Reverse Engineering

Idaho State University | Computer Science

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outcomes

After today's lecture you will:

- Have an understanding of the basic uses of Reverse Engineering
- Have an understanding of the Goals/Models/Tools paradigm
- Have an understanding of various techniques employed in reverse engineering
- Have a basic understanding of metrics employed
- Have an understanding of data reverse engineering
- Know what tools are out there to be used in reverse engineering

ROAR

Reverse Engineering

CS 4423/5523

ROAR

Reverse Engineering

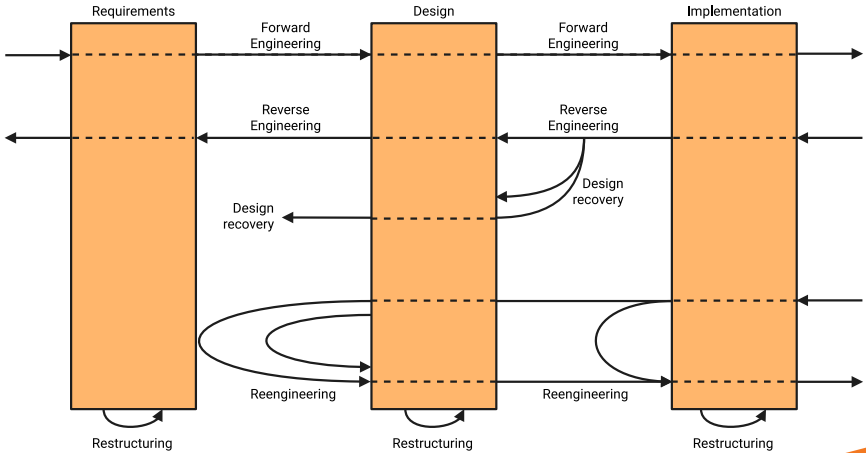
- Reverse engineering was first applied in electrical engineering to produce schematics from an electrical circuit
- It was defined as the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system
- In the context of software engineering, Chikofsky and Cross II defined reverse engineering as a process to:
  1. identify the components of an operational software
  2. identify the relationships among those components
  3. represent the system at a higher level of abstraction or in another form

Reverse Engineering

- Reverse engineering is performed to achieve two key objectives:
  - **redocumentation of artifacts**
    - It aims at revising the current description of components or generating alternative views at the same abstraction level. Examples of redocumentation are pretty printing and drawing CFGs
  - **design recovery**
    - It creates design abstractions from code, expert knowledge, and existing documentation

Reverse Engineering

- The relationship between forward engineering, reengineering, and reverse engineering



Reverse Engineering

- Six objectives of reverse engineering (Chkofsky and Cross II):
  - generating alternative views
  - recovering lost information
  - synthesizing higher levels of abstractions
  - detecting side effects
  - facilitating reuse
  - coping with complexity

Reverse Engineering

- Six key steps in reverse engineering (IEEE Standard for Software Maintenance):
  - **Local Analysis**
    - partition source code into units
    - describe the meanings of those units and identify the functional units
    - create the input and output schematics of the units identified before
  - **Global Analysis**
    - describe the connected units
    - describe the system application
    - create an internal structure of the system

Idaho State University
Computer Science

Reverse Engineering

Reverse engineering has been effectively applied in the following problem areas:

- redocumenting programs
- identifying reusable assets
- discovering design architectures
- recovering design patterns
- building traceability between code and documentation
- finding objects in procedural programs
- deriving conceptual data models
- detecting duplications and clones
- cleaning up code smells
- aspect-oriented software development
- computing change impact
- transforming binary code into source code
- redesigning user interfaces
- parallelizing largely sequential programs
- translating a program to another language
- migrating data
- extracting business rules
- wrapping legacy code
- auditing security and vulnerability
- extracting protocols of network applications

ROAR

Reverse Engineering

- A high level organizational paradigm is found to be useful while setting up a reverse engineering process, as advocated by Benedusi et al.
- The high level paradigm plays two roles:
  1. define a framework to use the available methods and tools
  2. allow the process to be repetitive
- The **Goals/Models/Tools** paradigm partitions a process for reverse engineering into three ordered stages:
  1. **Goals**
  2. **Models**
  3. **Tools**

Goals

- The reasons for setting up a process for reverse engineering are identified and analyzed
- Analyses are performed to identify the information needs of the process and the abstractions to be created by the process
- The team setting up the process first acquires a good understanding of the forward engineering activities and the environment where the products of the reverse engineering process will be used
- Results of the aforementioned comprehension are used to accurately identify:
  1. the information to be generated
  2. the formalisms to be used to represent the information

Models

- The abstractions identified in the Goals stage are analyzed to create representation models
- Representation models include information required for the generation of abstractions
- Activities in this phase are:
  - identify the kinds of documents to be generated
  - to produce those documents, identify the information and their relations to be derived from source code
  - define the models to be used to represent the information and their relationships extracted from source code
  - to produce the desired documents from those models, define the abstraction algorithm for reverse engineering

Models

- The important properties of a reverse engineering model are:
  - expressive power
  - language independence
  - compactness
  - richness of information content
  - granularity
  - support for information preserving transformation

Tools

- The tools needed for reverse engineering are identified, acquired, and/or developed in-house
- Tools are grouped into two categories:
  1. tools to extract information and generate program representations according to the identified models
  2. tools to extract information and produce the required documents.
     - Extraction tools general work on source code to reconstruct design documents
     - Therefore, those tools are ineffective in producing inputs for an abstraction process aiming to produce high-level design documents

Reverse Engineering Techniques

CS 4423/5523

ROAR

Reverse Engineering Techniques

- The well-known analysis techniques that facilitate reverse engineering are:
  1. Lexical analysis
  2. Syntactic analysis
  3. Control flow analysis
  4. Data flow analysis
  5. Program slicing
  6. Visualization
  7. Program metrics

Lexical Analysis

- **Lexical analysis**: the process of decomposing the sequence of characters in the source code into its constituent lexical units
- A program performing lexical analysis is called a lexical analyzer, and it is a part of a programming language's compiler
- Typically it uses rules describing lexical program structures that are expressed as regular expressions
- Modern lexical analyzers are automatically built using tools called lexical analyzer generators, namely, lex and flex

Syntax Analysis

- Syntactic analysis is performed by a parser
- Similar to lexical analyzers, parsers can be automatically constructed from a description of the programmatic properties of a programming language
- YACC is one of the most commonly used parsing tools

Syntax Analysis

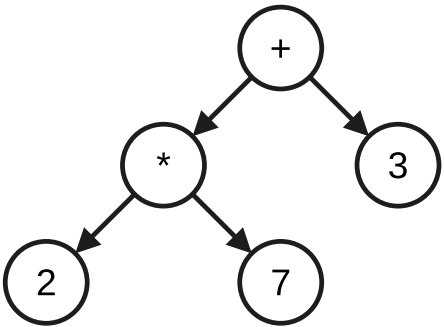
- Two types of representations are used to hold the results of syntactic analysis: **parse tree** and **abstract syntax tree**
  - A **parse tree** contains details unrelated to actual program meaning, such as the punctuation, whose role is to direct the parsing process
    - Grouping parentheses are implicit in the tree structure, which can be pruned from the parse tree
  - Removal of those extraneous details produces a structure called an **Abstract Syntax Tree (AST)**
    - An AST contains just those details that relate to the actual meaning of a program
- Many tools have been based on the AST concept; to understand a program, an analyst makes a query in terms of the node types
  - The query is interpreted by a tree walker to deliver the requested information

Idaho State University
Computer Science

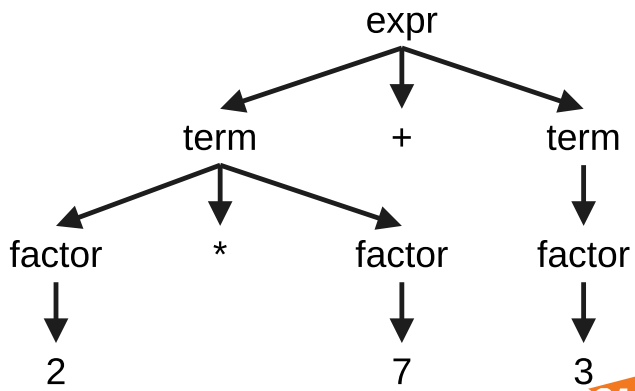
Syntax Analysis

2 * 7 + 3

AST



Parse Tree



ROAR

Control Flow Analysis

- After determining the structure of a program, control flow analysis (CFA) can be performed on it
- The two kinds of control flow analyses are:
  - **Intraprocedural**: it shows the order in which statements are executed within a subprogram
  - **Interprocedural**: it shows the calling relationship among program units

Intraprocedural Analysis

- The idea of basic blocks is central to constructing a CFG
- A basic block is a maximal sequence of program statements such that execution enters at the top of the block and leaves only at the bottom via a conditional or an unconditional branch statement
- A basic block is represented with one node in the CFG, and an arc indicates possible flow of control from one node to another
- A CFG can directly be constructed from an AST by walking the tree to determine basic blocks and then connecting the blocks with control flow arcs

Interprocedural Analysis

- Interprocedural analysis is performed by constructing a call graph
- Calling relationships between subroutines in a program are represented as a call graph which is basically a directed graph
- Specifically, a procedure in the source code is represented by a node in the graph, and the edge from node f to g indicates that procedure f calls procedure g
- Call graphs can be static or dynamic. A dynamic call graph is an execution trace of the program
- Thus a dynamic call graph is exact, but it only describes one run of the program
- On the other hand, a static call graph represents every possible run of the program

Data Flow Analysis

- **Data flow analysis (DFA)** concerns how values of defined variables flow through and are used in a program
- CFA can detect the possibility of loops, whereas DFA can determine data flow anomalies.
- One example of data flow anomaly is that an undefined variable is referenced.
- Another example of data flow anomaly is that a variable is successively defined without being referenced in between
- Data flow analysis enables the identification of code that can never execute, variables that might not be defined before they are used, and statements that might have to be altered when a bug is fixed

Data Flow Analysis

- Control flow analysis cannot answer the question: Which program statements are likely to be impacted by the execution of a given assignment statement?
- To answer these kinds of questions, an understanding of definitions (def) of variables and references (uses) of variables is required
- If a variable appears on the left hand side of an assignment statement, then the variable is said to be defined
- If a variable appears on the right hand side of an assignment statement, then it is said to be referenced in that statement

Program Slicing

- Originally introduced by Mark Weiser, program slicing has served as the basis of numerous tools
- In Weiser's definition, a slicing criterion of a program P is $S < p; v >$ where p is a program point and v is a subset of variables in P
- A program slice is a portion of a program with an execution behavior identical to the initial program with respect to a given criterion, but may have a reduced size
- Two approaches:
  - **Backward Slicing**
  - **Forward Slicing**

Program Slicing

- A **backward slice** with respect to a variable v and a given point p comprises all instructions and predicates which affect the value of v at point p
- **Backward slices** answer the question: **"What program components might effect a selected computation?"**
- The dual of **backward slicing** is **forward slicing**
- With respect to a variable v and a point p in a program, a forward slice comprises all the instructions and predicates which may depend on the value of v at p
- **Forward slicing** answers the question **"What program components might be affected by a selected computation?"**

Idaho State University
Computer Science

Example: Backward Slicing

Code

```
[1] int i;
[2] int sum = 0
[3] int product = 1;
[4] for (i = 0; ((i < N) &&
        (i % 2 = 0)); i++) {
[5]   sum = sum + i;
[6]   product = product * i;
    }
[7] printf("Sum = ", sum);
[8] printf("Product = ", product);
```

Slice with criterion `S <[7], sum>`

```
[1] int i;
[2] int sum = 0;
[4] for (i = 0; ((i < N) &&
        (i % 2 = 0)); i++) {
[5]   sum = sum + i;
    }
[7] printf("Sum = ", sum);
```


ROAR

Idaho State University
Computer Science

Example: Forward Slicing

Code

```
[1] int i;
[2] int sum = 0
[3] int product = 1;
[4] for (i = 0; ((i < N) &&
        (i % 2 = 0)); i++) {
[5]   sum = sum + i;
[6]   product = product * i;
    }
[7] printf("Sum = ", sum);
[8] printf("Product = ", product);
```

Slice with criterion `S <[3], product>`

```
[3] int product = 1;
[4] for (i = 0; ((i < N) &&
        (i % 2 = 0)); i++) {
[6]   product = product * i;
    }
[8] printf("Product = ", product);
```


ROAR

Visualization

- Software visualization is a useful strategy to enable a user to better understand software systems
- In this strategy, a software system is represented by means of a visual object to gain some insight into how the system has been structured.
- The visual representation of a software system impacts the effectiveness of the code analysis or design recovery techniques

Visualization

- Two important notions of designing software visualization using 3D graphics and virtual reality technology are
  - **Representation:** This is the depiction of a single component by means of graphical and other media
  - **Visualization:** It is a configuration of an interrelated set of individual representations related information making up a higher level component
- For effective software visualization, one needs to consider the properties and structure of the symbols used in software representation and visualization

Visualization

- When creating a representation the following key attributes are considered:

1. Individuality
2. High information content
3. Scalability of visual complexity
4. Flexibility for integration into visualizations
5. Distinctive appearance
6. Low visual complexity
7. Suitability for automation

Visualization

- The following requirements are taken into account while designing a visualization:

1. Simple navigation
2. High information content
3. Low visual complexity
4. Varying levels of detail
5. Resilience to change
6. Effective visual metaphors
7. Friendly user interface
8. Integration with other information sources
9. Good use of interactions
10. Suitability for automation

Program Metrics

- To understand and control the overall software engineering process, program metrics are applied
- The following table summarizes the commonly used program metrics

Metric	Description
Lines of Code (LOC)	The number of lines of executable code
Global Variable (GV)	The number of global variables
Cyclomatic Complexity (CC)	The number of linearly-independent paths in a program unit is given by the cyclomatic complexity metric [78]
Read Coupling	The number of global variables read by a program unit
Write Coupling	The number of global variables updated by a program unit

Program Metrics

Metric	Description
Address Coupling	The number of global variables whose addresses are extracted by a program unit but do not involve read/write coupling
Fan-in	The number of other functions calling a given function in a module
Fan-out	The number of other functions being called from a given function in a module
Halstead complexity (HC)	It is defined as effort: $E = D * V$, where: Difficulty (D): $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$; Volume (V): $V = N \times \log_2 n$; Program length: $N = N_1 + N_2$; Program vocabulary: $n = n_1 + n_2$ n_1 = the number of distinct operators n_2 = the number of distinct operands N_1 = the total number of operators N_2 = the total number of operands
Function points	A unit of measurement to express the amount of business functionality an information system provides to a user [79]. Function points are a measure of the size of computer applications and the projects that build them.

Program Metrics

- Based on a module's fan-in and fan-out information flow characteristics, Henry and Kafura define a complexity metric:
  - $Cp = (fanin \times fanout)$
- A large fan-in and a large fan-out may be symptoms of a poor design
- Six performance metrics are found in the Chidamber-Kemerer CK metric suite
  - **Weighted Methods per Class (WMC)** – This is the number of methods implemented within a given class
  - **Response for a Class (RFC)** – This is the number of methods that can potentially be executed in response to a message being received by an object of a given class
  - **Lack of Cohesion in Methods (LCOM)** – For each attribute in a given class, calculate the percentage of the methods in the class using that attribute. Next, compute the average of all those percentages, and subtract the average from 100 percent.
  - **Coupling Between Object classes (CBO)** – This is the number of distinct non-inheritance related classes on which a given class is coupled.
  - **Depth of Inheritance Tree (DIT)** – This is the length of the longest path from a given class to the root in the inheritance hierarchy.
  - **Number of Children (NOC)** – This is the number of classes that directly inherit from a given class

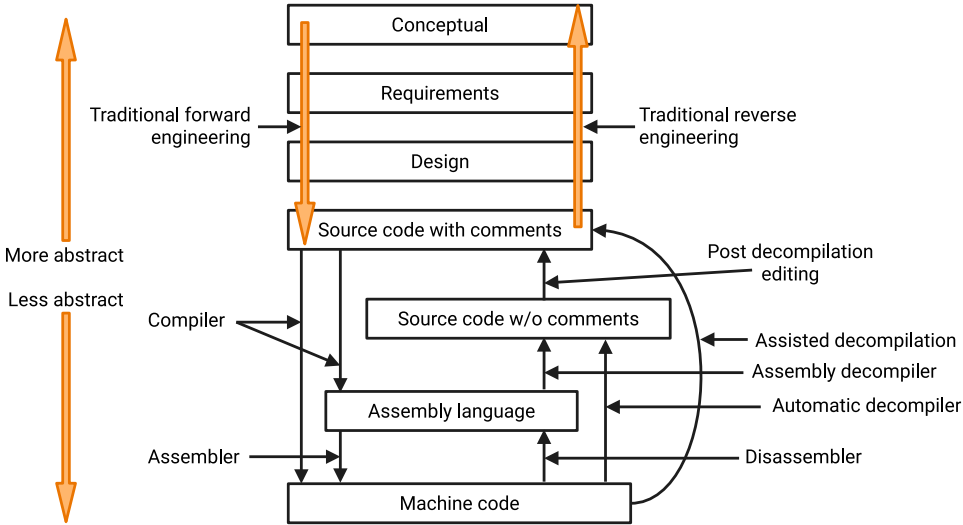
Program Metrics

- Kontogiannis et al. developed techniques to detect clones in programs using five kinds of metrics:
  - Fan-out
  - the ratio of the total count of input and output variables to the fan-out
  - Cyclomatic Complexity
  - Function Points metric
  - Henry and Kafura Information-Flow metric

Decompilation vs. Reverse Engineering

CS 4423/5523

Decompilation vs. Reverse Engineering



Decompilation vs. Reverse Engineering

- A decompiler takes an executable binary file and attempts to produce readable high-level language source code from it
- The output will, in general, not be the same as the original source code, and may not even be in the same language
- The decompiler does not provide the original programmers' annotations that provide vital instructions as to the functioning of the software
- Disassemblers are programs that take a program's executable binary as input and generate text files that contain the assembly language code for the entire program or parts of it
- Decompilation, or disassembly, is a reverse engineering process, since it creates representations of the system at a higher level of abstraction
  - Traditional reverse engineering from source code entails the recognition of "goals", or "plans", which must be known in advance
  - **Compilation** is **not** considered part of the forward engineering, since it is an automatic step

Data Reverse Engineering

CS 4423/5523

ROAR

Data Reverse Engineering

- **Data Reverse Engineering (DRE)** is defined as “the use of structured techniques to reconstitute the data assets of an existing system”
- By means of structured techniques, existing situations are analyzed and models are constructed prior to developing the new system
- The two vital aspects of a DRE process are:
  1. recover data assets that are valuable
  2. reconstitute the recovered data assets to make them more useful

Data Reverse Engineering

- The purpose of DRE is as follows:
  1. knowledge acquisition
  2. Tentative requirements
  3. Documentation
  4. Integration
  5. Data administration
  6. Data conversion
  7. Software assessment
  8. Quality assessment
  9. Component reuse

Data Reverse Engineering

- Reverse engineering of a **data-oriented application**, including its user interface, begins with DRE
- Recovering the specifications, that is the conceptual schema in database realm, of such applications is known as **database reverse engineering (DBRE)**
- A DBRE process facilitates understanding and redocumenting an application's database and files
- By means of a DBRE process, one can recreate the complete logical and **conceptual schemas** of a database **physical schema**

Data Reverse Engineering

- The **conceptual schema** is an abstract, implementation independent description of the stored data
- A **logical schema** describes the data structures in concrete forms as those are implemented by the data manager
- The **physical schema** of a database implements the logical schema by describing the physical construct
- Deep understanding of the forward design process is needed to reverse engineer a database

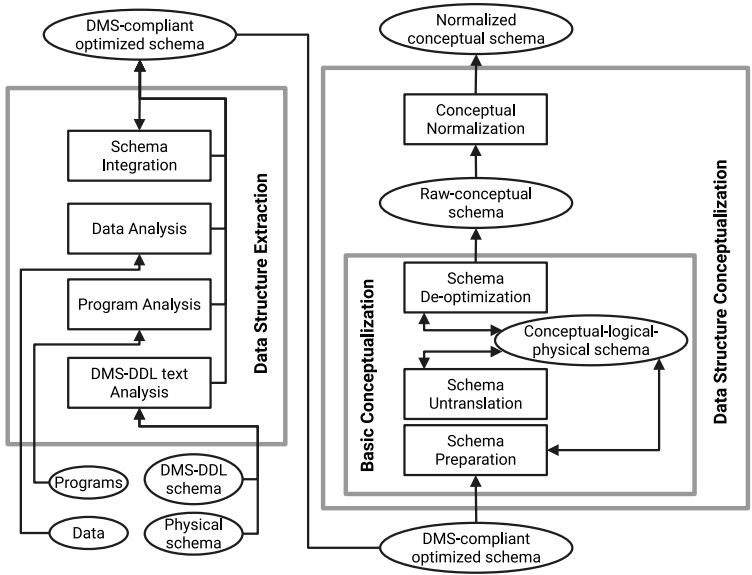
Data Reverse Engineering

- The forward design process of a database comprises three basic phases as follows:
  - **Conceptual Phase:** In this phase, user requirements are gathered, studied, and formalized into a conceptual schema
  - **Logical Phase:** In this phase, the conceptual schema is expressed as a simple model, which is suitable for optimization reasoning
  - **Physical Phase:** Now the logical schema is described in the data description language (DDL) of the data management system and the host programming language

Data Reverse Engineering

- A DBRE process is based on backward execution of the logical phase and the physical phase
- The process is divided into two main phases, namely
  - data structure extraction
  - data structure conceptualization
- The two phases relate to the recovery of two different schemas:
  1. the first one retrieves the present structure of data from their host language representation
  2. the second one retrieves a conceptual schema that describes the semantics underlying the existing data structures

Data Reverse Engineering



Data Structure Extraction

- The complete data management system schema, including the structures and constraints, are recovered in this phase
- Data structures are extracted by means of the following main processes:
  - **DMS-DDL text analysis:** Data structure declaration statements in a given DDL, found in the schema scripts and application programs, are analyzed to produce an approximate logical schema
  - **Program analysis:** This means analyzing the source code in order to detect integrity constraints and evidences of additional data structures

Idaho State University
Computer Science

Data Structure Extraction

- The complete data management system schema, including the structures and constraints, are recovered in this phase
- Data structures are extracted by means of the following main processes:
  - **Data analysis:** This means analyzing the files and databases to:
    1. identify data structures and their properties, namely, unique fields and functional dependencies in files
    2. test hypothesis such as “could this field be a foreign key to this file?”
  - **Schema integration:** The analyst is generally presented with several schemas while processing more than one information source. Each of those multiple schemas offers a partial view of the data objects. All those partials views are reflected on the final logical schema via a process for schema integration.

ROAR

Data Structure Conceptualization

- the phase comprises two sub-phases:
  - **basic conceptualization**
  - **conceptual normalization**

Basic Conceptualization

- In this sub-phase, relevant semantic concepts are extracted from an underlying logical schema, by solving two different problems requiring very different methods and reasoning:
  - schema untranslation
  - schema de-optimization
- However, first the schema is made ready by cleaning it before tackling those two problems

Making the schema ready:

1. The original schema might be using some concrete constructs, such as files and access keys, which might have been useful in the data structure extraction phase, but now can be eliminated
2. Some names can be translated to more meaningful names
3. Some parts of the schema might be restructured before trying to interpret them

Basic Conceptualization

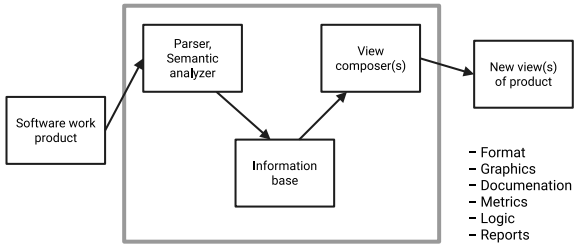
- **Schema untranslation:** The existing logical schema is a technical translation of the initial conceptual constructs. Untranslation means identifying the traces of those translations, and replacing them by their original conceptual constructs.
- **Schema de-optimization:** An optimized schema is generally more difficult to understand. Therefore, in a logical schema, it is useful to identify constructs included to perform optimization and replace those constructs

Conceptual Normalization

- The basic conceptual schema is restructured for it to have the desired qualities, namely:
  - simplicity
  - readability
  - minimality
  - extensibility
  - expressiveness
- Examples of conceptual normalization are:
  1. replace some entity types by relationship types
  2. replace some entity types by attributes
  3. make the is-a relation explicit
  4. standardize the names

Reverse Engineering Tools

- Software reverse engineering is a complex process that tools can only support, not completely automate
- There is a need of human intervention with any reverse engineering project
- The tools can provide a new view of the product
- The basic structure of reverse engineering tools is as follows:
  - The software system to be reverse engineered is analyzed
  - The results of the analysis are stored in an information base
  - View composers use the information base to produce alternative views of the system



Reverse Engineering Tools

- **Ada SDA (System Dependency Analyzer)** is a tool that supports the analysis and migration of Ada programs
- **CodeCrawler** is a language independent reverse engineering tool which combines metrics and software visualization
- **DMS (Design Maintenance System)** toolkit developed by Semantic Design, Inc. is composed of a set of tools for carrying out re-engineering of medium or large scale software systems
- **FermaT** is the generic name for a set of tools designed by Software Migration Ltd., specifically to support assembler code comprehension, maintenance, and migration
- **GXL (Graph eXchange Language)** is an XML-based format for sharing data between tools. GXL represents typed, attributed, directed, ordered graphs which are extended to represent hyper-graphs and hierarchical graphs.

Reverse Engineering Tools

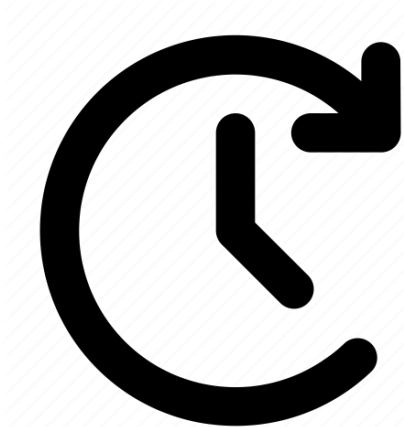
- **IDA Pro Disassembler and Debugger** by Hex-Rays is a powerful disassembler that supports more than fifty different processor architectures, including IA-32, IA-54 (Itanium), and AMD64
- **Hex-Rays** Decompiler is a commercial decompiler plug-in for IDA Pro
- **Imagix 4D** is useful in understanding legacy C, C++, and Java software
- **IRAP (Input-Output Reengineering and Program Crafting)** is a data reengineering tool developed by Spectra Research that provides a semi-automated approach to re-craft legacy software into an Intranet/Internet enabled application without compromising program computation integrity
- **JAD (JAva Decompiler)** is a Java decompiler written in C++
- **ManSART** is a tool to recover the architecture of a given software system
- **McCabe IQ** is capable of predicting some key issues in maintaining large and complex business software applications:
  1. locate error-prone sections of code
  2. identify the risk of system failure

Reverse Engineering Tools

- **PBS (Portable Bookshelf)** is an implementation of the web based concept called Software Bookshelf for the presentation navigation of information representing large software systems
- **RE-Analyzer** is an automated, reverse engineering system providing a high level of integration with a computer-aided software engineering (CASE) tool developed at IBM
- **Reengineering Assistant (RA)** aims to provide an interactive environment where software maintainers can reverse engineer source code into a higher abstraction level of representation
- **Rigi** is a software tool for comprehending large software systems. Software comprehension is achieved by performing reverse engineering on the given system
- **SEELA** is a reverse engineering tool developed by Tuval Software Industries to support the documentation and maintenance of structured source code
- **SciTools Understand** a program understanding tool which preforms metrics analysis, dependency analysis, control flow graph analysis, and many others.

For Next Time

- Review EVO Chapter 4.5 - 4.10
- Read EVO Chapter 5.1 - 5.3
- Watch the Lecture 10
- **4423** - Turn in Homework 01 by Sunday 1/31 at 11:00 pm
- **4423** - Quiz 03
- **4423** - Work on Project Part 02
- **5523** - Friday 1/29 Turn in Synthesis Topic Selection



Are there any questions?