

# **Reverse Engineering Tools and Practices in Software Maintenance**

**Drumm, Paul M.**

## Table of Contents

1. Abstract.....	3
2. Introduction .....	4
3. Approaches to Reverse Engineering.....	5
a. Looking for the source code .....	5
b. Reverse Engineering using the software requirement specification .....	7
c. Reverse Engineering the Graphical User Interface –GUI- based application by inspection.....	8
d. Reverser engineering via UML.....	9
e. Binary Reverse Engineering (Reversing) .....	10
f. Tools used to reverse engineer the system.....	14
4. Precautions on which approach to take when Reversing .....	18
a. Considerations When Looking for the Source Code .....	18
b. Reverse Engineering using the software requirement specification .....	18
c. Reversing the Graphical User Interface –GUI- based application by inspection .....	19
d. Reverse engineering via UML.....	20
e. Binary Reverse Engineering (Reversing) .....	20
5. Alternative Organizational Based Approaches to Reverse Engineering.....	22
6. Conclusions .....	26
7. References .....	27
8. Appendices .....	27

## **1. Abstract**

Part of performing software maintenance is to understand the software system you are working on in order to facilitate some type of change. One method of understanding software systems is reverse engineering. The focus of the paper will be on reverse engineering an application starting from the simplistic, looking for the source code, to the more complex issues in disassembling. An additional approach using a framework to incorporate those tools is also discussed.

## 2. Introduction

Software systems need changes all the time. In order to change a system you first need to understand the existing system to see if the changes are feasible. The desired changes, be they perfective, preventive, adaptive, or corrective, have to be studied and approved before work can be started. Having current source code and its documentation, requirements documentation, design documentation, and a functional help system will allow you to decide if work should proceed. Sometimes you may have no choice and you must understand the system and change it regardless of the documentation you have. The most important part of any system is having the source code as the basis for documentation about the system. You may not always have that luxury and have to understand the system on its current merits which poses a major problem to the software engineer that has to deal with it. There are several approaches to reverse engineering a system when you do not have the source code. One of the best choices for dealing with software systems that interact with the user is to study the GUI, document it and its actions, and recode it. If it is a .Net application that has not been obfuscated then it possible to recreate the entire source code by disassembly. Neither one of those situations will always be the case so you have to rely on other tools to reverse engineer a system from scratch. The worst case system would be a service based system that has no interactions with the user and works behind the scenes. In those cases you may have to actually attach to the process running the software and actually debug or dissemble it. There are also code profilers that allow one to map out the interactions between functions and interfaces and give some overall level of understanding of the system you are dealing with. The appendix lists a wide variety of tools and resources for reverse engineering.

### 3. Approaches to Reverse Engineering

There are many approaches to solve the problem of Reverse Engineering without having the source code. This may also apply to porting the application to a new platform such as Basic to Java. The listing below should suffice for most cases but is not all approaches one could take.

The list of approaches is further expounded on in later sections.

- Looking for the source code location
- Reverse Engineering the requirements via the software requirement specification
- Reverse Engineering the GUI based application by inspection
- Reverse Engineering via UML
- Disassembling the existing code
- Tools used to reverse engineer the system

The goal of this approach is to “facilitate change by allowing a software system to be understood in terms of what it does, how it works and its architectural representation” [Takang].

#### 3.1. *Looking for the source code*

This may seem obvious to the software engineer but maybe not to a configuration manager or deployment engineer. There are many places to look for the source code after the code has been deployed that may not be apparent unless you have developed, used configuration management, compiled, and deployed software packages before.

If looking in the file system and one is using Microsoft's IDE, Visual Studio then the programming source code should be under the Visual Studio projects directory under your user profile. Similarly if you use Sun Java Enterprise for Windows, the source code should be located in our profile under the name of the project that was created. In Solaris 10 on Sun Studio 11 your source code is located under the source directories for the program.

Source control is also a good place to find the source code. You must have your configuration manager give you the location of where the source code is kept along with permissions to read or write. Microsoft's Visual Source Safe is usually kept in a secure location of the local network and a small program is needed to access it. You can do quick searches to find the files you need. A lot of shops use Clear Case for source control. In that case you have to install the program and have the configuration manager allow you to access and "see" the files. It is a little more difficult for accessing and finding the code you need.

The build area is a very good place to find the source code. Typically source code is taken out of source control to a specific location on the build or staging system and compiled. There is no specific location for this for any language. You need to search for the various types of file extensions below of which is a partial listing.

ASPX, CS, VB, JS, VBS, H, SLN, PROJ are all typical Microsoft based source code files. Similarly for java, java, jsp, html, properties, xml, and others are indicative of source code files for java based applications. Finally, CPP, C, H, are typical of standard C or C++ source code files.

On rare occasions some of the source code may actually be stored in the same location as the binaries. For example, the wwwroot subdirectory on a MS Windows box typically contains the web application, the source code may be included as part of that file. For jsp files, they may be located in an application directory under the home\user director.

Question your staff and even take attempts to contact ex-employees and consultants to see if they may remember where the source code may be. Offer former employees and consultants a monetary reward if necessary. There are also programs available that will search every file system on a system or network given certain file characteristics if you cannot locate the source code in any other way.

Once you have the source code you can then use a tool to graphically show what the code is doing. This will be talked about in other sections of the paper.

### **3.2. *Reverse Engineering using the software requirement specification***

Not the most expedient process but if you have the software engineering specification you may reverse engineer the application in general but not at a algorithm or module level. This is essentially forward engineering if you have to recode the application without the source code but it is essential that you have the software requirements specification (SRS) to do it from. The original SRS should have use cases and usage scenarios that give the software engineer a higher understanding of why certain aspects of the program exist and function the way they do. Having this fundamental of the program allows the software engineer to make solid decisions when making changes to the software system. One of the big challenges a software engineer faces when porting an existing application to a new platform is to understand why the system behaves the way it does. The SRS allows the software engineer

fill in the gaps from missing or incomplete design diagrams and documentation in order to make similar application in a completely different environment.

### **3.3. Reverse Engineering the Graphical User Interface –GUI- based application by inspection**

The GUI and its events provide the software engineer a high degree of insight into an existing system that is a graphical system. The GUI point and click interfaces of modern non-compiled environments, such as Java and .Net, allow one to map out exactly what the code does by acting on the controls and observing the reactions. The main thing is to focus on the control and what changes based on the events of the control. A control may change by just moving your pointer over it for a few seconds or it may change when you fly over it. Control events maybe hierarchical and have events that cascade upon one another. A short observation list is shown below.

Control	Event	Reaction
Drop Down List	Click	Drop Down List expands
	Click on Expanded List Item	List Item Selected
	Double Click List Item	List Item Selected and list recedes. Related data shows in text box window
	Ctrl and Click List Items	Multiple Items Selected

Building out a full list for every control will give the software engineer a full model of how the program reacts. When the time comes for maintenance the list can be looked over to see what controls may be affected on the list and any events or subsequent affected controls that

may be acted upon by needed changes or other maintenance tasks. Side effects of making changes can be more readily determined by searching a “control” library.

### **3.4. *Reverse engineering via UML***

UML has many different views you can create an application from [Fowler04].

UML use cases – This is the basis for finding out how the system is used by capturing the functional requirements.

UML activity diagram - This shows workflows, the work processes, or/and the logic surrounding your application associated with a given activity like placing an order or how an order is handled.

UML Sequence Diagrams – This typically is used to show object relations in a single use case such as “place order” where a user does a final order submission from their shopping cart.

UML deployment diagram – Deployment diagrams show the physical layer with the software services laying on top of it.

Execution Graphs – Execution graphs are modeling and analysis tools derived from your sequence diagrams.

UML Class Diagram – Class diagrams show how your classes are constructed and the relations between them. They usually will show private and public members along with methods and any overloads.

Package Diagrams – Packages are objects, classes and other constructs that have been chunked together to form higher level objects or “packages”.

After seeing the wide variety of UML diagrams one may wonder “why use anything else for documentation?” Unfortunately business analysts, project managers, deployment engineers and others are not versed in UML even though it is a standard “Language”. However, for the proficient software engineer, having any UML means that you have the ability to reverse engineer the part of the application you have the UML for. There are many CASE tools available, such as Visio, which will let you generate source code for an entire application if you have the proper UML class diagrams. Entire documentation packages can also be generated from various types of UML diagrams. For example, if you have UML use case diagrams you may be able to reconstruct the entire software requirements specification based on it alone. Similarly for the code base, if you have the UML class diagrams and it is a truly object oriented program then you can regenerate the entire code base [Tonella]. You can also make preliminary changes to the classes in the UML diagrams and look for ripple effects on the remaining classes.

### **3.5. *Binary Reverse Engineering (Reversing)***

Probably one of the most difficult things in software maintenance is to have to reverse engineer a software system for which the source code is unavailable and you have to derive the information from the binaries or other intermediate language source. Since there are many different programming languages there are many problem domains associated with each one.

To facilitate starting the process one should look at reversing at two different levels. The first level is a more high level view of the system and is referred to as system-level reversing [Eilam] in which you use high level interactions and operating system tools to gather information about the program for documenting. The second is code-level reversing in which one examines the code at a lower level both statically (not running) and dynamically (at runtime).

Tools - Tools are the key to this type of reversing. At the highest level are affects on the operating system such as file, registry, network communication, and other forms of messaging that the application must make with the outside world. Tools from sysinternals.com provide means to monitor and log file, registry, and network calls by a program. The operating system itself provides a wide variety of tools to monitor process use of resources. SAR is a tool the is prevalent on unix like systems that can be automated to gain a wide variety of information about processes. Microsoft Windows operating systems provide basic tools such as process thread and handle counts, disk i/o, paging and other metrics directly from a GUI in the OS and also by the provided perfmon tool. Various APIs are also provided to hook into the operating system and monitor exactly what a application process is doing.

Disassemblers (and to some extent assemblers) are basis for most code reversing and are processor and language specific.

Debuggers are similar to disassemblers in so much that they both look at the binaries.

Debuggers can look at both binary and high level language code execution. Debuggers have runtime capabilities to trace and record code execution and set breakpoint in the code in

order to evaluate and manipulate objects at those breakpoints. Documenting these objects and how the code traces through a certain event path is the key to getting a good reversing. The testers workbench presented in [Bass] provides something similar by graphing all the function call between components in order to understand the architecture by reconstruction.

Decompilers actually reverse the compiled binaries be they virtual machine binaries or fully compiled binaries. Decompilers are language and platform/processor dependent and take on many flavors. Decompilers try to reproduce the high level language from the low level binaries in a fashion that should make the generated code look like what the developer originally wrote.

Approaches to Reversing - There are two basic approaches to reversing, static and dynamic. The static method is analyzing and reverse engineering the code, usually the binaries, while the code is not running, also called Dead-Listing [Eilam]. The dynamic approach is debugging while the code is running.

The static analysis approach to reversing typically involves a disassembler. The disassembler takes the static binary code and tries to convert it to a human readable format usually in a specific high-level or low level language as it pertains to a specific platform or virtual machine. For the non virtual machine platforms a disassembler should take the machine code instruction in its platform specific instruction code format (say IA-32 for instance) which, for IA-32, consists of an opcode, and a couple operators [Eilam] and then convert into assembler format such as `add,esp,40h` which says to add the value of 40h to the value of the register esp. Once all the machine code is converted, you can then start grouping functions and calls to function in a call graph. The call graph gives you insight into program

logic and how it call out to other known functions like operating system APIs or other well known, and documented interfaces. Given that you know have the call graphs, the inputs or responses, and the outputs, you should be able to document the behavior of the application enough to perform, or prepare to perform the required software maintenance activities that forced you to reverse the application.

If the binary code is for a virtual machine and that code has not been obfuscated then you should be able to generate a complete code and documentation set based from the appropriate disassembler. Even if it was obfuscated it may still be possible to generate complete source code for the application but with different class and method names. Again, the software maintenance task that forced you to perform reverse engineering should be possible to accomplish at this point using a wide variety of tools –see the tools section for a description of the appropriate tool for virtual machines.

Debuggers – A good debugger, such as Microsoft’s WinDbg, should be able to disassemble besides doing debugging in addition to being able to do all the standard features a debugger should do such as setting breakpoints and evaluating variables during the debug session. A good debugger should also be able to give up information about the processes that are running and how the operating system is relating to those process calls. For example if I want to see the register instructions for a given thread and the memory resources it is processing then that data should be available. The type of data that is available to a debugger depends on if it is a user mode debugger or a kernel mode debugger. A software engineer would typically use a user mode debugger which is typically the IDE(integrated development environment) that they are using. Microsoft’s WinDbg is also a good example of a user mode debugger. A kernel-mode debugger, such as Microsoft’s WinDbg, gives you not only a

picture of what the whole system is doing but also allows you to drill down into a process. This will allow you to see what drivers are being called and lower level APIs may be getting called by the application of interest. One variation on kernel mode debugging is that to perform it on a virtual machine rather than that of a real host system. This way the speed of the debugging than it would be on a remote system or having the OS keep freezing on you while performing local kernel mode debugging.

Decompilers – Native code, as opposed to virtual machine code, decompilation has never been a straight forward undertaking. There are no real good tools for native code decompilation however a reference to Boomerang has shown that it was actual useful in several instances. Boomerang, an open source (sourceforge.net) program does not lay claims to be able fully recover a program but enables you to perform reverse engineering rather than producing the original code as laid out by the programmer. As for virtual machine code, there are many programs out there that will do a reasonably good job at converting the virtual machine code to the virtual machine language of your choice providing a full set of documentation and giving you the ability to reverse UML on the program once the source code is generated.

### **3.6. *Tools used to reverse engineer the system***

There are many tools you can use to reverse engineer a system. There are several categories of tools such as Documentation, CASE, Disassembling, Debugging, and Reversing. Some tools cross multiple categories.

3.6.1. Microsoft Word – Documentation - This is the most simplistic documentation tool you can use. Word can be programmatically manipulated through an extensive macro library to generate documentation for your software if you have the source code. If you do not have the source code then word functions as a normal text editor for you to manually enter documentation.

3.6.2. Microsoft Visio – Documentation, CASE - Visio has a wide composition of UML compliant objects. If you have the source code you can automatically document your application in UML through the reverse engineering option. The class documentation is sufficient so that if you ever lose your source code that Visio can generate the framework within your classes and methods hence the CASE capability. If you do not have the source code, Visio functions as a top notch UML modeling and diagramming tool allowing you to map out a variety of UML views. Visio provides Use Case, Static Structures (classes, interfaces, etc), Sequence, Statechart, Deployment, and several other views that cover most of the object types presented in Version 2.0 OMG UML Standard [Fowler].

3.6.3. IDAPro – Disassembler, Decompiler (in beta), Debugger, CASE, Reversing – IDAPro is an fully featured interactive disassemble capable of taking machine code or virtual machine code and creating human readable representations of it including complex call graphs displaying relations between all functions and subroutines. It works on most every platform and across a wide variety of compilers for those platforms. A variety of plug-ins are available for IDAPro such as a newly released decompiler that generates human readable C++ source code similar to what the developer may have intended. The IDAPro debugger allows you to step through the assembly level language, which was generated from the machine code, while interacting with the program or by just doing a static run through. The

eight main registers, plus a few others, are available for viewing throughout the debug session which allows you to analyze the behavior before, during, and after the various calls to subroutines and functions. This lets the software engineer analyze the capability to perform software maintenance by understanding just how the current program acts.

IDAPro is a must for any software engineer that has to deal with software systems where either in whole or in part the source code is missing and there is little documentation on the system [IDAPro].

3.6.4. ILDASM – Documentation, Disassembler, and Reverser – ILDASM allows you to see the actual Intermediate Language code that is generated inside the portable executable (PE) from a .Net compilation. .NET code and other virtual machine (VM) code stores metadata about the code in IL. This allows you to see all your variable names and functions by using ILDASM. ILDASM can generate complete a complete hard code listing which you can use as your documentation. ILDASM also can generate a IL based file. This file you can then make changes to using IL code syntax and then use ILASM to put the code back into a PE file. This allows a software engineer to perform maintenance on the code without having the source code to work on. ILDASM allows an software engineer to completely understand what the original .NET developer had in mind when they wrote the code and allows software maintenance to achieved at a higher level than most tools will let you accommodate [MSIL] [Lidin].

3.6.5. Lutz Roeder's Reflector for .NET – Documentation, Disassembler, Reverser - This will take a non-obfuscated .Net PE or other .Net assembly and generate C#, VB.net, or other .NET language source code just as the author wrote it. It is a good reverse engineering tool

that allows you to take the generated source code and save it to a file which you can then pull up in your Visual Studio environment to work on [Roeder].

### 3.6.6. WinDbg – Documentation, Decompiler, Debugger (user and kernel mode)

– WinDbg was discussed in the Debugger section of this paper. WinDbg was written by Microsoft for debugging Windows programs and the operating system itself. WinDbg allows a software engineer to see the performance and behavior of the actual code while it is running and also in static assembly code. A good tutorial on WinDbg can be found here <http://software.rkuster.com/>.

## **4. Precautions on which approach to take when Reversing**

Given all the choices that you have for reverse engineering how do you know what may be a good approach and what may or may not work when you need to create alternate views of the system? Each approach given in the prior section has strengths and weaknesses depending on how it could be applied and to what type of system.

### **4.1. *Considerations When Looking for the Source Code***

Taking the approach of finding the source code has a very high reward level and will save you resources in the long run if you do find it. There are a couple considerations to take into account.

- Consider how well the source code is of representing the current software system before you spend resources looking for it.
- Take into consideration the issue that the maintenance on the software system may be such that the source code would not help even if found.
- There will be a breakeven point in how much revenue can be relinquished before the resources expended finding the source code outweighs the benefits.

### **4.2. *Reverse Engineering using the software requirement specification***

The software requirement specification (SRS) can be used to abstract the intended use of the system at the time the SRS was written. The strength behind using the SRS is that you know what the system was required to do when it was signed off by the stakeholders. If the organization has incorporated change orders into the software application then you may also see how the specifications have changed overtime and the purpose for those

changes orders. The SRS can be used to understand why the system was designed the way it was at a high level. The drawback to using the SRS is that it is only a specification with some possible design documents attached to it. The SRS cannot show you how changes you make to system will affect other parts of the system. The SRS and change orders may not be kept up to date and lack detail and design so if you did rely on the SRS to abstract any system information it would prove erroneous and the proposed software maintenance would most likely fail.

#### ***4.3. Reversing the Graphical User Interface –GUI- based application by inspection***

A GUI based application, such as a web-based ecommerce shopping site, can provide the software maintenance engineer a wealth of system information they can abstract. The benefits of basing a new view off of the GUI are that it is an interactive black box environment. You can give the system an input and it provides a specific response and output to that stimuli. Reversing using the GUI may give you a mid-level design view that you may use to develop a good abstraction of the system with. However you are working with black-box functionality. You do not know what is happening behind the scenes. You may develop a design view only to see that it affects or calls several other functions that does not have visible output to the GUI. Porting the GUI to another platform using the existing GUI as the basis may result in key features being left out of the system. Using a GUI to reverse engineer a software system should be used in conjunction with other methods to give a more holistic view or abstraction of the system.

#### **4.4. *Reverse engineering via UML***

Reverse engineering via UML is a task in understanding how the components relate by using various views [Fowler]. The benefits of the UML abstraction of the system depend on the reason for the extraction. If you already have a deployment view then it may not make much sense to try and make a package view. You may be better off looking at sequence graph to further understand the program. UML is a standard method of modeling software representations and therein lays its strength as a standard. By promoting a standard method of software representation you are promoting other aspects of software maintenance. Even though it is a standard not everyone uses it or takes the time to sit down and layout their software system in UML and it ends up as a resource issue, as with other documentation, that organizations do not want to deal with. A good point for reversing with UML is that once you have fully modeled the software you can actually do a forward engineering effort and recover the data objects and their types. A drawback to this is that the actual implementation within a method will not be recovered and is more of a Black Box type recovery.

#### **4.5. *Binary Reverse Engineering (Reversing)***

Binary reversing strengths lie in its ability to show actual implementation features that is mostly invisible to other forms of reverse engineering. Although the actual output from disassembly or calls graphs may be hard for the novice to understand, for the seasoned professional it can provide a wealth of information about the software system prior to making any changes. Not only does the implementation information become available but since call graphs show actual call between various functions you have the ability to look

at the high level architectural design based on calls to groups of functions. The software maintenance engineer can create a high level design document for future changes. The biggest weakness to binary reversing lies in the fact that it consumes a lot of resources and you must have a good understanding of the tools in order to gain information about the compiled system. However, for virtual machine code that has not been obfuscated, it provides the quickest way to document the system out of any of the reversing methods presented since you now have the source code for an object oriented system [Tonella].

## 5. Alternative Organizational Based Approaches to Reverse Engineering

The various approaches presented so far are individual approaches taken on their own merits. They all have their own strengths and weaknesses and the organization must take note of this when approaching any software maintenance project where we need enhanced program understanding. What an organization needs is a framework to incorporate some of these approaches into in order to have a more focused approach for knowing when to use some of these approaches instead of have an agglomeration of some individual tools and approaches. Tilley [Tilley] provides one such approach as a Reverse-Engineering Framework. This framework is more concerned with classifying the attributes of reverse-engineering environments. This SEI sponsored work looks at Cognitive Model Support, Reverse Engineering Tasks, Canonical Activities, Quality and Other Miscellaneous Attributes.

### ***Cognitive Support Model***

What we need is an understanding of the program in order to provide maintenance. The cognitive model shows us how we may undertake to understand the program using the various approaches we have. The bottom up approach to program understanding is more of a functional nature that looks at the smaller pieces and then chunks them together to form an overall understanding of the system. You may use the binary reversing technique to look at the individual functions and calls and then start coalescing those parts until you end up with a top level understanding. The top down approach starts out at the top with the specifications or requirements and starts breaking down the program, maybe using the GUI for reference, as

a method of completing the program understanding at the lower levels. The last cognitive model is the opportunistic model that uses both the top down and bottom up. I would equate this to looking at the software specification and then going to binary reversing to complete the understanding or vice versa.

### ***Reverse Engineering Tasks***

We need to know how to apply our approaches. There are several tasks we need to perform that we can use some of our approaches on.

Program analysis tasks use various tools and approaches to look at abstracting information out of the system. Usually this involves having the source code and then forming some meaningful abstraction from it. If you do not have the source code then binary reversing may be the approach to use.

Plan recognition is another task we might use. Good software engineering has certain best practices design patterns we should look for. Understanding where these patterns are and how they fit them together can be used to further our program understanding by abstracting this data out of the system. Binary reversing may be able to show these plans in some cases.

Concept Assignment tasks us with looking at the source code, if we have it, and recognizing conceptually what is supposed to do. This may be from code fragments or whole classes within the program.

Redocumentation is key to every reverse engineering project. We can use UML or plain text to abstract the information from the system. We can also use the interfaces themselves (GUI or not) and redocument those features too.

Architecture recovery tasks us with structural redocumentation of the system. This may be done using UML in combination with GUI , Specification, and binary reversing approaches to give us an overall picture of the system.

### ***Canonical Activities***

Canonical activities refer to the artifacts that can be changed and molded during reverse engineering.

Data gathering activities refer to any data we can gather about the system. It may be from static or dynamic analysis, document scanning, or experience capture from end users. Once data is gathered it must be represented and retrieved properly. Use of filters on data retrieval techniques may be needed to filter some of the data especially in large projects.

Knowledge management activities refer to the task of taking existing knowledge and organizing, discovering, and evolving knowledge about the system in a coherent manner that facilitates program understanding.

Information exploration tasks are those tasks of exploring the data we have using a method of navigation through the system to look at the data, analyzing the data when we find it, and then present the data we found in a meaningful fashion.

### ***Quality and Other Miscellaneous Attributes***

Quality attributes by definition are non-functional attributes but they are architected into all systems. Quality attributes can be quantitatively defined to some extent and should be considering your particular reverse engineering environment.

Applicability – Your reverse engineering approaches should be valid across multiple application and implementation domains. Try to make it easy to tailor your reverse engineering processes and approaches to other domains.

Extensibility – The reverse engineering environment should be able to be extended using a variety of integration mechanisms, programmability, and automation in order to make program understandability capable of a wide breadth of programs.

Scalability – The reverse engineering environment should be able to apply to small systems just as well as it does to large systems.

Computing platform – If possible the reverse engineering environment should cross multiple hardware, software, and operating system platforms.

Cost – The reverse engineering environment should be cost effective. The approaches and tools you use should not exceed the benefits of program understanding.

## 6. Conclusions

The approaches used to accomplish reverse engineering vary in their complexity and their returns. Knowing the reverse engineering task at hand and the skill sets available will guide you to choose the right approach. The most simplistic, looking for the source code makes a short attempt to look in the logical places to recover the source code. Using the software requirements to document the system is also valid as it provides you the initial insight into building the program but that may be invalid if the specification is not kept up and only provides high level insight. Using the GUI as a guide to program understanding only applies to GUI systems but the drawback is that there may be underlying program constructs that escape your program understanding. Using UML as an approach is an exercise in redocumentation. It can provide the software maintenance engineer a variety of choices and when done may even provide a means to regenerate the source code minus some of the implementation detail in the methods. Reversing, rather binary reversing is a most complex approach that requires a skill set or training in order to be a valuable tool. Once it is used properly can be made to reconstitute the source code or to show the underpinning of the architecture and how it works.

Finally a reverse engineering framework perspective was summarized that showed various components of the environment. This environment can be used to decide which reverse engineering approach one may take at various points in the framework.

## 7. References

- 7.1. [Tilley] Tilley S *A Reverse-Engineering Environment Framework* ,  
[www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr005.pdf](http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr005.pdf)
- 7.2. [Tonella] Tonella P, Potrich A, *Reverse Engineering of Object code* Springer 2005
- 7.3. [Eilam] Eilam E, *Reversing: Secrets of Reverse Engineering* Wiley 2005
- 7.4. [Bass] Bass L, *Software Architecture in Practice* Addison-Wesley 2003
- 7.5. [Lidin] Lidin S, *Inside Microsoft .Net IL Assembler* MSPress 2002
- 7.6. [RevEng ] <http://www.acm.uiuc.edu/sigmil/RevEng/> Introduction to Reverse Engineering Software
- 7.7. [Takang] Grubb P, Takang A *Software Maintenance Concepts and Practice* World Scientific 2003
- 7.8. [MSIL] Microsoft *MSIL Disassembler* [http://msdn2.microsoft.com/en-us/library/f7dy01k1\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/f7dy01k1(vs.71).aspx)
- 7.9. [Fowler04] Fowler, M *UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language* Addison-Wesley, 2004
- 7.10. [IDAPro] IDAPro Disassembler website <http://www.hex-rays.com/idapro/>
- 7.11. [Roeder] Loetz Roeder Reflector website <http://www.aisto.com/roeder/dotnet/>

## 8. Appendices

Please refer to <http://www.laatuk.com/tools/> for a complete listing of tools.