

Reverse Software Engineering Using UML tools

Jalak Vora¹ Ravi Zala²

^{1,2} Department of Computer Science and Engineering, Nirma University

Abstract--- In this paper, we have taken the source code for social Networking sites and we are going to perform Reverse engineering. For this we are using UML Tools like Microsoft Visio and visual paradigm. Here, we are going to compare the outcome of these UML tools. This comparison will show how we can effectively produce meaningful and semantically accurate UML models using reverse engineering.

Keywords: Reverse-Engineering, UML tools, Microsoft Visio, Visual Paradigm.

I. INTRODUCTION

Many software engineers want a tool which can accurately obtain UML class models from source code. Unfortunately, the UML which are used do not perform the task efficiently. Study of various UML tools shows that they primarily focus on producing class diagrams but fails to represent the design details. This creates problem when we want to recover software models semantics required for high-level program comprehension. This paper gives the comparison of UML tools like Microsoft Visio and Visual Paradigm Tools.

II. MOTIVATION

“Reverse engineering”-it is the term which was originated from the analysis and decoding of hardware for commercial or military. The purpose is to deduce from the design decisions from the products with no given additional knowledge about how they are produced. These same techniques of reverse engineering are currently used in industrial or defence and to recover incorrect, incomplete, or unavailable documentation.

A common problem experienced by software developer people is of understanding legacy code. Legacy code is a semi-formal term which says about the coded programs which become difficult to understand when they grow in size and complexity. Much amount of effort is spent on maintaining existing software systems rather than time taken to develop new ones. Almost 50% to 80% time is taken for maintenance of software system. Also 47% of time is spent on improvement of programs and 62% on correction and for proper documentation. Without diminishing the importance of software engineering activities focusing on initial design and development, empirical evidence suggests that significant resources are devoted to reversing the effects of poorly designed or neglected software systems. In a perfect world, all software systems, past and present, would be developed and maintained with the benefit of well-structured software engineering guidelines. In the real world, many systems are not or have had their structured design negated, and there must be tools and methodologies to handle these cases.

- The ability to exchange and use information.
- **Lost documentation:** Reverse engineering is done mostly because particular device has been lost or the person who developed is not available.

- **Product analysis:** To determine how product works and too find its estimated cost.
- **Security auditing.**
- Acquiring sensitive data by disassembling and analysing the design and code of a software component.
- **Military or commercial surveillance:** Knowing about enemy’s latest research by stealing or capturing a design and disassembling it.
- Removal of copy protection, evasion of access restrictions.
- Creation of unlicensed/unapproved duplicates.
- Academic/learning purposes.
- **Competitive technical intelligence:** To know about your competitor is actually doing.
- **Learning:** To learn from the mistake and correct them.

III. THE STATE OF THE ART

A. Early Work in Reverse Engineering:

There are number of various CASE tools available to map given source code to structural models. Reverse Engineering is a well-established branch in which these CASE tools are used. These CASE tools were used alongside programming languages like COBOL for design documentation. Actually, the concept of Reverse engineering came from hardware where circuits were reverse-engineering to create clones. When this technique was adopted by software engineers there was dearth of well-defined terminology for both technical and market-place discussions.

B. Taxonomy:

Reverse Engineering is classified into two parts: *Redocumentation* and *Design Recovery*.

1) *Redocumentation* is defined as creation or modification of the semantically same type of representation abstract level. It provides the software developers an easier way to visualize the relationship among various software components.

2) *Design Recovery* is one of the part of reverse engineering where external information, domain knowledge and other deduction and reasoning are added to make more meaningful abstraction.

Reverse Engineering is one of the tool to reduce the complexity of the software system by building models be seen at various abstraction level and there could also be various multiple views of the given system. This can be seen analogous to the different views of blueprint of the building where a single view cannot describe the entire system architecture.

C. Difficulties in Reverse Engineering:

The difficulties in reverse engineering lies in knowing the level of the abstraction needed. Computer languages are formal while humans can have reasoning, hence the result of reverse engineering is very subjective in nature. The program is analysed in such a way so that reverse engineer

can build correct structural models from the low details available in the program.

The choice of the representation of the given problem and tools which are used will derive usefulness of the generated reverse engineering information. The work done on integrating both the approaches i.e. top-down and bottom-up to understand a given program to called the Synchronized Refinement which is described. This approach is based on the finding the design decisions in the available source code and the information Structure suitable for software maintainers.

However, the process suggested is labour intensive, though automating the individual tasks in the process can reduce the rigor involved.

Reverse engineering is a challenging task because it involves mapping between different worlds in five distinct areas:

1) *Application Domain <--> Programming Language*

A programming language is just a model environment to solve some real problem. While there are tools which helps us in understanding what code is doing from source code itself but it is very difficult to determine what is occurring in code from domain perspective. So, in these field there is little assist for the reverse engineers.

2) *Machines and Programs <--> Abstract, High-Level Design*

Simple, abstract concepts (“sort the list of customers by last name”) quickly become lost in the minute detail of programming.

3) *Original Coherent, Structured System <--> Actual System, With Structure Decaying*

Even when good documentation is available for a system, maintenance over time causes the structure to drift from the original specification. So, the reverse engineer must be able to solve any kind of unsynchronized methods in the current design and implementation of it.

4) *Hierarchical Programs <--> Cognitive Association*

Computer programs and formal, hierarchical expressions. Humans think in associative “chunks” of data. A reverse engineer must be able to “build up correct high level chunks from the low level details evident in the program”.

5) *Bottom-Up Code Analysis <--> Top-Down Application Analysis*

Code analysis is by its nature a bottom-up exercise. It requires, simultaneously, higher level meaning to be extracted from code fragments, and higher level concepts to be mapped to lower level implementations. To make this task even more difficult, the engineer must be able to handle obfuscations such as *interleaving*. Interleaving is the intentional (for optimization) or accidental (poor design or sloppy maintenance) colocation of logically separate tasks within the same spatial sequence.

Currently, reverse engineering is currently heavily dependent on human interaction and steering. While there are tools to assist the reverse engineer in program comprehension, it is not a fully automated process. The human element present in program comprehension is the subject of another field, *software psychology*, pioneered by Shneiderman. Software psychology measures

Human performance while interacting with computer and information systems.

IV. APPROACHES TO AUTOMATING REVERSE ENGINEERING:

A variety of approaches for automated assistance are available for the reverse engineer in program comprehension. Some of the more prominent approaches include:

A. *Textual, lexical and syntactic analysis* – this approach is based on the source code and its representations. These include the use of UNIX’s lex, lexical metrics (counting assignments, identifiers, etc.) and even automated parsing of the code searching for clichés .Clichés are standard approaches to problem solving that can extracted from the source code to give hints about design decisions. The unit of examination is the program source itself.

B. *Graphing methods* - there are a variety of graphing approaches for program understanding. These include, in increasing order of complexity and richness: graphing the control flow of the program, the data flow of the program, and program dependence graphs. The unit of examination is a graphical representation of the program source.

C. *Execution and testing* - there are a variety of methods for profiling, testing, and observing program behaviour, including actual execution and inspection walkthroughs. Dynamic testing and debugging is well known and there are several tools available for this function. For large systems, a technique called “partial evaluation” is available to identify and test isolate components of a system.

V. REVERSE ENGINEERING PROCESS

Reverse engineering process (REP) is a process in which a system is analysed for identifying its components and relationships among them and to represent it at higher level of abstraction or in another form.

Since, reverse engineering is a process of acquiring higher level of abstraction, there are five phases in which abstraction levels can be specified as follows:

1. Requirements
2. Features
3. Architecture
4. Design
5. Implementation

For reverse engineering process to be succeed two goals must be obtained:

1. Information extraction (e.g. gathering raw data)
2. Information abstraction (e.g. views and documents, design artifacts etc.)



Fig 1: Generic Structure of Reverse Engineering tools

VI. CASE STUDY OF UML TOOLS USED IN REVERSE ENGINEERING:

Here we have taken the comparison of two UML tools which generate UML structure for social networking sites from the available source code

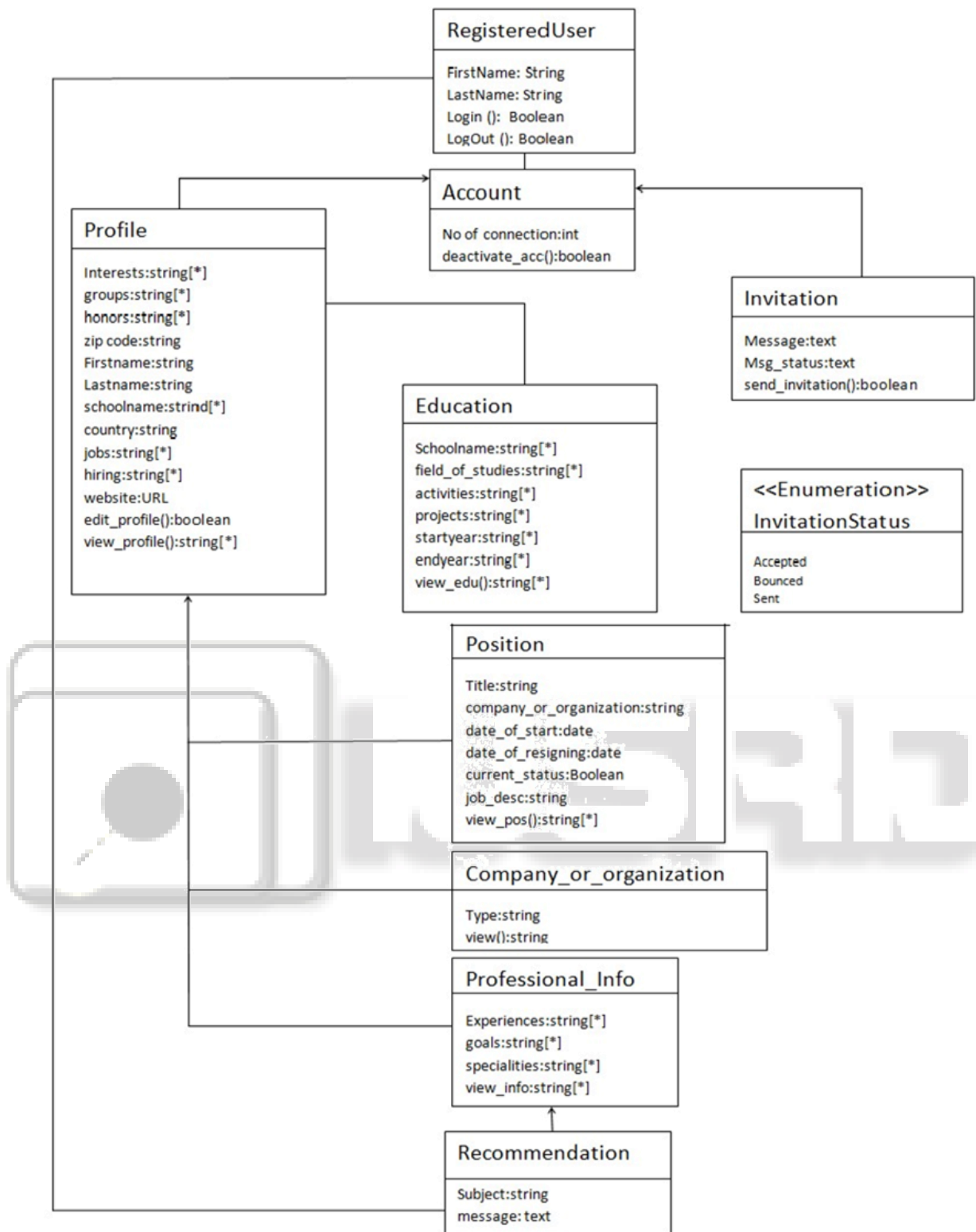


Fig. 2: UML diagram generated by Microsoft Visio from source code.

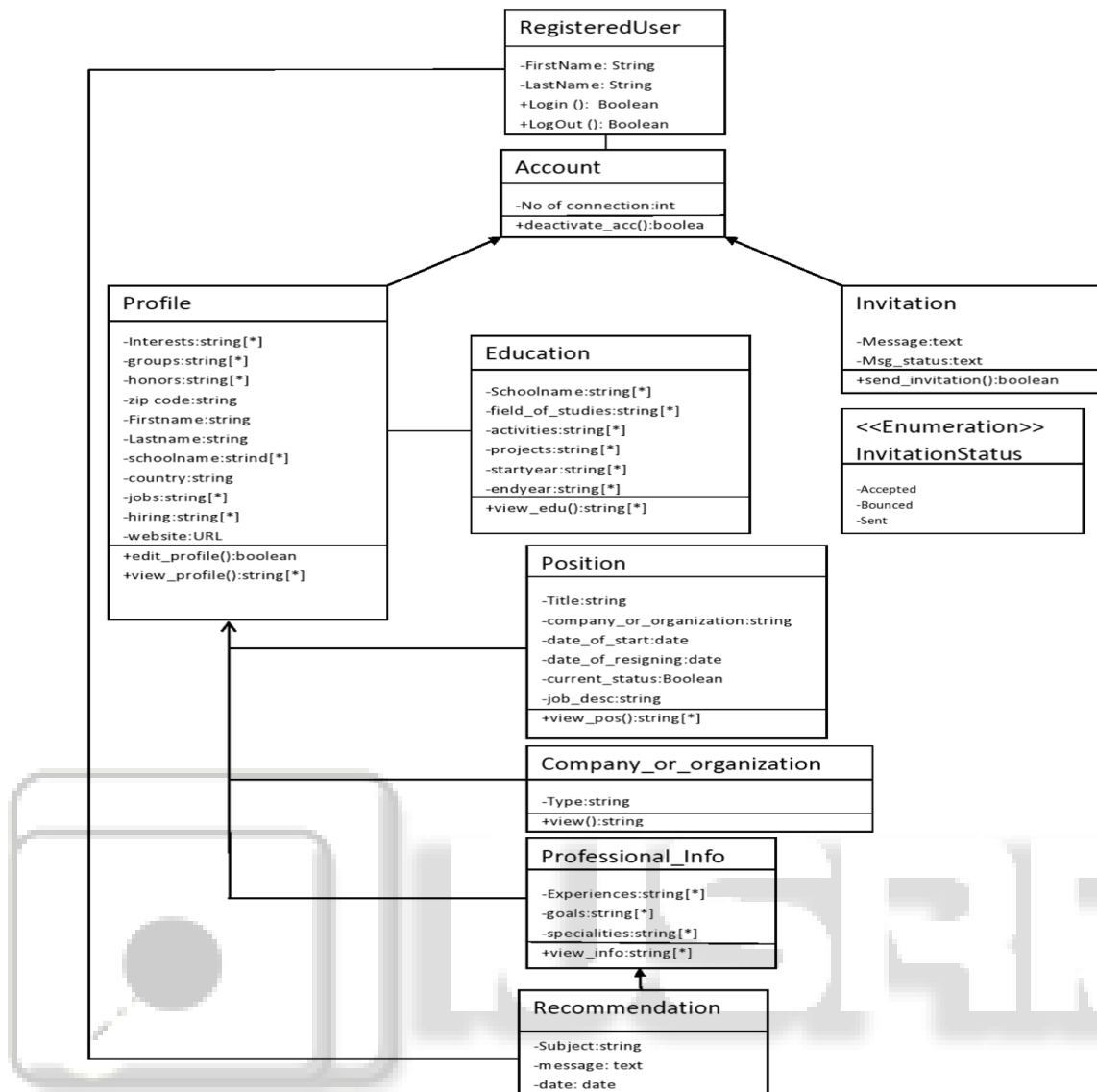


Fig. 3: UML diagram generated by Visual Paradigm from source code

VII. ADVANTAGES OF VISUAL PARADIGM TOOL OVER MICROSOFT VISIO

- VP-UML allow you to incorporate Visio drawings into any UML diagrams.
- Seamless integration with the major IDEs (e.g. Netbeans, eclipse, Microsoft Visual Studio etc.)
- Full integration with Microsoft Office
- Cost effective and affordable tools
- The most easy-to-use UML tools
- Multiple-platform support
- Textual analysis for identifying candidate classes
- Support for CRC cards
- Instant conversion of source code, binary and executable files into models
- Automatic diagram layout

VIII. CONCLUSIONS

This paper concludes with the study that although the resulting models are not so much consistent.

The primary reason for this inconsistency is the sizeable semantic gap between UML and source code language.

Microsoft Visio finds difficulty in reverse engineering associations while Visual Paradigm creates dependencies when associations are appropriate.

But still we can conclude that Visual paradigm is better UML for generating UML structure models than Microsoft Visio.

ACKNOWLEDGMENT

We(the authors), sincerely thank Prof.Dhaval Jha, Department of Computer Science and Information Technology, Nirma University for helping us throughout this paper.

We also thank anonymous readers for their valuable guidance.

REFERENCES

- [1] Elliot J. Chikofsky and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, January 1990.
- [2] <http://www.visual-aradigm.com/aboutus/10reasons.jsp>.
- [3] Tonella, P. and Potrich,, "Reverse Engineering of the UML Class Diagram from C++ Code in the Presence of Weakly Typed Containers", in Proceedings of

- International Conference on Software Maintenance (ICSM'01), Florence, Italy, Nov 6-10 2001, pp. 376-385.
- [4] P. Benedusi, A. Cimitile, and U. de Carlini. Reverse engineering processes, design document production, and structure charts. *Journal of Systems and Software*, 19(3):225–245, 1992.
- [5] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NH, 1981.
- [6] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, Jul 1989.
- [7] Chikofsky, E. J. and Cross, J. H., "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, vol. 7, no. 1, Jan.1990, pp. 13-17.

