that there are 3 cases to consider Case 1: there is a single active mission. That means that this is a sensible and appropriate thing to do (apparently). Case 2: there are 0 active CAS missions. This is suspect and perhaps should be logged. Case 3: there is more than a single active CAS mission. This is also suspect, and possibly an error. In any case, anything except the expected case will be silently ignored. This makes the application harder to test and debug.				

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintain-ability	Extens-ability	Testability
18.14			Magic value embedded in the THSC2PCView:: OnC2PCMouseUp. Comment: "If the point was a target and the user clicked the "CFF" button then we need to make a new CAS or CFF mission The return value of popupDialog will tell us which button got clicked but "IDC_CFF" is not defined in the .exe resource, so I hardcoded it's resource value (1012) from the core."		Severe	Severe	Severe	Severe
18.15			The method OnC2PCMouseUp is 610 lines long with no method documentation and sparse internal documentation. There are cases of code that should be refactored. The lack of refactoring is a maintainability and testability issue.			Moderate		Moderate
	Map-Wrapper	GeoPoint. {h,cpp}						
19.1			Single best Class level comment that I have seen in the system; single most frightening as well. This class is the "focal class for mapping applications." This class is in fact quite central to the proper function of this application. It supports not only name, geolocation, elevation, and coordinate system support methods, but also a popup window, a listener notification mechanism, serialization and deserialization, and database support (separate archive and db support). This class supports too much functionality and puts maintainability, extensibility, portability, and testability at risk.	Severe	Moderate	Severe	Severe	Moderate
19.2			The semantics for using this class are confusing. In the class level comment, it discusses how to create subclasses of this class. The subclasses that it discusses are supported by an enumeration within the superclass, which is a gross violation of encapsulation.			Moderate		Moderate

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

19.3			The semantics for using this class are confusing. In the class level comment, it discusses how to create subclasses of this class. In the comment for the "Line" type subclass, it outlines the differences between a point and a line (enumerated type set in this class and understood by the MapHandler). One of the differences is the following: "moving the object doesn't move the line. Instead, it moves the label associated with the line." This is a violation of polymorphism and how inheritance is to be used. The point of exposing a method at the superclass level is that each subclass treats the superclass method or data in the same way. In other words it means the same thing even though the implementation of the handling is different. This is a maintainability problem.					
19.4			The comments at the class level are detailed but confusing. Not enough support is given for the usage model which is proposed. For example: "Endpoints must be added on the stack (REPEAT STACK). Don't use new(). This is because endpoints don't use an Observable notification mechanism, so maintaining pointers isn't required. Also, we want to avoid confusing the system. We don't want to add a something like a target instance as an endpoint, because if we delete the shape, we don't know what to do with the target. So, we use the stack for endpoints." This line of reasoning is not clear. Creating the GeoPoint on the stack does not prevent the stack object from being a Target (which is a valid subclass of GeoPoint). This type of comment is very confusing and may cause serious maintainability and reliability problems. It seems as though the method which is used in the example (addEndPoint) makes a copy of the instance passed in. It is interesting that addEndPoint is not explicitly proscribed as the proper way to interact with this class.					



Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

19.5			The comments at the class level are detailed but confusing. The following comment is provided in support of the Circle subclass of this class: "understand that any changes that you make to this point don't act exactly like they did with the Point subclass. Specifics are: 1. deleting this object results in the map removing the associated line from the display. 2. moving the object doesn't move the circle. Instead, it moves the label associated with the line. 3. To draw the line, the circleCenter and radius must be set. As with LINE subclass endpoints, the circleCenter should be created on the stack, or the circle center moved directly. 4. Any change to circleCenter_ or to radius will NOT cause the circle to be redrawn. Your code must issue the notifyChanged." There are mixed references to "the line" in this comment, but it is a circle which is being described. This is confusing.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
19.6			Comments for method "getPointAtRangeBearing" are confusing and make using the method impossible without reading the implementation. Consider: "From this location, given the range, bearing (**TRUE NORTH**and elevation angle to another point, get the absolute location of that point. @param pointToFill point whose position will be filled by this @param lineOfSightRange to other point in meters. @param bearing to other point in radians (TRUE NORTH) @param elevationAngle to other point in radians Usage: use code like this: CGeoPoint myPosition("STL", 39.0, -40.0, 200); CGeoPoint otherPosition; getPointAtRangeBearing(otherPosition, 1050.0, 1.25 0.37);" The problems here are the following: (1) the parameter naming does not encode units, this makes the job of the method's implementer, maintainer, and consumer more difficult. (2) it is not clear what the valid ranges for the input parameters are (3) it is not clear what will happen if the parameters are not in the valid, undocumented range. This method returns a void, not a bool, and item passed in is a GeoPoint it does not have an explicit validity check in it. How are you to know if the method failed?				
19.7			The constructors do not range check the input values. This problem seems to be pervasive throughout the implementation of this class, and likely through all classes. Range checking input values is good defensive programming and is very useful to reliability and testability. With range checking, errors can be identified early and debugged when they occur -- not at some later date where the time that the corrupt data was introduced is no longer clear.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
19.8			The methods enableNameChangeInPopup and isPopupNameChangeEnabled show good, clear naming; however, these methods are another indicator that this class is too intimately associated with the display of information. Display of information is not intrinsic to a location and so should be separate. Some points may allow for name changiPoPng and some may not; however, there is an intrinsic meaning these cases and they are likely tied up in what the point represents. The representation should make the name changeable policy decision and the information should not be stored in the GeoPoint class.					
19.9			The method "setDtedFileName(LPCTSTR fileName)" exists in the header file and has an implementation. This method has multiple problems: (1) The comment references a method which does not exist, "setElToDted()". The method's actual name is "setElevationToDted()". (2) This method's implementation does nothing. The body of the method is commented out in it's entirety. This in and of itself is inelegant, but more imporatantly the programmers model of how this class should work is not correct. If this method is invoked by a consumer who does not read the implementation, haeaningful value and it does not log an error or warning to any log file. The consumer of the method will have no way of notifying the user that the operation failed.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

				Portability	Reliability	Maintainability	Extensability	Testability
19.10			The method "getEndpoints(CArray<CGeoPoint*, CGeoPoint*>& arrayToFill)" does not document how the consumer is to treat the contents of the Carray returned to the caller. This may cause misunderstandings and memory corruption. Further, this method uses an MFC datastructucture in the method signature. This severely limits portability.					
19.11			The methods "RestoreState" and "SaveUndoInformation" are both declared virtual in this class; however, they methods have empty implementations. This indicates that they are either empty in error, or that they should be defined as pure virtuals so that subclasses are obligated to implement them. The way it is done now is an odd mix which may lead to interesting defects. If a subclass overrides one method but not the other what will happen is nothing good.					
19.12			The method "Save" contains an embedded SQL string which is tied to a specific database schema. One thing which should be considered for future work is that the information which defines a GeoPoint may change (especially since it is so broadly defined in this system). At this point, the developer would have to modify the embedded SQL strings in this class to be able to extend or modify the set of fields that define this class.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

19.13			The code to save and serialize the contents of this class are completely independent (though there may be some interdependence between them in usage, it is implied in the documentation but not explicitly addressed). This is a maintainability and reliability problem. The database serialization and archive serialization do not save the same information. It is not clear why and which is correct, or even if both are correct and to be used for different purposes.					
19.14			The code to save the contents of this class contains an embedded sql string					
19.15			There are a number of SQL generating methods in this class. The getInsertString method is implemented but not used. The "Save" method generates it's own SQL string on the fly. This is a place where defects are likely to occur. The other methods which generate SQL seem to be used, so this seems to be a single deviation which make is more difficult to identify and address by the consumer of this class or maintainer of this code. This is a bug waiting to occur if it is not occurring already.					
19.16			The methods which interact with the Database catch the database exceptions. In the catch statements there are comments which indicate that some error handling should be done; however, no error handling is being done. Some routines fall through the handlers and return the same return codes on success and failure. This will lead to database corruption and unpredicatable behavior. Error handling is hard, and presently is not being done consistently. This is a risk to maintainability, and reliability.					

Portability Reliability Maintainability Extensability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
19.17			The public methods which may throw exceptions have documentation advertising the exceptions; however, exceptions are not documented as potentially thrown for the protected members when they may throw them. This causes doubt that they are handled cleanly. Even if they are properly handled presently, without supporting documentation telling the consumer of the methods that Exceptions can be thrown this is a serious maintainability issue. If they are not properly handled presently, they are a serious reliability issue.			Moderate	Severe		
19.18			The method "getElInMeters" has a comment which reports that the value returned is in feet. This is either a case of cut and paste documentation, or extremely poor member naming.				Severe		Moderate
19.19			This class uses "helper" functions in saving, loading, and deleting information from the database. These functions, PostSave for example, modify the data in the database. It is not clear what would happen if one of these methods failed. The PostSave method always return the same return value regardless of whether or not they were successful. In general, no transactions are used in this system that I can find. This means that multi-step database processes like this are risky.			Moderate			Moderate
	THSCore	StauderShape.{h,cpp}							
20.1			The setType method fails silently if the argument passed in as the requsted type is not in the valid type list for the instance. This silent failure will make maintainability, extensibility, and testability harder to achieve.			Moderate	Moderate	Moderate	Severe

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

20.2			The StauderShape class contains public methods which overlap with the functionality presented by its direct superclass, GeoPoint. Specifically, the methods "GetElevation(Celevation&)" and "SetElevation(Celevation&)" are present in StauderShape and GeoPoint supports "double getElevation()" and "getElInMeters()". StauderShape does not contain an elevation member variable, which is good since the superclass does contain an elevation member variable, so it should not contain an elevation accessor member function. The methods in the subclass do use the accessor member functions in the superclass to do their work; however, placing the methods to handle elevation in two distinct places is poor design and limits maintainability and extensibility.					
20.3			The member functions Load, Save, and Delete are virtual and override the superclass versions. This is a good practice if implemented properly; however, a key principle has been missed in this implementation. The superclass GeoPoint stores the following set of information when Save is executed: latitude, longitude, elevation, mapSelectable, dialogAvailible, CirclePointsCreated, subtype, drawtype, radius, changenotifyenabled, parentgeopointid, rotationangle, and geopointID. This class, StauderShape, stores the following information when Save is called: GeoPointID, latitude, longitude, name, elevation, object_type, and locked. There are a number of problems here:					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

20.3.1			1) The save method in StauderShape does not invoke the save method in the superclass. By inspection it seems that the following things are true: a) the subclass is not storing all of the information availible in the superclass. This means that the superclass will not be placed in the same state when loaded again. b) the subclass must be aware of the structure of the superclass. This is a significant violation of encapsulation.					
20.3.2			2) The StauderShape and GeoPoint save, load, and delete methods are not interacting with the same database tables. It seems as though the GeoPointIDs which are being assigned are not unique system wide -- only within each class of objects. A GeoPoint and a StauderShape may share a GeoPointID number.					
20.4			StauderShape contains methods which do not seem appropriate in this class. Specifically, the methods "SetRangeBearingToObserver()", "GetAzimuthToObserver()", and "GetRangeToObserver()". These methods all begin by getting the Observer object instance from the application. These methods may be helpful; however, they seem to be convoluted as structured. It seems that they would be more natural as members of the CObserver class.					
	THSCore	Target.{h,cpp}						
21.1			This class inherits from StauderShape, and hence also from GeoPoint. GeoPoint defines a number of drawing shapes in an internal enumeration. Target redefines a number of these drawing shaped in another enumeration. This is confusing and poorly documented. There is not documentation addressing why the inherited type can not be used.					

Portability Reliability Maintainability Extensability Testability

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintain-ability	Extens-ability	Testability
21.2			This class defines the TargetManager class to be a friend class. This is dangerous since it leaves the functionality of Target and TargetManager in question if either Target or TargetManager is modified.						
21.3			The Target class is supporting to many features within it. The fact that the class contains an enumeration for target shape, and that a number of methods change behavior depending on the current target shape is a problem.						
21.4			This class seems to have a semantic problem with some of its methods. The "SetType" method has the following comment: "Inherited method from CStauderShape. Do Not Use This one. To change the type of a Target, use SetTarget-Shape and SetMissionTarget. @see SetTargetShape, @see SetMissionTarget" This is a significant problem! This indicates that the semantics of this one class are not the same as the rest of the system. This is a maintenance nightmare. Even developers experienced in the use of this system can be expected to make significant errors in the use of this class. Lastly, it is quite troubling that the method's implementation takes no action if invoked. It simply does nothing silently. This will be a rather nasty bug to track down.						
21.5			Cut and paste commenting for the GetKnownPoint method -- the comment is incorrect.						
21.6			The KnownPoint functionality in the Target class is not explained to the user at any point. It is not clear how it is to be used or what functionality it provides.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
21.7			In the operator = method, which does assignment, there is an unanswered question: "Do we need to copy the Known Point?" This is an excellent question. It is not answered, but the code does not do so. This is troubling since it seems that the developer is not clear on how this method is to work. The method comments are not clarifying or helpful in this regard.						
21.8			The comment for SetTargetShape is bad for a number of reasons. First of all it exposes the consumer to the implementation detail of the method. Secondly, the detail which is discussed is not clarifying.						
21.9			The Draw method of this class clearly shows that this Class should be subclassed. This method contains a switch within it. Each case is completely independent. This is a clear indicator that there should be subclasses and that this class should be pure virtual in this class.						
21.10			The methods SetLength, SetAttitude, and related methods all share a single, prevasive defect. These methods are used to change the drawable representation of this class. Not all of these drawable dimensions are appropriate for each of the shape selections. These methods all change the value which they are to change, and then they test the shape type to see if it has an effect of the drawn representation. This is poor because it is likely that if you are calling a method which has no effect on the system, it is being called in error. This should at a minimum be logged to a system log somewhere so that it can be debugged. This is another case of silent failure						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability

21.11			In the method SetAttitude, the input variable is bounds checked… a little. This method contains the following code: "if(attitude.GetRadians() > M_PI) { //make sure the saved attitude is between 0 and 3200 mils attitude.SetRadians(attitude.GetRadians() - M_PI);
			}
			m_dAttitudeRadians = attitude.GetRadians();
			
			This code does not ensure that the store value is in the valid range as specified in the comment. First of all it does not check for a negative value. Secondly, if any value over (2*M_PI)+1 is passed in this routine will not live up to the comment. In general, this is far better than most of the rest of the system, in that it attempts to validate the arguments to the function; however, it is lacking in that fails to actually validate.					

21.12			This class, as do the classes it inherits from, tries to track the state of the persistent storage instead of just querying the persistent storage. This class, StauderShape, and GeoPoint all have a variable of the form m_bIsNew (though CGeoPoint actually calls the member isNew_). This variable is used to try to keep track of whether or not this point has been saved to the database. Another, more robust, strategy is to simply see if the data is in the database before trying to insert, modify, or delete an entry.					
21.13			There is a defect in the Database schema definition of a Target. In the table, the attitude column is misspelled "atitude" Not fatal, but an issue for maintenance long term.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
21.14			The method GetTargetNumber is made to leak memory. This method allocates memory on the heap and returns a pointer to it to the user, but does not document that the user is getting heap memory that they must deallocate. This is quite poor, because the user can not deallocate the memory without knowing that they should, AND how it was allocated (new vs malloc). If not now, in the future, use of this method will cause memory leaks which will impact long term stability of the system at runtime.					
21.15			The Save, Load, and Delete methods on this object use the more appropriate form of virtual class overriding for this application and chain up to the superclass to store it's own data instead of trying to do it independently (as StauderShape does with GeoPoint). That said, with the chaining, it is not clear that the errors are being handled in a sensible way. If the superclass form fails, the the subclass form is already doomed -- it is not even clear what it means if the subclass logic executes correctly at that point. The superclass errors are caught and logged, but then the subclass logic is executed anyway. This feels like an odd defect waiting to occur.					

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
21.16			The storage of TargetType in this class, and the interaction of that stored type and the TargetManager is unusual. The target types are stored as strings, and the TargetManager seems to store a set of TargetTypes which are able to convert the TargetTypes to platform specific target types. This seems odd. It seems that the Target could contain its type in addition to a string description. As it is here, it seems that the type encoded in the target is not directly useful, which is unfortunate. A Target should be a well defined atomic thing; as it stands it can not be used without being able to access the TargetManager.						
	MapHandler	ObjectDialog.{h,cpp}							
22.1			Constructor does not check arguments to validate. Dereferences one of the arguments. If that argument is NULL the thread will behave badly at a minimum and likely exit. This is a reliability problem. There is a single, brief comment in the header file which is "// don't use NULL!" While a helpful hint, this doesn't substitute for good coding practices.						
	THSCore	TargetManager.{h,cpp}							
23.1			Memory allocations not checked in Load method. This may lead to unexpected and unpleasant behavior at runtime. If the memory allocation fails, the routine may either cause the executing thread to exit unexpectedly, or worse corrupt memory and continue execution.						
23.2			This class implements two Delete methods. These methods take different arguments: a string vs an index. Other than how they find the item to delete, they execute the same exact proceedure for the actual deletion. This is a place where refactoring of code would make it more maintainable and hence reliable.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

					Portability	Reliability	Maintainability	Extensability	Testability
23.3			The code in the method CreateTarget(CGeoPoint*) makes assumptions about the rest of the software with which it interacts. Specifically, there is a piece of code which does not specify an else for an if which checks modes. In short is says the following: if (mode1) {do stuff} else if(mode2) {do stuff}. What this does not do is notice if there is a mode3 which is selected. In a situation such as this, it is good to check and at least log that something unexpected has happened. Right now, it is another silent failure in the system.						
23.4			Related to defect 23.3, having a decision of the type outlined above (which is making a decision based on the application mode) indicates that the application is poorly structured. In general, a method should have a single job that it can do without querying the application, or some other large unrelated, piece of knowledge in the system.						
23.5			The application and it's database structure and interactions, are structured with the implicit assumption that items can be created and deleted at almost anytime in the lifetime of the system. Specifically, targets can be deleted so long as they are not part of an active mission. This is a significant restriction which is deeply embedded in the system. It calls into question the auditability of the system and the ability to replay past events to understand what has happened.						

Minor	A Minor defect is one which presents low risk to application functionality at present, or which would be an irritation in manitenance.
Moderate	A Moderate defect is on which presents a measurable risk to operational capability at present, or which could easily cause measurable risk to be introduced in maintenance mode.
Severe	A Severe defect is one which presents a significant and immediate risk to operational capability, or which makes the system unmaintainable.

23.6			The Load method in this class will allow the load of a Target from the database to fail and then continue. On failure to load a Target, it will write a message to a logfile and then add the Target into the TargetManager list of Targets anyway. This will leave the list of Targets in an interesting, and likely corrupted, state. It is not clear how the system will behave in a situation where a Target in the TargetManager is not validly constructed. This behavior is likely helpful to a developer; however, it is not helpful to the end user in the field.					
23.7			The Targetmanager assign target numbers to new targets. Each TargetManager is initialized with the same values used to create target numbers. These values are valid. On the execution of the Load method in this class it tries to reset the values to things loaded from the database. If this load fails, a message is written to a log and the system continues on using the default values. This is another case where the messages will be useful to a developer during debugging, but it will not help the end user in the field. It is not clear that it is or is not important to be able to generate unique target numbers over a set of more than a single FAC/FO.					

Appendix D

Index

Introduction

CMU's evaluation of the THS(X) software as provided by Stauder included several runtime usability tests on the Rugged Handheld Computer platform. Two identical RHCs were used. The software on RHC #1 was configured by Stauder at their offices in St. Peters, Missouri. RHC #2 was configured at CMU using the process and materials provided by Stauder

(See Appendix A). The tests were conducted over the period of a week, by two individuals.

Resource Utilization

As part of usability tests, resource utilization data was collected. The results were poor. On RHC #2, THS(X) constantly used from 98%-100% of the CPU, regardless of user activity. This condition could also be seen sporadically on RHC #1.

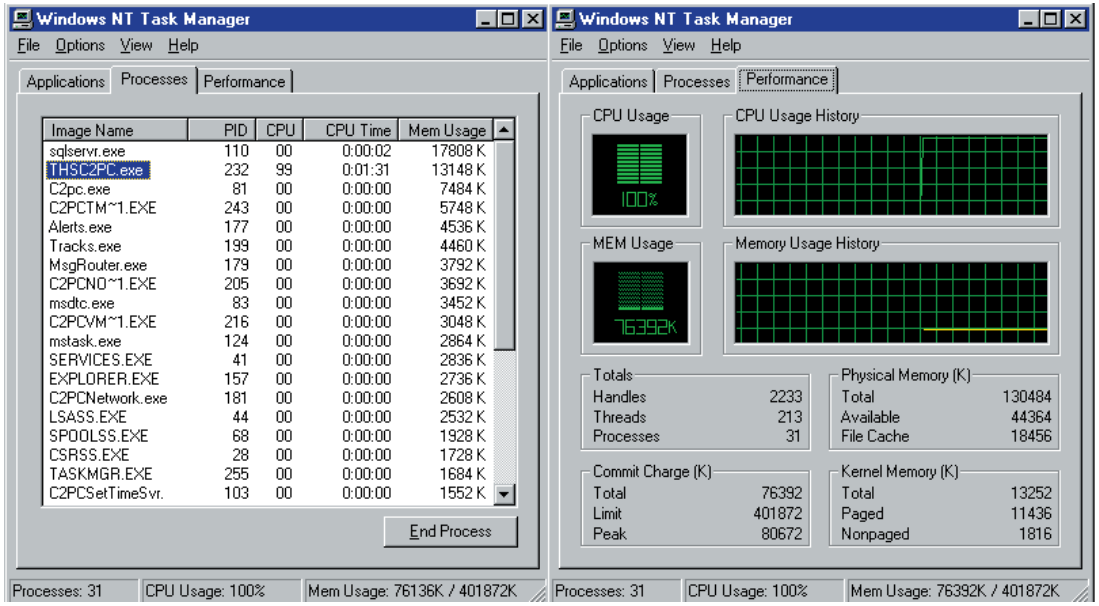


Figure A-1*: CPU usage in idle state*

The THS(X) install on RHC #2 also consumed memory at the rate of approximately 533 Kb per hour without user activity. This condition was also seen sporadically on RHC #1 After 39 hours and 38 minutes of THS(X) running without user activity, RHC #2 became completely unresponsive and a hard reset was required. After the reset, the RHC was unable to detect a hard disk as being present. The graph below represents the idle memory consumption of the period of 2 hours. Optimally this should a linear curve with a slope of zero; THS(X) instead displays a linear curve with positive slope.

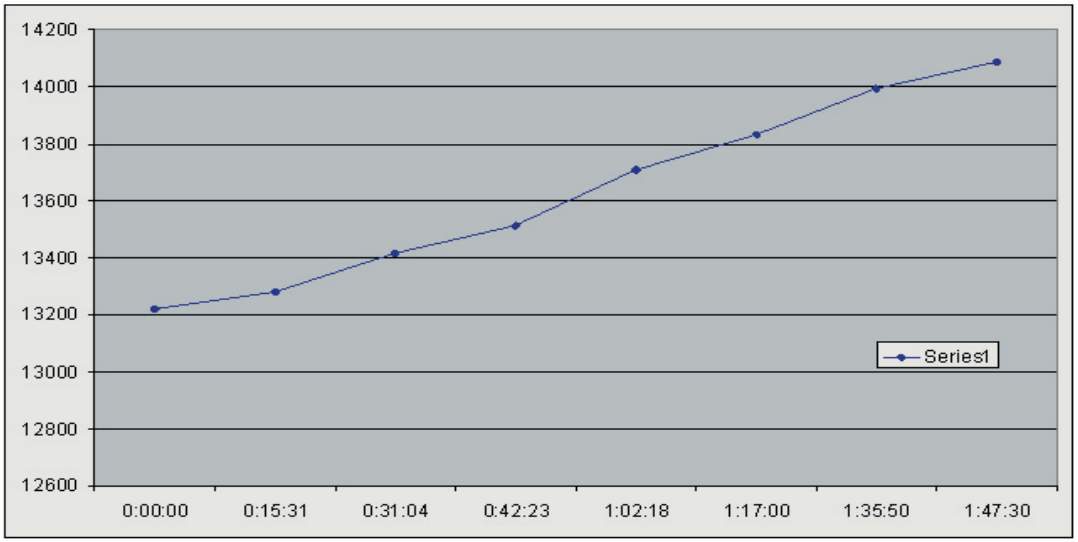


Figure A-2: *Idle state memory consumption over time*

Time	Memory Consumption
0:00:00	13220 K
0:15:31	13280 K
0:31:04	13416 K
0:42:23	13512 K
1:02:18	13708 K
1:17:21	13832 K
1:35:50	13992 K
1:47:30	14088 K
16:43:27	22020 K
19:37:40	23544 K
21:20:40	24468 K
24:11:20	25968 K
39:38:21	34208 K

Table A-1: *Idle state memory consumption*

Tabular Summary of Runtime

Successful Terminations	Brief Description	Occurs on RHC#1?	Repetitions	Occurs on RHC#2?	Repetitions
1.)	Started with Server Busy and LH41C errors and exited normally	Yes	2	No	N/A
2.)	Started with LH41C error and exited normally	Yes	5	No	N/A
3.)	Started normally and exited normally	No	N/A	Yes	8

Table A-2: *Sucessful terminations*

Fatal Errors	Brief Description	Occurs on RHC#1?	Repetit-ions	Reproduc-ible?	Occurs on RHC#2?	Repetitions	Reproducible?
1.)	LH41C Error with Server Busy window	Yes	10	*	No	N/A	N/A
2.)	Department of Defense Warning Exit	Yes	6	Yes	Yes	4	Yes
3.)	LH41C: Mk7Interface Port Error	Yes	4	*	No	N/A	N/A
4.)	Visual C++ Runtime Error	Yes	2	*	No	N/A	N/A
5.)	Department of Defense Warning with Server Busy window	Yes	5	*	No	N/A	N/A
6.)	Duplicate Mission Name	Yes	7	Yes	Yes	4	Yes
7.)	Initial Point Creation	No	N/A	N/A	Yes	1	No
8.)	Opening Log File	Yes	2	Yes	No	N/A	N/A
9.)	Creating Quick Fire Plan	Yes	1	No	No	N/A	N/A
10.)	Adding Target to Duplicate Fire Plan After Removing a Target	Yes	2	+	No	N/A	N/A
11.)	Closing THS(X)	Yes	2	*	No	N/A	N/A

Table A-3: *Fatal Errors*

+ Occurred multiple times, occasionally able to be reproduced

* Occurred multiple times, unable to be reproduced at will

Summary of Errors Causing Involuntary Termination

The following is a brief description of all the fatal errors encountered while using THS(X). If the system was still responsive enough to take one, a screen shot is included with the description.

Fatal Error #1: LH41C and Server Busy Error (occurred on RHC#1 and RHC #2)

When the LH41C error appeared in conjunction with the Server Busy error, the program entered a completely unresponsive state. The program must then be halted via the Windows NT Task Manger. After the process was stopped the system entered a completely unresponsive state, requiring the manual reset of the RHC to return to a usable state.

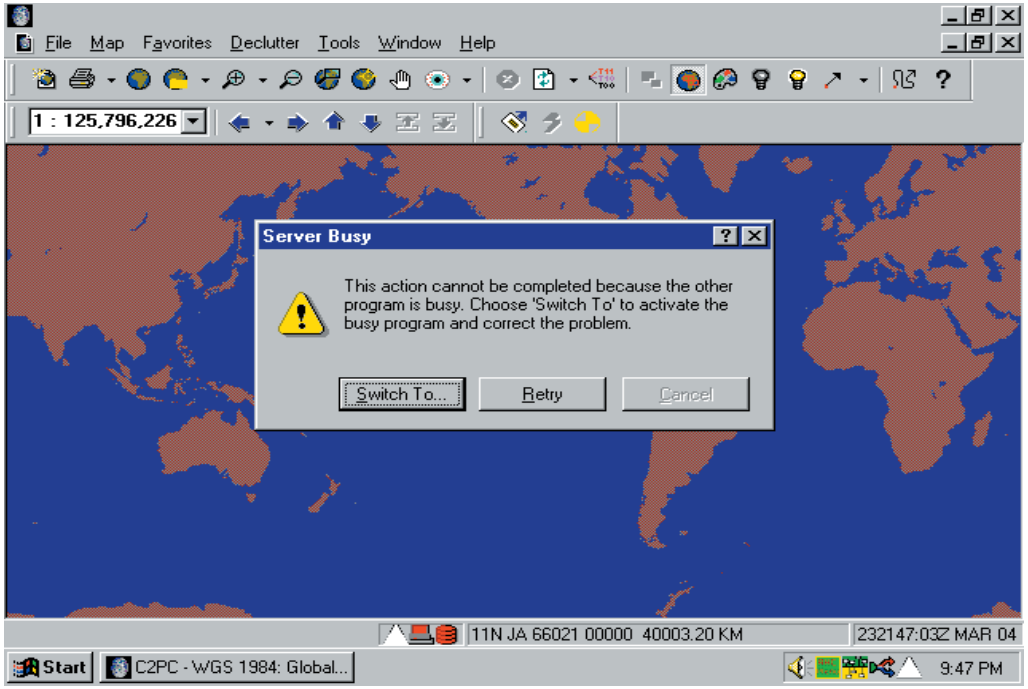


Figure A-3*: Server-Busy Window*

Fatal Error #2: Department of Defense Warning Exit (occurred on RHC #1 and RHC #2)

Clicking exit on the Department of Defense warning resulted in the entire system entering a completely unresponsive state, requiring a manual reset of the RHC to return to a usable state.

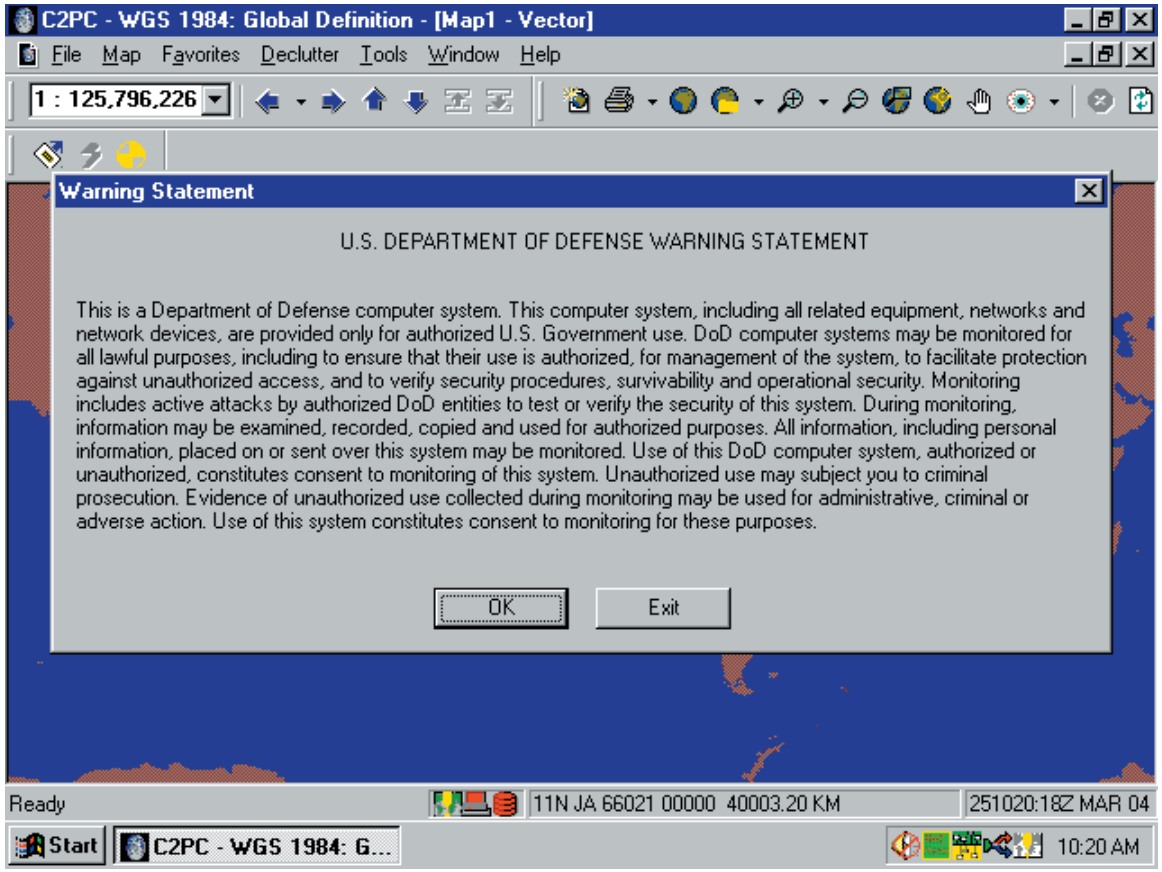


Figure A-4: *Department of Defence Warning*

Fatal Error # 3: Mk7Interface Port Error (occurred on RHC #1)

This error occurred when starting THS(X). Each time this error occurred, it was accompanied by the 'Server-Busy Error'. After this error occurred, two instances of THS(X) were noted as running, according to the Windows NT Task Manager.

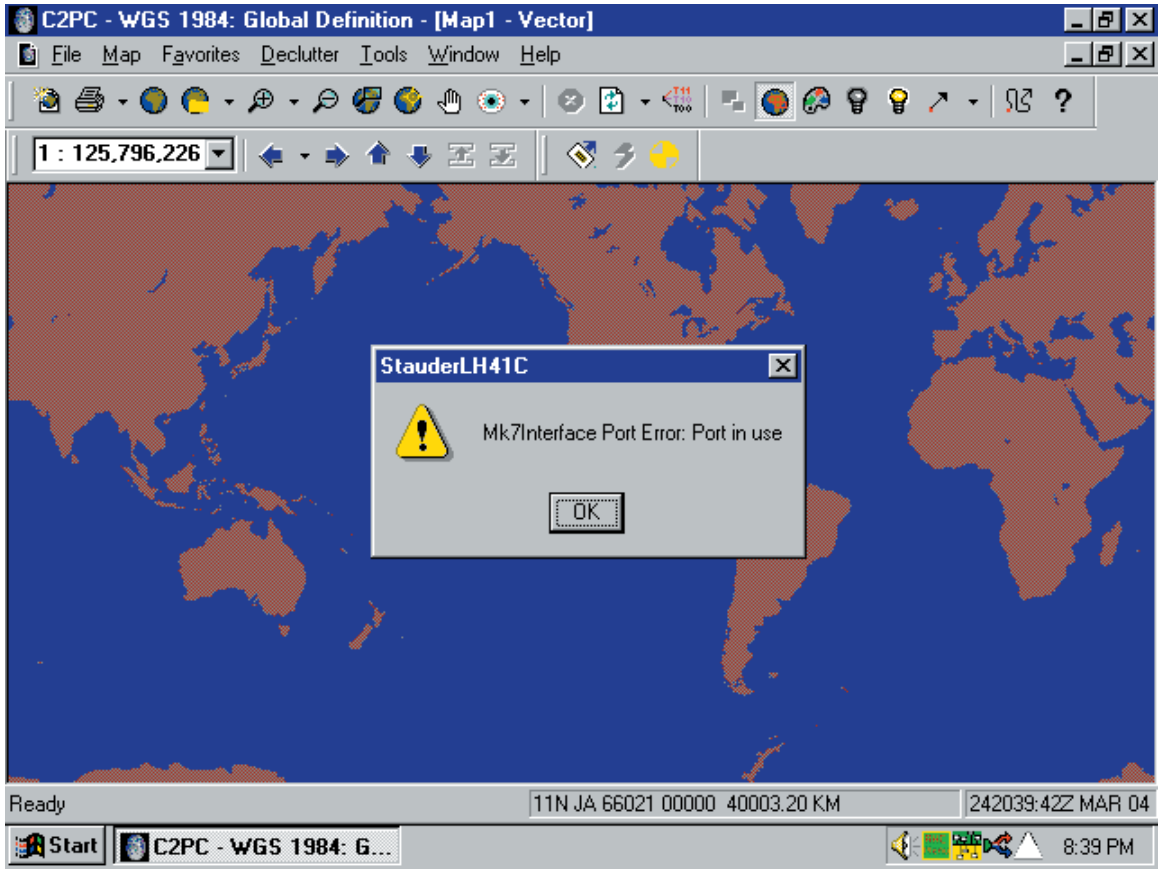


Figure A-5*: Mk7Interface Port Error*

Fatal Error #4: Microsoft Visual C++ Runtime Error (occurred on RHC #1)

This error occurred while attempting to start THS(X). After this error occurred the system was not responsive enough to take a screen shot. This error was fatal to both THS(X) and the operating system, requiring a manual reset of the RHC to return to a usable state.

Fatal Error #5: Department of Defense Warning with Server-Busy Error (occurred on RHC #1)

This error occurred while attempting to start THS(X). After the Department of Defense Warning Statement appeared, the Server-Busy window appeared. After attempting to clear the Server-Busy window (by clicking '*Switch to…*' several times), the application crashed. Windows NT then generated a Dr. Watson error dialog with the following text:

> "An application error has occurred
> and an application error log is being generated.
> THSC2PC.exe
> Exception: access violation (0xc000005), Address: 0x00000000"

At this point, the system was not responsive enough to take a screen shot. This error was fatal to both THS(X) and the operating system, requiring a manual reset of the RHC to return to a usable state.

Fatal Error # 6: Duplicate Mission Name (occurred on RHC #1 and RHC #2)

This error occurred when a mission with a duplicate name is created. When a new mission was created, if the name field has the same name as a previously created mission and the create mission dialog box is closed, an error dialog appeared. The error dialog reappeared after every attempt to close it. THS(X) was then closed through Task Manager, and the RHC was rebooted in order to clear the dialog box that persists on the desktop. In addition, when THS(X) was started the next time, a new mission with a duplicate mission name had been added to the working missions list. The duplicate-named mission could then be opened, and unless the name field was altered the error repeated.

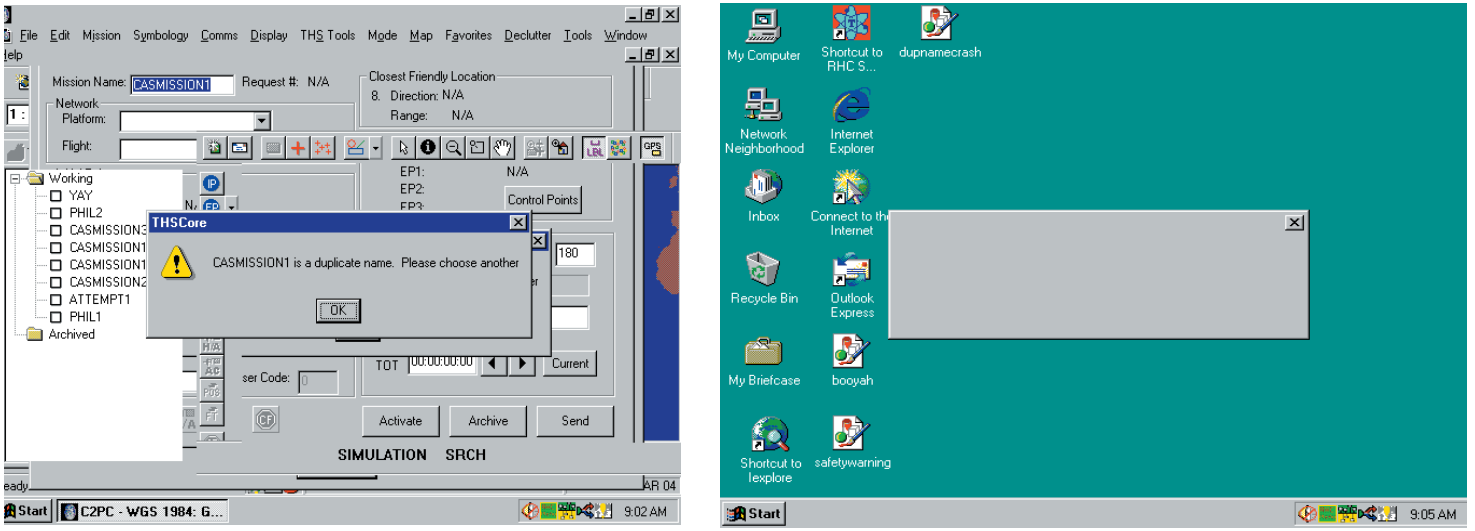


Figure A-6: *Duplicate Mission Name Error*

Fatal Error #7: Initial Point Creation (occurred on RHC #2)

This error was not able to be reproduced due to the fact that it was either intermittent or influenced by factors that were not considered when recording the steps that led to the crash.

When creating an initial point for a mission was attempted, THS(X) and the RHC stopped responding. A manual reset of the RHC was required.

Fatal Error #8: Opening Log File (occurred on RHC #1)

When attempting to view a log of events from the Audit File Maintaince tool, the program became unresponsive. THS(X) and sqlserver were contending for CPU time, and rapidly consuming memory. This crash was fatal only to THS(X).

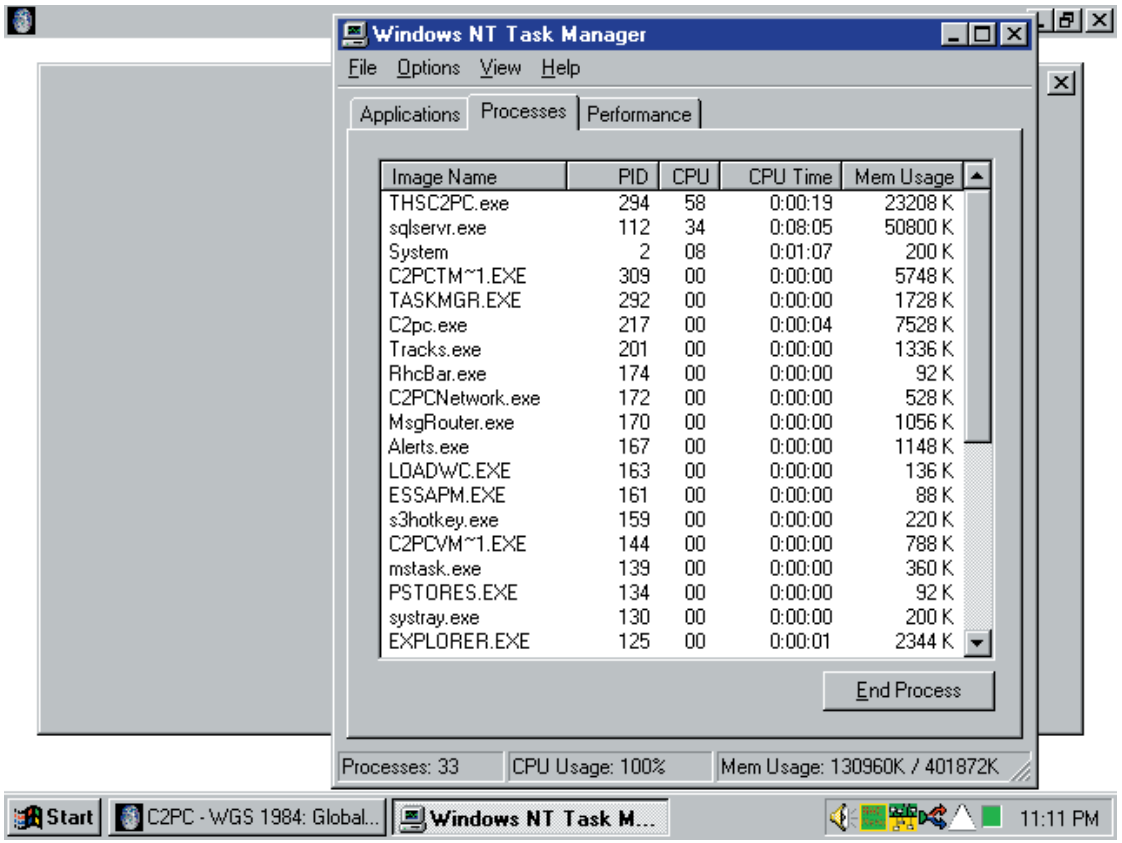


Figure A-7*: Opening Log File Error*

Fatal Error #9: Creating a Quick Fire Plan (occurred on RHC #1)

This error occurred while using the 'Quick-Fire Plan' portion of the Fire-Planning tool. When 'New Plan' was clicked, the application and operating system entered an unresponsive state and a hard reset of the RHC was required. The system was not stable enough to capture a screenshot for this error.

Fatal Error #10: Creating and Editing Duplicate Fire Plans (occurred on RHC #1)

This error occurred while using the 'Quick-Fire Plan' portion of the Fire-Planning tool. Once a fireplan was created, it was possible to create second fireplan of the same name. When this was done, the program and system became unstable while editing (adding and removing targets) the plans. Eventually THS(X) would crash and a hard reset of the RHC was required.

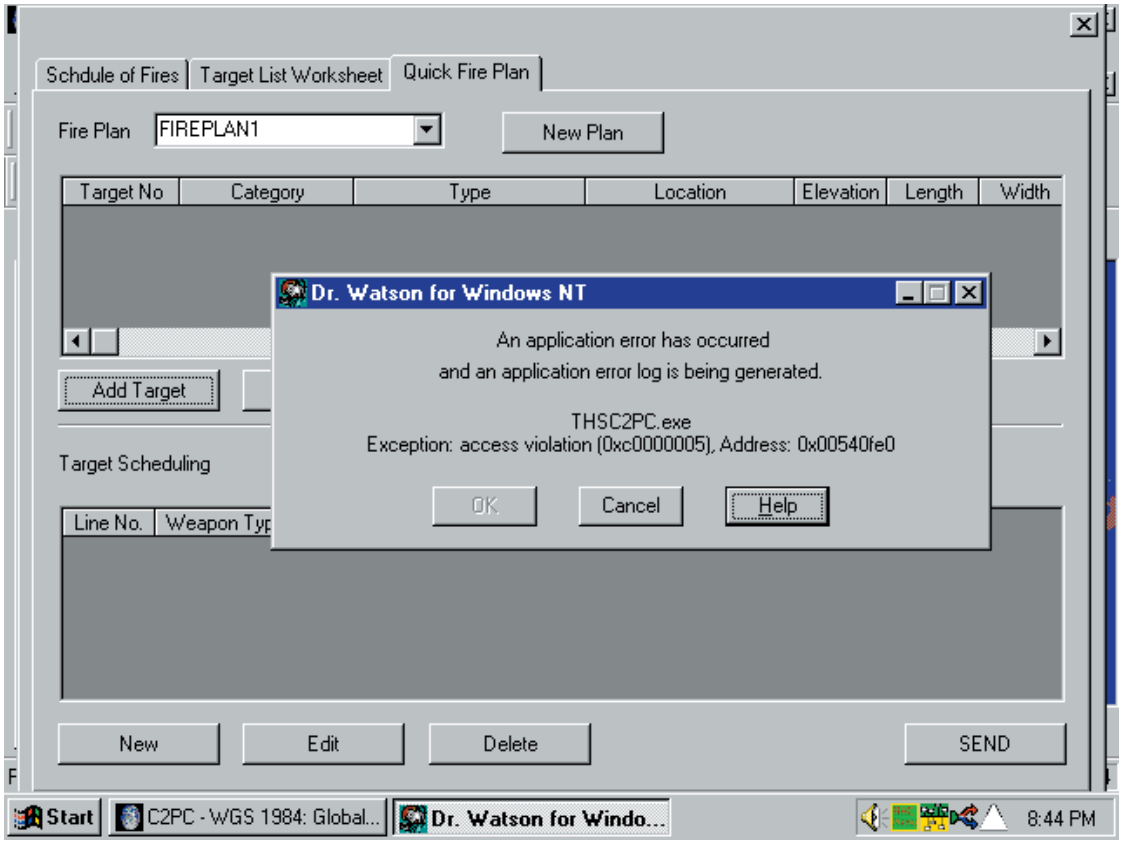


Figure A-8*: Duplicate Fire Plan Error*

Fatal Error #11: Closing THS(X) This error only occurred on RHC #1.

This error occurred after a successful start of THS(X). When attempting to exit THS(X) via the 'Close' button, THS(X) and the operating system entered an unstable state and a hard reset of the RHC was required.

Paths to Errors Causing Involuntary Termination

The following is a summary of errors that occurred in the THS(X) software. These errors resulted in THS(X) entering an unstable and unusable state. Several of these errors were fatal not only to THS(X) but to the operating system as well. Listed below are the number of times a given error occurred, the system on which it occurred, and the path of operation leading to the error.

Fatal Error #1: LH41C: Failed to Load System Status Settings error with Server Busy Window

Path 1 (10 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	Error LH41C will appear, after 10 seconds Server-Busy Window appears
3	At the Server Busy Window, click OK repeatedly
4	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #2: Department of Defense Warning Exit

Path 1 (6 Repetitions on RHC #1, 4 Repetitions on RHC #2)	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click EXIT
3	At this point, the program stops responding
4	Close the program in Windows Task Manager
5	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #3: LH41C: Mk7Interface Port Error

Path 1 (4 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	Error LH41C appears, contains the text: *"MK7Interface Port Error: Port in Use"*
3	Server Busy Window appears, click 'Switch-To' repeatedly
4	C2PC becomes unresponsive, kill process via Task Manager
5	Error LH41C appears, contains the text: *"Failed to load system status settings."*
6	At Error LH41C, click OK
7	At Department of Defense Warning click OK
Path 1 (4 Repetitions on RHC #1) – Continued	
Step	Procedure
8	NOTE: At this point nothing is drawn to the screen
9	Check Task Manager, 2 instances of THS(X) are running
10	End one of the THS(X) processes
11	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #4: Visual C++ Runtime Error

Path 1 (2 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	Microsoft Visual C++ Runtime Error Window appears, contains the text: *"Program E:\THS(X)\THSC2PC.EXE* *R6025* *- pure virtual function call"*
3	At Visual C++ Error Window, click OK'
4	Clear Hot/Abort: THSC2PC.EXE - Application Error Window appears, contains the text: *"The instruction at '0x7710510e' referenced memory at '0x00000048.' The memory could not be 'written'.* *Click on OK to terminate the application."*
5	At Application Error Window, click OK

6	Clear Hot/Abort: THSC2PC.EXE - Application Error Window appears, contains the text: *"The instruction at '0x00000000' referenced memory at '0x00000000.' The memory could not be 'written'.* *Click on OK to terminate the application."*
7	At Application Error Window, click OK
8	Clear Hot/Abort: THSC2PC.EXE - Application Error Window appears, contains the text: *"The instruction at '0x7710510e' referenced memory at '0x00000048.' The memory could not be 'written'.* *Click on OK to terminate the application."*
9	At Application Error Window, click OK
10	Clear Hot/Abort: THSC2PC.EXE - Application Error Window appears, contains the text: *"The instruction at '0x00000000' referenced memory at '0x00000000.' The memory could not be 'written'.* *Click on OK to terminate the application."*
11	At Application Error Window, click OK
12	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #5: Department of Defense Warning with Server Busy Window

Path 1 (4 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At Error LH41C, click OK
3	Department of Defense Warning appears
4	Server Busy Window appears
5	At Server Busy Window, click '*Switch-To…*' repeatedly
6	NOTE: Unable to clear Server Busy Window or Department of Defense Warning
7	Kill C2PC Application from Task Manager
8	Dr. Watson for Windows NT Window appears, displays the following text: *"An application error has occurred and an application error log is being generated.* *THSC2PC.exe* *Exception: access violation (0xc000005), Address: 0x00000000"*
9	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #6: Duplicate Mission Name

Path 1 (2 Repetitions on RHC #2)	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode Warning, click OK
4	At the Select Networks dialog, click Next, then Finish
5	From the Mission menu, select New Mission
6	Enter a duplicate name in the MISSION NAME field
7	Click the X in the upper right corner of the Mission dialog
8	At this point, the "*THSCore: <MISSION NAME> is a duplicate mission name. Please choose another.*" dialog box appears. Clicking OK brings up the same dialog box again. This loops until the application stops responding.
9	Close the program in Task Manager.
10	Reboot the RHC to clear the dialog box that persists on the desktop.

Path 2 (2 Repetitions on RHC #2)	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode Warning, click OK
4	At the Select Networks dialog, click Next, then Finish
5	From the Mission menu, select Open Mission
6	Select a mission with a duplicate name from the list.
7	In the Mission dialog, click the X in the upper right corner of the window.
8	At this point, the "*THSCore: <MISSION NAME> is a duplicate mission name. Please choose another.*" dialog box appears. Clicking OK brings up the same dialog box again. This loops until the application stops responding.
9	Close the program in Task Manager.
10	Reboot the RHC to clear the dialog box that persists on the desktop.

Path 3 (5 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At LH41C Error, click OK
3	At the Department of Defense Warning, click OK
4	At the Simulation Mode Warning, click OK
5	At the Select Networks dialog, click Next, then Finish
6	From the Mission menu, select New Mission
7	Enter a duplicate name in the MISSION NAME field
8	Click the X in the upper right corner of the Mission dialog
9	At this point, the "*THSCore: <MISSION NAME> is a duplicate mission name. Please choose another.*" dialog box appears. Clicking OK brings up the same dialog box again. This loops until the application stops responding.
10	Close the program in Task Manager.
11	Reboot the RHC to clear the dialog box that persists on the desktop.

Fatal Error #7: Initial Point Creation

Path 1 (1 Repetition on RHC #2)	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode Warning, click OK
4	At the Select Networks dialog, click Next, then Finish
5	From the Mission menu, select New
6	Click the Target button, then select a target and click Close
7	In the Mission dialog, click Activate
8	Close the Mission dialog
9	On the map, create a Control Point
10	On the map, create an Initial Point
11	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #8: Opening Log File

Path 1 (2 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At LHC41C Error, click OK
3	Select '*Audit File Maintenance*' from THS Tools menu
4	Select log from '*3/17/2004*', click View
5	NOTE: At this point THS(X) becomes unresponsive, THS(X) and sqlserver are contending for CPU time, and THS(X) has used over 62 MB of memory.
6	Kill THS(X) process from Task Manger
7	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #9: Creating a Quick Fire Plan

Path 1 (1 Repetition on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At the LH41C Error, click OK
3	At the Department of Defense Warning, click OK
4	At the Simulation Mode Warning, click OK
5	At the Select Network Dialog, click Next, then Finish
6	At the map, create a new Target
7	In the Symbology menu, select Fire Planning, select Quick Fire Plan
8	Click the New Plan button to create a new fire plan.
9	NOTE: At this point the RHC locks up and stops responding.

Fatal Error #10: Adding Target to Duplicate Fire Plan After Removing a Target

Path 1 (2 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At LH41C Error, click OK
3	At the Department of Defense Warning, click OK
4	At the Simulation Mode Warning, click OK
5	At the Select Networks dialog, click Next, then Finish
6	Create a fireplan named 'FIREPLAN1'
7	Create another fireplan named 'FIREPLAN1'
8	Add a target to the first fireplan
9	NOTE: the target has been added to both fireplans
10	Remove the target from the second fireplan
11	Add a target to the second fireplan
12	NOTE: At this point the RHC locks up and becomes unresponsive.

Fatal Error #11: Closing THS(X)

Path 1 (2 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At LH41C Error, click OK
3	At the Department of Defense Warning, click OK
4	At the Simulation Mode Warning, click OK
5	At the Select Networks dialog, click Next, then Finish
6	Close THS(X)
7	NOTE: At this point THS(X) crashes, and the RHC becomes unresponsive.

Summary of Non-Fatal Errors

Non-Fatal Error #1: Zooming Anomalies (occurred on RHC #1 and RHC #2)

Occasionally while zooming in or out, the view would be re-centered over Asia. This error was reproducible.

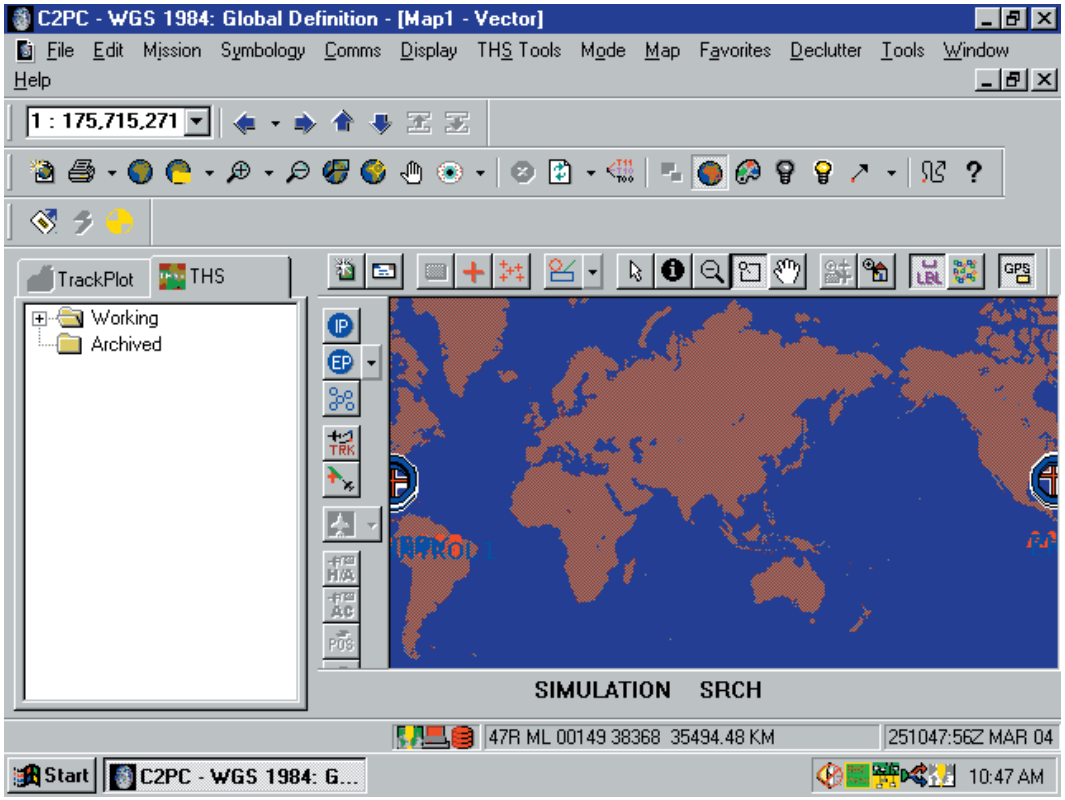


Figure A-9: *Zooming Anomalies*

Non-Fatal Error #2: New Address Book Entries (occurred on RHC #2)

While in the Networks menu, THS(X) was unable to draw the window for entering new addresses into the Address Book. Whenever the 'New' button was clicked, the screen briefly flashed (as if an attempt to draw has been made). Many features of THS(X), such as flight creation in CAS mode, were unavailable on RHC #2 due to this issue.

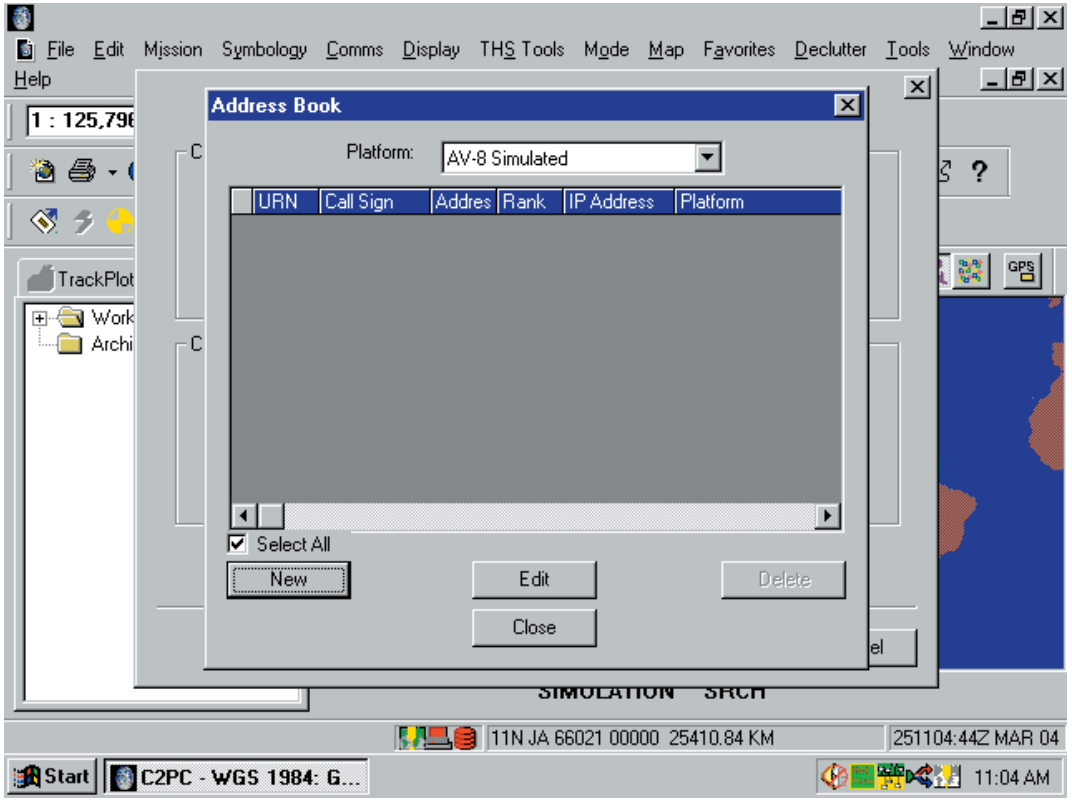


Figure A-10: *New Address Book Entries*

Non-Fatal Error #3: Message Sending (occurred on RHC #1 and RHC #2)

When a new Quick Fire Plan was created and sent, via the Fire Planning tool, a dialog box displaying the text "*Message Has Been Sent. Booyah"* was drawn to the screen. This dialog box was displayed regardless of whether the message was actually sent. This finding is confirmed in the source code in the file 'THSCoreQuickFirePlanPage.cpp'.

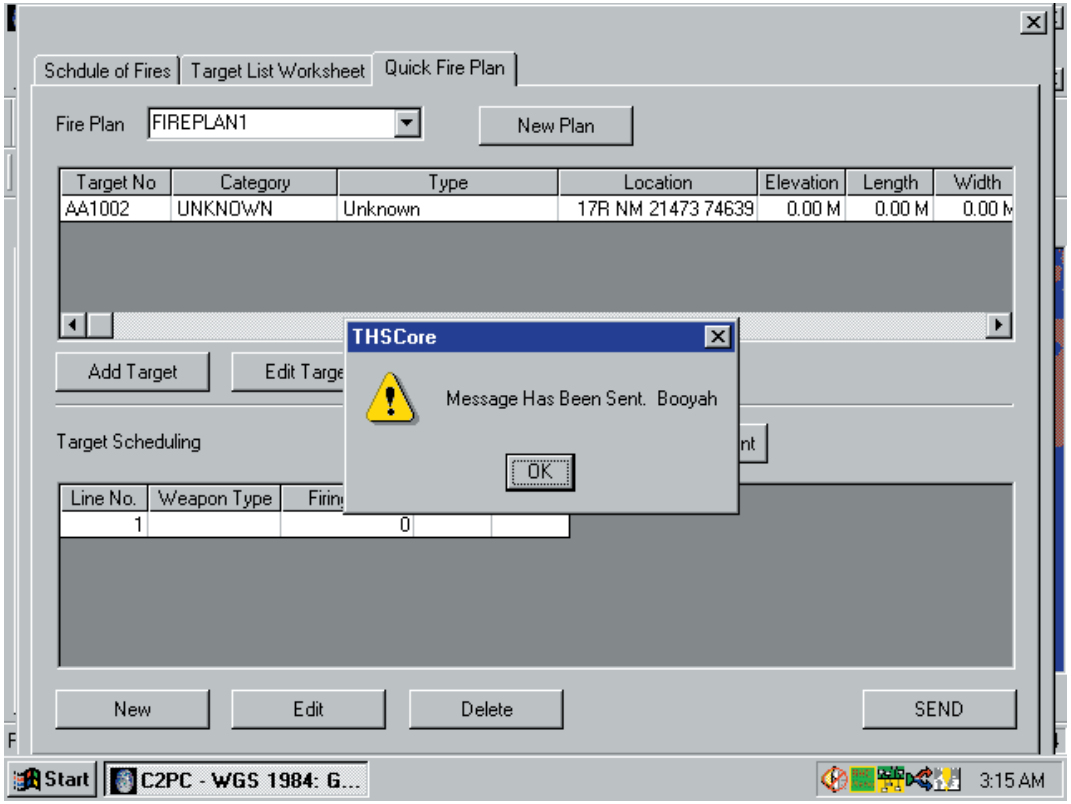


Figure A-11*: Message Sending Dialog Box*

Non-Fatal Error #4: Ebiosd.sys (occurred on RHC #1)

When attempting to load a mission, a 'Duplicate Mission Name' error occurred (see Fatal-Error #6), and a hard-reset of the RHC was required. After this reset took place, the RHC issues the following warning upon every boot.

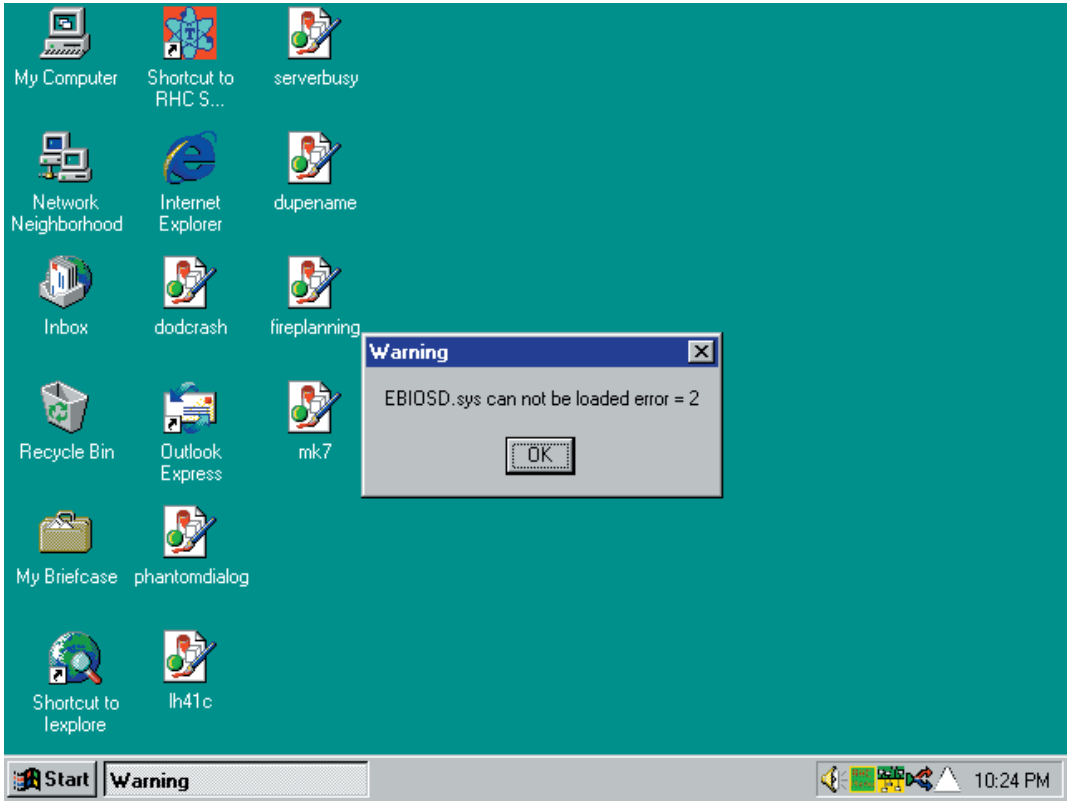


Figure A-12: *Ebiosd.sys Warning*

Non-Fatal Error #4: StauderLH41C (occurred on RCH #1)

When starting THS(X), the following error dialog is displayed. This error occurred every time THS(X) was started, with the exception of the runs in which the Mk7Interface error occurred (See Fatal-Error #3).

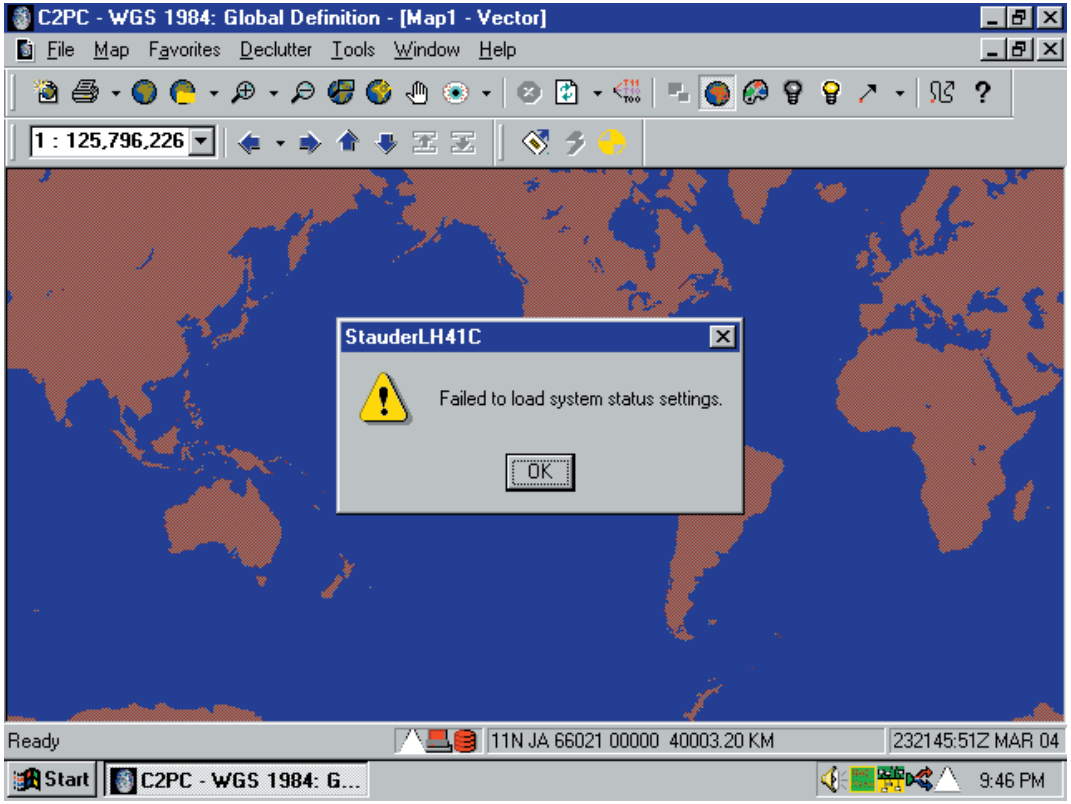


Figure A-13: *Stauder LH41C*

Paths to Successful Termination

The following is the list of paths that led to a successful run of the THS(X) software. *Success is not defined by completing an actual mission, but rather by being able to start and exit the THS(X) software.* Also listed is the number of instances any given run was completed, and the machine on which the successful run occurred.

Success #1: Started with Server Busy and LH41C errors, and exited normally

Path 1 (2 Repetitions on RHC #1)	
Step	Procedure
1	Start THS(X)
2	At LH41C error, click OK
3	At Department of Defense Warning, click OK
4	At Server-Busy Window, click '*Switch-to…*' at least 5 times
5	At Simulation-Mode Notification, click OK
6	At Select Network, select '*F-16 Simulated*' for Channel 1 Platform, select '*F-16 Simulated*' for Channel 2 Platform, click Next
7	At Server-Busy Window, click '*Switch-to…*'
8	NOTE: At this point THS(X) should be running
9	Exit THS(X)

Success #2: Started with LH41C error and exited normally

Path 1 (6 Repetitions on RHC #1):	
Step	Procedure
1	Start THS(X)
2	At LH41C error, click OK
3	At Department of Defense Warning, click OK
4	At Select Network, select '*F-16 Simulated*' for Channel 1 Platform, select '*F-16 Simulated*' for Channel 2 Platform, click Next
5	NOTE: At this point THS(X) should be running
6	Exit THS(X)

Success #3: Started Normally and Exited Normally

Path 1 (3 Repetitions on RHC #2):	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode warning, click OK
4	At the Select Network Dialog, click Next, then Finish
Path 1 (3 Repetitions on RHC #2): *Continued*	
Step	Procedure
5	From the Mode menu, select CFF Mode
6	From the Comms , select Network
7	In the Platform field, select AFATDS Simulated
8	Click on Address Book
9	At the Address Book dialog, select New
10	NOTE: The new address book entry dialog does not appear
11	At the Address Book dialog, click close
12	Exit Application
	In the Platform field, select AFATDS Simulated

Path 2 (1 Repetition on RHC #2):	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode warning, click OK
4	At the Select Network Dialog, click Next, then Finish
5	From the Symbology menu, click Fire Plan
6	At the Fire Plan dialog, select Quick Fire Plan
7	At the Quick Fire Plan dialog, click New Plan

8	In the Enter Name field, enter FIREPLAN1, then click OK
9	At the Quick Fire Plan dialog, click New Plan
10	In the Enter Name field, enter FIREPLAN1, then click OK
11	NOTE: Two fire plans with the same name are allowed to be created
12	In the Fire Plan field, choose FIREPLAN1
13	Click the Add Target button, then choose a target and click OK
14	At the THSCore - Continue with duration 0? dialog, click OK
15	At the Quick Fire Plan dialog, click SEND
16	At the THSCore: '*Message has been sent. Booyah dialog*', click OK
17	Exit the Fire Plan dialog
18	At the map, select Create Initial Point
19	In the Name field, put a name, then select OK
20	At the map, select Create Egress Point: Egress Point 1
21	In the Name field, put a name, then select OK
22	At the map, select Create Control Point
23	In the Name field, put a name and click OK
24	From the Comms menu, select Flights
25	In the Flights dialog, select New
26	At the THS Core Message, select OK
25	At the Flights dialog, select Close
28	From the Comms menu, click Networks
29	In the Platform field, select F-16 Simulated
30	Click the Address Book button
Path 2 (1 Repetition on RHC #2) *Continued*:	
Step	Procedure
31	In the Address Book dialog, select New
32	NOTE: No dialog for creating a new address book entry appears
33	NOTE: This is the same no matter what platform type is chosen

34	In the Address Book dialog, click Close
35	In the Select Network dialog, click Next, then Finish
36	In the Comms menu, select Flights then New
37	At the "THSCore: You must activate one CAS platform before creating a flight" dialog, click OK
38	In the Flights dialog, click Close
39	From the Mission Menu, click New Mission
40	In the Mission Name field, fill in a name
41	Click the Target button, then choose a target in the Target field
42	Close the Target dialog
43	Click Activate to activate the mission
44	Click the Send button.
45	At the " THSCore: You must select a network first" dialog, click OK
46	Close the Mission dialog
47	Deselect the check box on the mission that was just created to deactivate it
48	Click the '*X*' in the upper right corner of C2PC to close the application.

Path 3 (4 Repetitions on RHC #2)	
Step	Procedure
1	Start THS(X)
2	At the Department of Defense Warning, click OK
3	At the Simulation Mode warning, click OK
4	At the Select Networks dialog, click Next, then Finish
5	Set a mission active by selecting the checkbox next to its name in the Working Missions list
6	In the Symbology Menu, click Control Points
7	Select a control point, then set it to be the Initial Point
8	Select another control point, then set it to be the Egress Point
9	Close the Control Point dialog
10	Deselect the active mission by deselecting the checkbox next to its name in the Working Missions list
11	Exit the program by clicking the X in the upper right corner of C2PC

Appendix E: The Microsoft Component Object Model

Structure of a COM/ATL Project

This section provides additional detail on use of the Microsoft Component Object Model (COM) and the COM Active Template Library (ATL) to create and register dynamically linked functionality, and specific examples of the use of these technologies in THS(X).

COM Interfaces

COM interfaces are based on inheritance. If an object supports a requested interface, it must inherit from that interface. To support multiple interfaces, the object may inherit from multiple superclasses. The superclasses provide a virtual definition of how the COM subclass should behave, and the subclass provides the implementation. In the simplest versions of COM, the superclass methods are purely virtual, and all implementation is done by the subclass. This provides the separation of interface and implementation which makes a COM server reliable and extensible. Below is an inheritance diagram which shows the relationship between a COM class, its interfaces, and a generic COM interface called IUnknown.

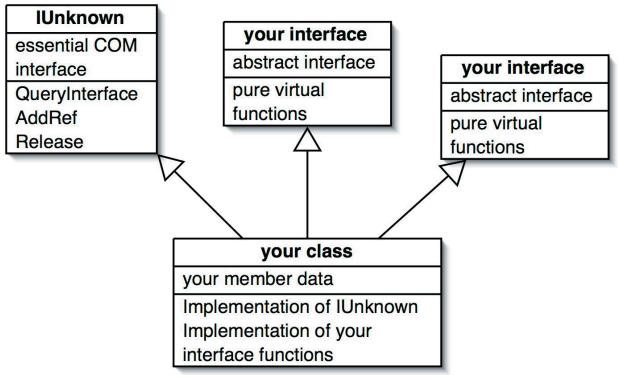


Figure 1*: Inheritance diagram for a COM object exposing required interfaces*

Because COM depends on inheritance, a straightforward implementation of COM imposes certain limits on the use of inheritance elsewhere in the design of the program, thus reducing the usefulness of inheritance as a tool in object oriented design. For example, interfaces must be abstract superclasses, the COM subclasses can not derive any amount of method implementation from them without violating COM's separation of interface and implementation. COM also limits interface inheritance. There are several ways to get around these limitations, but they are less straightforward. However, tools exist to simplify the use of COM, reducing the amount of work needed to use COM without limiting overall program design. One such tool is ATL (the Active Template Library).

THS(X) uses an ATL based implementation of COM. ATL uses much of the basic COM model, but also adds its own functionality. For example, it provides a template class to implement the methods of COM for a class, without imposing the above limitations on that class's inheritance hierarchy. Below is an inheritance diagram of a simple implementation of an ATL COM class, adapted from [Rector 99].

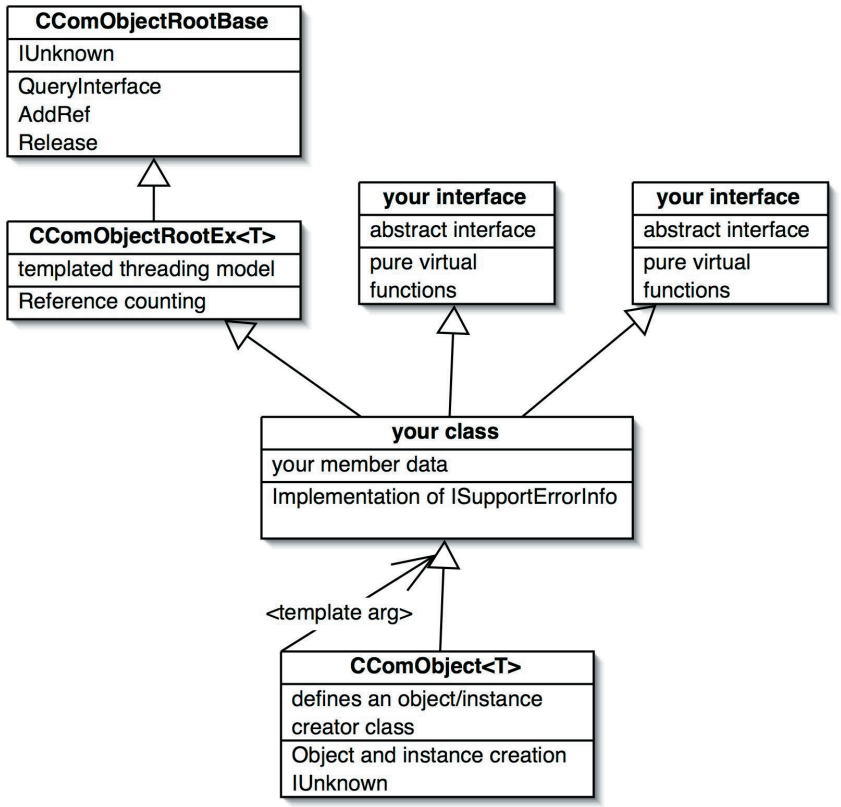


Figure 2: *Inheritance diagram for an ATL COM object exposing required interfaces*

ATL provides a significant number of options for using COM. There are many ways of using COM functions to access a server object, and many ATL definitions, methods, and classes involved in calling those COM functions. Some settings rely on ATL macros, others on what templates are used, many can be, and frequently are, defined in one file and affect objects in other files. Determining the type of threading model being used in THS(X), for example, required a search for files containing one of three possible ATL macros. ATL adds many options for COM, but in the process adds a significant amount of complexity. For example, below is a partial inheritance diagram for an implementation of COM in THS(X).

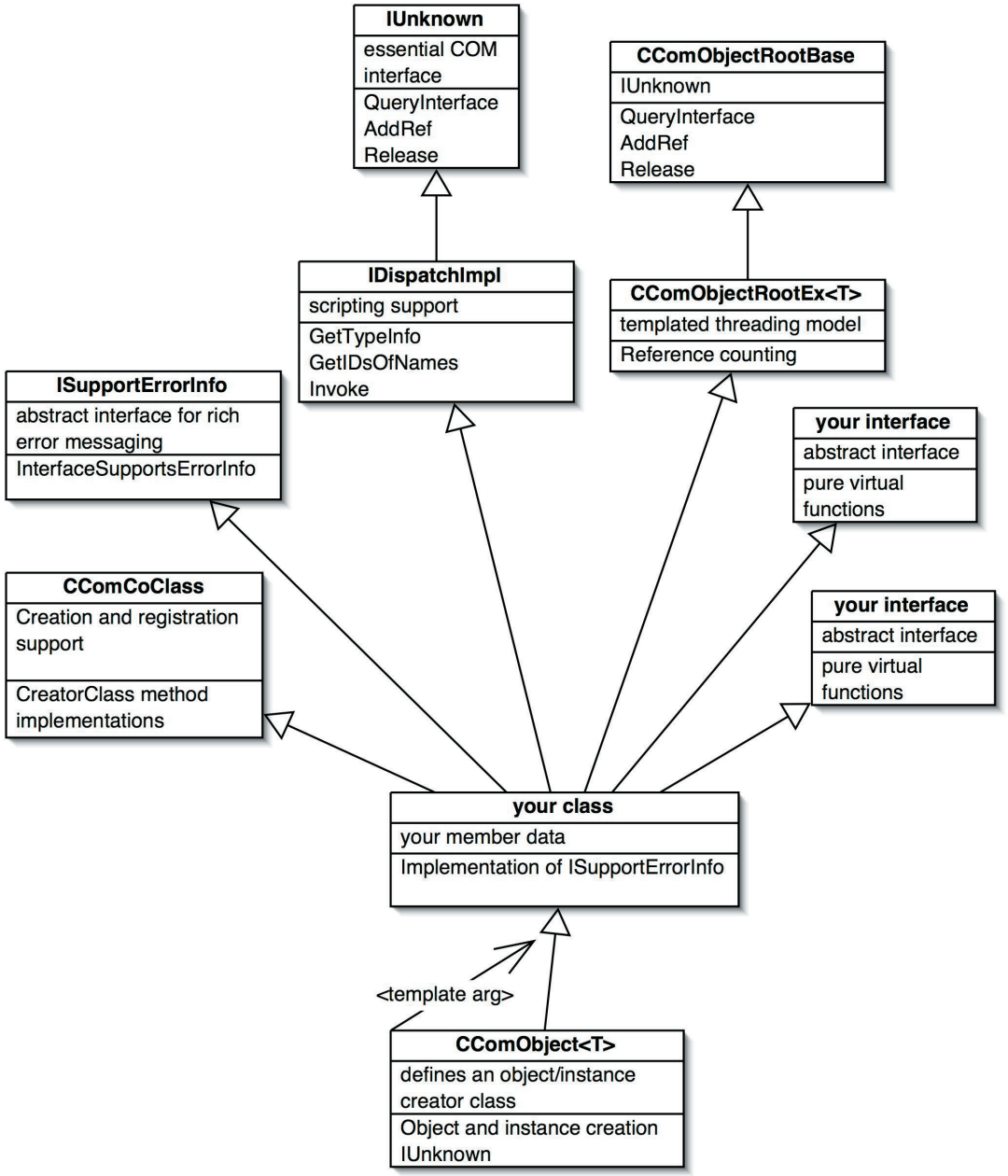


Figure 3: *Inheritance diagram for a THS COM object*

THS(X) COM Interface Example

The CAFAPDF169LineBriefMsg class inherits from 4 COM/ATL classes, and one THS class, CTMessageType. Some of the superclasses implement the basic methods of ATL COM, while others provide additional functionality. One of these classes, IDispatchImpl, is a templated class which supports scripting. IDispatchImpl is used as a base class for both CAFAPDF169LineBriefMsg and CTMessageType. There are a few major concerns with the use of this superclass.

First, it is not clear why scripting support is needed for a type of message. This class should be internal to the project, and doesn't need to be accessed by outside methods. This contributes to the overhead of COM, increasing the size of the DLL for a function which should never be used, and allows internal data to be even more accessible.

Second, the IDispatchImpl interface does not need to be included in the superclass. Every class which derives from CTMessageType will support a scripting interface, whether or not it is necessary. The type of scripting interface used is not an attribute of a message type, as is proven by the fact that CAFAPDF169LineBriefMsg needs to override the CTMessageType implementation with its own version of the class.

Finally, an object which derives from 2 different versions of IDispatchImpl is implementing 2 different versions of the same scripting interface. Choosing between these interfaces can be done by QueryInterface, but if a scripting client can use QueryInterface it does not need the IDispatch methods. Therefore, there is no good reason to support multiple IDispatch implementations [Rector 99].

The following section from the THS(X) SDD, pp. 254-255, may explain the use of IDispatchImpl in this manner. Note: a coclass is another term for a COM class.

"Typically a coclass extends the IDispatchImpl ATL template for every interface supported, which pulls in the declarations for the methods of each interface. The implementation for each is then added to the class. These coclasses, however, have been set up in a slightly different manner. A template class has been created for the IMessage interface implemented by each message. This template extends the IDispatchImpl for the IMessage interface. The coclass, then, extends this template instead of IDispatchImpl in order to support the IMessage interface.

The end result of this is the same. However, this now allows the implementation for the IMessage interface methods to reside in the template class rather than the actual coclass. Now other message coclasses can extend the template, and they will automatically inherit the interface AND its implementation."

There are several points to address with this reasoning:

- ➥ The IDispatchImpl class is just another superclass. It can not "pull in the declarations for the methods of" other interfaces.
- ➥ A COM class extends all of its interfaces separately; this allows a decoupling of interfaces which is an important concept of COM.
- ➥ IDispatchImpl supports scripting, which is not a requirement of all COM objects. Thus, it is not as typical as the document implies.
- ➥ If scripting must be supported, and the intention is for the IMessage interface to provide that support, there is absolutely no reason to duplicate the functionality of IDispatchImpl by inheriting from it twice.
- ➥ The coclass can extend the methods of the IMessage class separately from IDispatchImpl.

Overall, the use of IDispatchImpl seems confused and unnecessary, and the explanation provided does not clearly correspond to the actual purpose of this particular ATL class.

Another interface which the CAFAPDF169LineBriefMsg class inherits from is ISupportErrorInfo, a useful interface for all classes which support scripting. CTMessageType does not inherit from this class, thus separating scripting support from the error message interface which supports that scripting. ISupportErrorInfo is just an interface, and the implementation in this case is supplied by the CAFAPDF169LineBriefMsg subclass. This is a very standard implementation, creating a great deal of repetitive code in the project. If the class had inherited from the ATL class ISupportErrorInfoImpl instead, that class would have provided a default implementation. This sort of inherited implementation is used frequently throughout the code, for example in the above mentioned use of IDispatchImpl, so it is not clear why it is not used here as well.

Registration

As already mentioned in the analysis relating to dynamic linking, dynamic linking with COM involves the Windows Registry, thus creating problems of portability and reliability. Dealing with the Registry means dealing with the details of how the operating system will find the components of the program. This creates a relation with the specific operating system which reduces portability. The Registry is used to find files for dynamically linking, so it must know the location of those files. The registration scripts for ATL/COM register some of their data based on the current location of the program executing the scripts, since they should all be located together. This means that the correct COM data for a dynamically linked file will be in the registry at the time that the file is loaded. It still leaves the problem of loading the correct file to begin with, and not overwriting Registry entries. It also allows for the possibility of an error in the registration scripts. The registration process is not simple, and there is a lot of room for error. Even if much of the work is accomplished by an ATL Wizard tool supplied by the compiler, as is the case in THS(X), the developers must still be aware of the details of the automated implementation in order to extend it.

To register a COM class using ATL, the class can implement a registration function itself or derive one from a templated ATL superclass which implements the function for it. It also must define a registry resource name or integer ID. An ATL macro uses this information to call a function which runs a script for that resource. Each class can have a different script file, as can the server as a whole. The script file specifies what registry changes must be made for a given class. These include the universally unique ID for that class, a name, a program ID, threading information, and other details. The name of the registry script for that class must be included in a project resource file. The server registry script must be run before the class registry scripts, to register any categories the classes belong to. A method in the top level file of the library calls an ATL function which runs all of the registry scripts.

In THS(X), the top level registry function does not explicitly call a server level registration. This may be due to a documented bug in the ATL wizard which automatically sets up many details of COM implementation. Because of this, entries in the class registration files, specifically an application ID, may silently fail at registration time. The class will still be registered, and will work fine if the application ID is not required. Since an in-process server may not always require an application ID, this may not be a fatal error. But this sort of bug demonstrates the complexity of using the registry in this manner.

Using COM

This section deals with the use of COM and the ATL to use a dynamically linked class, once that class has been structured to support COM as described above. Outlined below are the steps necessary to create and use a COM object, and some of the functions which can be used to implement each of these steps. Examples are then given of the use of COM in the THX project, along with comments on the effectiveness of particular methods of using COM and their overall results on a project.

Before a COM object can used, several steps are necessary to load the object and establish an interface. The COM library must be loaded by each thread that will make use of any COM objects, the binary which contains those objects must be loaded and registered, an object of the desired type must be created, as well as an instance of that object, and the correct interface to access the desired member functions must be obtained. There are a number of different functions which can be used to do all of this. The ones described below are common implementations basic to COM, while many of the examples given later will involve additional layers of ATL or other methods and constructs which are meant to simplify the use of these functions. Because there are so many ways of using ATL, only the usage in the examples will be discussed.

Loading COM

CoInitialize / CoInitializeEx

All use of COM must begin with a call to one of these functions. CoInitialize loads the COM library in the current thread, and creates a new apartment for that thread. The client can choose to enter a single or multithreaded apartment.

COM Apartments and Threading:

A COM apartment is an abstraction which defines groups of objects based on their threading requirements [Box 98]. All multi threaded objects can reside in the same apartment and execute concurrently. A single threaded object resides in its own apartment, and COM protects it from concurrent access by multiple threads. While the client must choose a single or multi threaded apartment, the COM objects can choose to be single or multithreaded, or either, or apartment threaded – the case where instances are single threaded, but shared (global and static) data can be accessed by multiple threads.

Proxies and Method Remoting:
If the desired object is in a different apartment, its interface must be imported. A proxy can be accessed by the importing apartment, and passes control to the object's apartment. COM marshals the interface from the object's apartment and unmarshals a proxy in the client's apartment. (See method calls).

Creating an Instance of a Com Object

To work with a COM object, first you must obtain a pointer to a desired interface of the object, based on universally unique CLSID (Class ID) and IID (Interface ID).

Object vs. Instance:
In COM, a class object can contain/implement instance-independent data (like a static class member), and data to create and manage instances. Like an instance, a class object has an interface, and it can expose the object-specific data described above, or a generic interface (such as IClassFactory) to produce instances of itself.

CoGetClassObject

If the threading requirements are the same, in process activation can put the DLL information in the same apartment. Otherwise, the pointers returned here are pointers to proxies.

The Steps of an Activation Request:

- ➥ COM Service Control Manager loads COM server
  - ■ Check if file is already loaded.
  - ■ Check Registry for class configuration information (find the file information corresponding to the requested CLSID)
    - ⊙ If the file is not available, and the program is running over a network, ask the Class Store to make it available and update the registry.
  - ■ For in-process requests, load the DLL file into the client's address space.
  - ■ Get address (GetProcAddress) of the DLL function DllGetClassObject.

- The COM server uses the DLL file's DllGetClassObject method to get a pointer to the required class object and interface.
  - The first time DllGetClassObject is called, it initializes (static) objects for all accessible classes, if the server is local but not in-process. If it is in-process, it only initializes classes as requested. It caches the IUnknown interface for each object.
  - Look up object in Object Map by Class ID.
  - Use QueryInterface to get a pointer to the requested object of the type requested by the interface.
    - QueryInterface will increment the server lock count.
    - The interface type can be one of the exposed interfaces specific to the object, or it can be IID_IClassFactory.

The specifics of how CoGetClassObject calls DllGetClassObject aren't completely clear. According to DllGetClassObject documentation, "When an object is defined in a DLL, CoGetClassObject calls the CoLoadLibrary function to load the DLL, which, in turn, calls DllGetClassObject." [URL **1**]. But according to documentation on the CoGetClassObject function [URL **2**], and the CoLoadLibrary function [URL **3**], "The CoGetClassObject function does not call CoLoadLibrary." Finally, according to [URL **4**], the CoGetClassObject function calls DllGetClassObject directly.

IClassFactory::CreateInstance

The object created by CoGetClassObject cannot be used for instance-specific functions. To get a specific instance of a class from that object, request the IClassFactory interface when calling CoGetClassObject, and use that to create instances. CreateInstance is just a method call using the object interface you just received. It doesn't go through the SCM (Service Control Manager), since it already has an object pointer, so it can be faster than CoCreateInstanceEx (described below) for repeatedly creating new instances.

- Creates new class instance (new).
- Calls QueryInterface for the requested interface to that class.

CoCreateInstanceEx

This function creates a single instance of a COM object, without first creating the object as done in CoGetClassObject. From the server's perspective, this works exactly the same as a call to GetClassObject, followed by a single call to create an instance, and a call to release the original object pointer.

.

CoCreateInstanceEx can also obtain multiple interface pointers to the same instance in one call. Return values include an array of HRESULTs corresponding to the success of each interface query.

The following diagram was taken from [URL **5**]

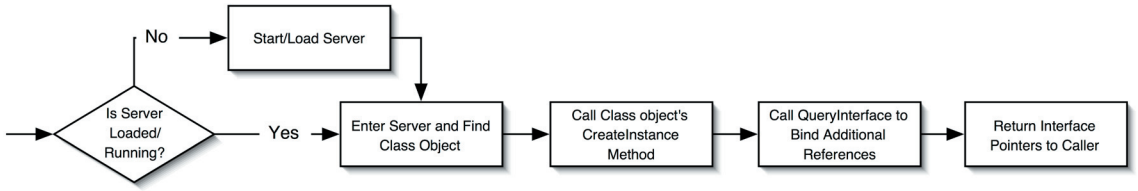


Figure 4: *COCreateInstanceEx*

According to COM documentation at the above link [URL **5**], CoCreateInstanceEx doesn't actually call GetClassObject; rather, it executes calls to create a class instance and repeatedly calls QueryInterface from inside the process of the class object (an optimization). However, books on ATL COM and some more recent documentation suggest otherwise: "The CoCreateInstanceEx helper function encapsulates three calls: first, to CoGetClassObject to connect to the class object associated with the specified CLSID, specifying the machine location of the class; second, to IClassFactory::Create Instance to create an uninitialized instance, and finally, to IClassFactory::Release, to release the class object." [URL **6**]

CoCreateInstance

This method is used when only one interface is needed for the instance. It only works on a local system. Otherwise, it is exactly the same as CoCreateInstanceEx.

AddRef

This function, which performs reference counting for objects, instances, and interfaces, is called automatically by the above COM functions for every pointer they return. In the case of non-heap-based objects, it does nothing.

Getting a Pointer to a Specific Interface

All use of COM Objects must be through exposed interfaces, since the point of COM is to separate the interface from the implementation. The interfaces in C++ COM are typically the base classes from which the COM class is derived. If a base class contains pure virtual functions, the implementation of those functions is found in the derived class, thus separating the (base) interface from the (derived) implementation. Once an interface pointer is obtained, it can be used as just another object in the code.

QueryInterface

QueryInterface obtains a vptr to a vtable (essentially a pointer to a table of function pointers), based on a universally unique IID (Interface ID). This interface pointer is put in a void** argument (thus it is not type safe), after it is set to the requested interface type.

Obtaining the correct pointer could be based on static typecasts to the correct base class, which is requested interface type. More

often, QueryInterface is implemented via a table containing IID's, a function to help find the right table entry, and corresponding vptrs or offsets.

For inheritance from multiple derived classes with common bases, QueryInterface must choose a specific derived class when querying for a base that is not unique. For example, all COM classes must implement the IUnknown interface (the interface which includes QueryInterface). A query for IUnknown in an object inheriting from multiple interfaces, all of which contain IUnknown, always returns the first of those interfaces (the beginning of the class's vtable) to represent the IUnknown functions it contains.

QueryInterface uses a reinterpret cast to IUnknown to call AddRef before returning the interface pointer.

AddRef (again)

For COM class instances, AddRef either counts references separately for each interface, or counts all references to one instance. Specific implementation is up to the object.

Method Calls

Standardized method call conventions and return type are used to allow cross-compiler compatibility.

CoMarshalInterface and CoUnmarshalInterface handle method calls between apartments. CoMarshalInterface turns an interface pointer into a serialized representation that can be passed between apartments. It writes this data to a caller-supplied byte stream through an IStream interface. CoUnmarshalInterface then returns an equivalent pointer to the original object or a proxy, which can be legally accessed in the apartment that called CoUnmarshalInterface.

Destruction

Release

Release automatically calls delete on an instance when its reference count is 0. ATL uses smart pointers to keep memory clean even when there are unhandled exceptions. Release does nothing for non-heap based objects.

DllCanUnloadNow

DllCanUnloadNow is used by CoFreeUnusedLibraries to clear unneeded libraries. Unloading is based on reference counts and server locks (set by IClassFactory::LockServer). For multithreading, if a DLL is accessed by multiple threads, COM adds the DLL to list for freeing and calls CoFreeUnusedLibraries a second time for actual removal, after a delay. During that delay, if the DLL is accessed again, it is taken off of the removal list.

COM Examples

Apartments and Overall Threading Issues

Since I was given two versions of some of the same files, in two folders, I will look at the potential issues with threading in COM using both. This should illustrate more of the possible problems and the amount of effort that was required to avoid them.

Version 1: DigitalComms

All of the COM objects in THS(X) are apartment threaded, which means that COM will protect each instance of an object from access by multiple threads outside its own apartment. Global and static data, however, must be made thread safe by the programmer. All COM classes inherit from CComObjectRootEx <CComSingleThreadModel>, which means that reference counting is implemented without thread-safe locking functions. This is acceptable if access to any given COM object is only from a single thread, which should be true, since all objects are apartment threaded.

However, multiple threads can exist in THS(X). If a COM object in an existing single threaded apartment creates a new thread, COM does not create an apartment for it, it must call CoInitialize before using COM or accessing a COM object. These threads are no longer in the same apartment, but they can access information about the COM objects in other apartments through marshalling.

For example, in the AFAPD Simulation Network, new threads are created to send messages (CAFAPDSimulationNetwork::SendMsg), and passed information about COM objects. The new threads enter their own COM apartment in the construction of the SimWorkerThread, and then, in ThreadMsg.cpp, use QueryInterface on the COM objects they had been passed. COM is completely responsible for marshaling the calls for thread safety. This adds some overhead in the message sending process, but does contribute to thread safety.

While the above code appears to be thread safe, (assuming COM's marshalling is successful), the current design requires the developer to be that much more careful of threading issues. The other single threaded COM object which creates multiple threads is CommCenter. In CommCenter.h, the following comment illustrates the complexities created by this approach:

```
/**
 * @todo The network and message collections should be protected from
 *    being accessed by multiple threads at the same time. This could
 *    be difficult because many time one method uses a collection, and
 *    then calls another method that also uses the collection. One option
 *    would be to wrap the collections in another class that would enter
 *    a critical section anytime the collection was accessed. Need to
 *    research... When an iterator is used, does it too need to be in the
 *    code that protects the collection from multiple access?
 */
```

COM is used here to deal with communication between all threads, but there is a significant amount of marshalling overhead involved. Alternatively, if the processes which were meant to be multi-threaded had been created in multi-threaded apartments, the use of COM would no longer affect threading issues. The trade off, however, is more overhead to make each class thread safe, which may result in slowing down the entire program.

Version 2: DigitalCommsNet

In the DIGITALCOMMSNET version of THS(X), the above complications were avoided by entering a multithreaded COM apartment with CoInitializeEx, but all COM objects were still apartment threaded. Therefore, CommCenter is created as a single threaded COM object, but the threads it creates become multithreaded COM clients.

The way this is used is that the object launches a new thread, which uses CoinitializeEx to enter a multi threaded COM apartment. CommCenter and other objects which should only be created once, such as CommsDB, are implemented as singletons. The same singleton can then be accessed from multiple threads without COM marshalling, as described in the following comments from CommCenter.h:

// Since this is a different thread than the one the database connection was established on,
// COM will not automatically marshall to it. Since CommsDB is a singleton, we can get around
// this problem by simply requesting a new CommsDB, which will get the interface of the
// singleton object. That interface can then be used in this thread.

Since these objects are still created without thread safety (specifically, they still inherit from CComObjectRootEx <CComSingleThr eadModel>), this is a bad idea. Singleton classes in single threaded apartments in COM must be thread safe, must not require thread affinity, and must be apartment neutral [Rector 99].

Initialization: DigitalComms\AFAPDF169LineBriefMsg

Since the class is itself a COM object, instantiated by another COM object, it is already in a single threaded apartment. Its member data consists of COM objects which are initialized in the final initialization function. Final initialization and destruction functions are used rather than simple constructor and destructor calls so that success values can be returned. Exceptions cannot be used on failure since ATL does not coexist well with the C Runtime Library (CRT), so the default setting for an ATL project is to not link with the CRT.

The final initialization of the AFAPDF169LineBriefMsg object uses a SMART_POINTER_ITEM_CCOMOBJECT macro. For this particular implementation, the macro, shown in the next line, gets a pointer to an instance of a CComObject<ObjectType>:

hReturnStatus = CComObject<ObjectType>::CreateInstance(&ObjectName);

This means the CComObject object of the appropriate type creates a new instance, and uses QueryInterface to get an interface to return. CComObject< ObjectType> implements IUnknown for the ObjectType, using the functions provided in CComObjectRootEx.

This is a call to a static member function of the class CComObject<ObjectType>. Although a specific instance of a CComObject isn't needed for a static member function, the function still must be loaded into memory before it is used here. Therefore, at an earlier point of initialization, GetClassObject must be called for the CComObject class.

The following is an outline of the steps taken by the ATL class CComObject, and its base classes, to create an instance of itself.

CComObject<T>::CreateInstance

- Uses an ATL smart pointer function CreateInstance.
  - Calls T::_CreatorClass::CreateInstance.
    - Evaluates to CComCreator2 < CComCreator < CComObjectCached < T > >, …> ::CreateInstance.
    - CComCoClass provides this implementation.

CreatorClass::CreateInstance

- Creates a new instance (new).
- Calls SetVoid().
  - Defined in CComClassFactory to cache the creator function if this was an object creation.
  - Else, defined in CComObjectRootBase to do nothing.
- Uses the CComObjectCached AddRef function.
- Calls the FinalConstruct method for the object._
  - Performs initializations
- Calls QueryInterface
  - Calls CComObjectRootBase::InternalQueryInterface
  - Gets the requested interface Pointer.
  - Increments the reference count based on CComObjectRootEx

This chain of function calls must be executed for every COM-based member variable. This is an exceptional amount of overhead for objects which are contained in the same library, and should not require any form of dynamic linking, using COM or any other technology. Nonetheless, once the AFAPDF169LineBriefMsg object has initialized its data through COM creation functions, that data is accessed through COM interfaces, as described in the next section.

Use of Member Objects

An example of something being done to one of the member objects of AFAPDF169LineBriefMsg:

spField is of type CComObject< CAFAPDBstrFld>.
spFieldClass is of type CAFAPDBstrFld.

```
spFieldClass * pFld = ( spFieldClass * ) spField##;
pFld->AddRef();
hReturnStatus = pFld->put_Value( newValue );
pFld->Release();
```

While a call to QueryInterface is frequently similar to a static cast to the appropriate base class, it is not always the same, depending on how the interfaces are put together (since COM can be implemented in other ways besides multiple inheritance), and whether or not the call resolves to a unique base class. In this case, casting will very likely work because CComObject inherits from its templated type. However, this use is dependent on the particular implementation of a COM class, which is outside of the control of the developers using that class.

Second, spFieldClass (CAFAPDBstrFld) is not an interface; it's a class, with its own set of interfaces. This means that a call to QueryInterface would still be appropriate even if c-style casting is an acceptable way to get an spFieldClass instance.

Use of Class (CAFAPDF169LineBriefMsg)

The following illustrates a very straightforward use of COM:

…

```
CComObject<CAFAPDF169LineBriefMsg>* pNewMsg = NULL;
hStatus = CComObject<CAFAPDF169LineBriefMsg>::CreateInstance(&pNewMsg);
if (SUCCEEDED(hStatus))
{
        pNewMsg->QueryInterface(ppMessage);
}
```

…

In another case, an already instantiated object is passed to this function as 'pmessage', and used as follows:

```
if (pMessage)
{
        pMessage->AddRef();

        // Type cast to the generic message template
        CTMessageType* pBaseMsg = (CTMessageType*)pMessage;
        pBaseMsg->AddRef();

        …
        (use pBaseMsg methods)
        …

        pBaseMsg->Release();
        pBaseMsg = NULL;

        pMessage->Release();
}
```

Once again, QueryInterface is not used, instead the object pointer is cast directly to the desired type. A possible reason for that is that CTMessageType is not a COM class – it inherits from one of the classes also used by the COM classes, IDispatchImpl, but it is not COM and it does not support IUnknown. Therefore, what is really going on is that an interface was cast to something else entirely, which is then used as an interface because it is a base class for the COM object. This may work for this implementation, but it gets around every effort of COM to separate interface from implementation and to make changes to the implementation safer.

Conclusions

COM was intended to allow different programs and libraries to communicate with each other without concern for compatibility and versioning issues. Since THS(X) was developed by one vendor as a complete product, minimal use of COM should be needed, to communicate with any external software. However, almost all of the classes considered (the AFAPD classes) within this project use COM to communicate with each other. There does not seem to be a clear reason for this use of COM internally, nor does the COM implementation used emphasize the separation of interface and implementation which is necessary for the above stated use of COM. This leaves open the possibility that COM was used as a default design structure, resulting in unnecessary overhead and confusion in a large project. Additionally, COM is not being properly implemented for what benefit it may offer.

Carnegie Mellon

NSH 2105, Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pa 15213

p: (412) 268-6556
f: (412) 268-5895