CHAPTER 8
REVERSE ENGINEERING

- Overview of Reverse Engineering
- Program Analysis
- Architecture Recovery
- Software Complexity and Maintenance Metrics
- Program Visualization

"If you don't know where you are, you can't be sure you 're not travelling in circles "
Pickard & Carter

OVERVIEW OF REVERSE ENGINEERING

- Understanding a software system precedes any type of change.
- The *comprehension process* takes up a great deal of the total time spent on carrying out the change.
- The reasons for this include
  - Incorrect, out-of-date or non-existent documentation
  - The complexity of the system
  - Lack of sufficient domain knowledge on the part of the maintainer.
- ***Reverse engineering*** is a technique to abstract from the source code relevant information about the system, such as the specification and design, in a form that promotes understanding.
- ***Reverse engineering*** is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at higher levels of abstraction.
- Reverse engineering simply paves the way for easier implementation of the desired changes.
- Changes are implemented using techniques such as forward engineering, restructuring, and reengineering.

OVERVIEW OF REVERSE ENGINEERING

Definitions:

i. **Abstraction** - a "model that summarizes the detail of the subject it is representing".

ii. **Forward engineering** - the traditional software engineering approach starting with requirements analysis and progressing to implementation of a system.

iii. **Reengineering** - the process of examination and alteration whereby a system is altered by first reverse engineering and then forward engineering.

iv. **Restructuring** - the transformation of a system from one representational form to another.

v. **Reverse engineering** - the process of analyzing a subject system to:

   a. Identify the system's components and their interrelationships and

   b. Create representations of the system in another form or at higher levels of abstraction.

OVERVIEW OF REVERSE ENGINEERING

- The goal of reverse engineering 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.
- The objectives in pursuit of this goal are:

Objectives	Benefits
1. To recover lost information	1. Maintenance
2. To facilitate migration between platforms	(a) enhances understanding, which assists identification of errors
3. To improve and/or provide documentation	(b) facilitates identification and extraction of components affected by adaptive and perfective changes
4. To provide alternative views	(c) provides documentation or alternative views of the system
5. To extract reusable components	2. Reuse: Supports identification and extraction of reusable components
6. To cope with complexity	3. Improved quality of system
7. To detect side effects	
8. To reduce maintenance effort	

OVERVIEW OF REVERSE ENGINEERING

Abstraction

- Abstraction is achieved by highlighting the important features of the subject system and ignoring the irrelevant ones.
- There are **three types of abstraction** that can be performed on software systems:

I. **Function Abstraction:** eliciting functions from the target system - those aspects which operate on data objects and produce the corresponding output.

II. **Data Abstraction:** eliciting from the target system data objects as well as the functions that operate on them.

- An example of data abstraction at the design level in object-oriented systems is the encapsulation of an object type and its associated operations in a module or class.

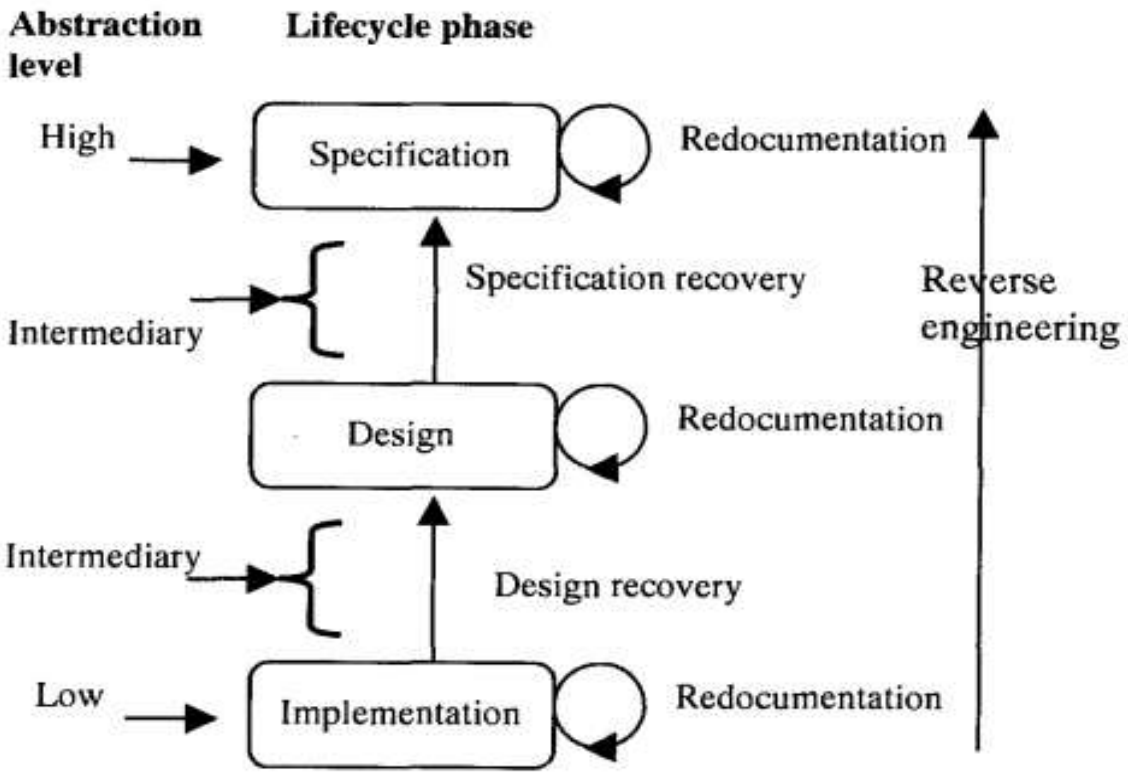
III. **Process Abstraction:** This is the abstracting from the target system of the exact order in which operations are performed.

- **Concurrent** processes communicate via shared data that is stored in a designated memory space.
- **Distributed** processes usually communicate through 'message passing' and have no shared data area.

OVERVIEW OF REVERSE ENGINEERING

Levels of Reverse Engineering

- Reverse engineering involves performing one or more of the above types of abstraction, in a bottom-up and incremental manner.



OVERVIEW OF REVERSE ENGINEERING

Factors that motivate the application of reverse engineering

Indicator	Motivation
Missing or incomplete design/specification	Product / environment
Out-of-date, incorrect or missing documentation related	
Increased program complexity	
Poorly structured source code	
Need to translate programs into a different programming language	
Need to make compatible products	
Need to migrate between different software or hardware platforms	
Static or increasing bug backlog Maintenance process	Maintenance process related
Decreasing personnel productivity related	
Need for continuous and excessive corrective change	
Need to extend economic life of system	Commercially related
Need to make similar but non-identical product	

PROGRAM ANALYSIS

- Program Analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.
- Extracting information, in order to *present abstractions* of, or *answer questions* about, a software system.

Static Analysis

- Examines the source code
- Static code analysis is a method of debugging by examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules.

Dynamic Analysis

- Examines the system as it is executing
- Dynamic analysis is the testing and evaluation of a program by executing data in real-time.
- The objective is to find errors in a program while it is running.

PROGRAM ANALYSIS

TECHNIQUES USED FOR REVERSE ENGINEERING

- To extract information which is not explicitly available in source code, automated analysis techniques are used.
- The well-known analysis techniques that facilitate reverse engineering are
  I. Lexical analysis
  II. Syntactic analysis
  III. Control flow analysis
  IV. Data flow analysis
  V. Program slicing
  VI. Visualization
  VII. Program metrics

PROGRAM ANALYSIS

TECHNIQUES USED FOR REVERSE ENGINEERING

I. **Lexical Analysis:** Lexical analysis is the process of decomposing the sequence of characters in the source code into its constituent lexical units.

- Various useful representations of program information are enabled by lexical analysis.
- the most widely used program information is the cross reference listing.
- 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 in a mathematical notation called regular expressions.
- Modern lexical analyzers are automatically built using tools called lexical analyzer generators, namely, lex and flex

PROGRAM ANALYSIS CONT…

Lexical Analysis …

- A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.
- Tokens in compiler design are the sequence of characters which represents a unit of information in the source program.

Example of Lexical Analysis, Tokens,

- Consider the following code that is fed to Lexical Analyzer

```
#include <stdio.h>
int maximum(int x, int y) {
// This will compare 2 numbers
if (x > y)
return x;
else {
return y;
}
}
```

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

Non-Tokens

Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

PROGRAM ANALYSIS

II. Syntactic Analysis:

- Compilers and other tools such as interpreters determine the expressions, statements and modules of a program.
- Syntactic analysis is performed by a parser.
- the requisite language properties are expressed in a mathematical formalism called context-free grammars.
- Usually, these grammars are described in a notation called Backus–Naur Form (BNF).
- In the BNF notation, the various program parts are defined by rules in terms of their constituents.
- Similar to syntactic analyzers, parsers can be automatically constructed from a description of the programmatical properties of a programming language.
- YACC is one of the most commonly used parsing tools.

PROGRAM ANALYSIS

II. Syntactic Analysis …

- Two types of representations are used to hold the results of syntactic analysis: parse tree and abstract syntax tree.

a) **Parse tree** is the more primitive one of the two.

- A parse tree contains details unrelated to actual program meaning, such as the punctuation, whose role is to direct the parsing process.
- For instance, grouping parentheses are implicit in the tree structure, which can be pruned from the parse tree.

b) 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.

PROGRAM ANALYSIS

III. 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 analysis 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.

PROGRAM ANALYSIS

III. Control Flow Analysis:

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.

PROGRAM ANALYSIS

IV. 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.

PROGRAM ANALYSIS

IV. 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 this kind 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 ANALYSIS

V. 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.
- 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 slide 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 effected by a selected computation?"***

PROGRAM ANALYSIS

V. Program Slicing (Example of Backward Slice)

```
[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);
```

Figure 4.11: A block of code to compute the sum and product of all the even integers in the range $[0, N)$ for $N \geq 3$.

```
[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);
```

Figure 4.12: The backward slice of code obtained from Figure 4.11 by using the criterion $S <$ [7]; sum $>$.

PROGRAM ANALYSIS

V. Program Slicing (Example of Forward Slice)

```
[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);
```

Figure 4.11: A block of code to compute the sum and product of all the even integers in the range $[0, N)$ for $N \geq 3$.

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

Figure 4.13: The forward slice of code obtained from Figure 4.11 by using the criterion $S <$ [3]; product $>$.

PROGRAM ANALYSIS

VII. Program metrics: …

- To understand and control the overall software engineering process, program metrics are applied.
- 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 [74].
Read coupling	The number of global variables read by a program unit
Write coupling	The number of global variables updated by a program unit
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: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$; Volume: $V = N \times \log_2 n$ 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	It is a unit of measurement to express the amount of business functionality an information system provides to a user [75]. Function points are a measure of the size of computer applications and the projects that build them

PROGRAM ANALYSIS

VII. Program metrics:

- To Based on a module's fan-in and fan-out information flow characteristics, Henry and Kafura define a complexity metric,

  Cp = (fan-in × fan- out).

- 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 attributes. Next, compute the average of all those percentages, and subtract the average from 100 percent.
  - Coupling between Object Class (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.

ARCHITECTURE RECOVERY

What is Software Architecture?

the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

- The architecture of a system consists of:
- the *structure(s)* of its parts
  - including design-time, test-time, and run-time hardware and software parts
- the *externally visible properties* of those parts
  - modules with interfaces, hardware units, objects
- the *relationships and constraints* between them in other words:
  - The set of design decisions about any system (or subsystem) that keeps its implementers and maintainers from exercising “needless creativity”
    - D’Souza, Will

ARCHITECTURE RECOVERY

Architectural Viewpoints

- "A viewpoint is a specification of the conventions for constructing and using a views. A pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis."

Run-time	How are responsibilities distributed amongst run-time entities?
Process	How many concurrent threads/processes exist; how do they communicate and synchronize?
Dataflow	How do data and tasks flow through the system?
Deployment	How are components physically distributed?
Module	How is the software partitioned into modules?
Build	What dependencies exist between modules?
File	How is the software physically distributed in the file system?

ARCHITECTURE RECOVERY

Architectural Styles

- An architectural style defines a family of systems in terms of a pattern of structural organization.
- More specifically, an architectural style defines a vocabulary of **components** and **connector** types, and a set of **constraints** on how they can be combined.

— Shaw and Garlan

- Each style describes a system category that encompasses:

(1) a set of components (e.g., a database, computational modules) (e.g., a database, computational modules) that perform a function required by a system,

(2) a set of connectors that enable "communication, coordination and cooperation" among components,

(3) constraints that define how components can be integrated to form the system, and

(4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

ARCHITECTURE RECOVERY

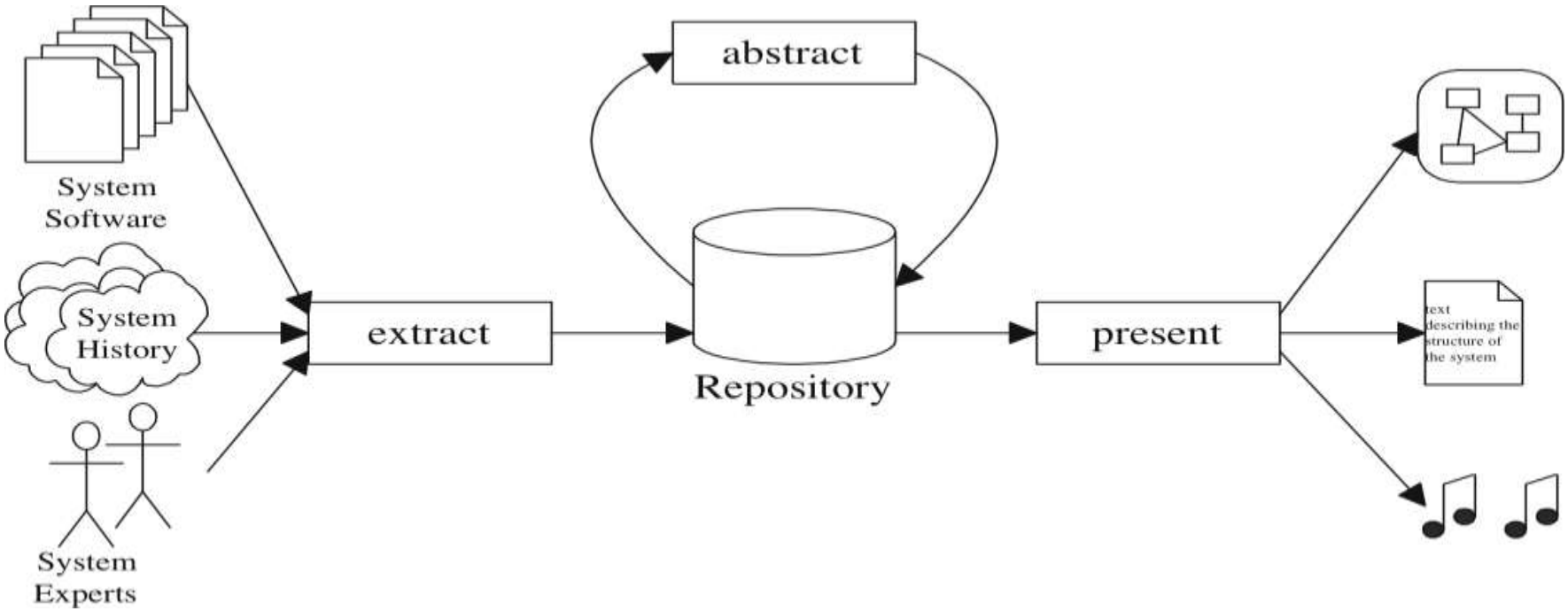
Some Classical Styles

Layered	• A layer is a set of subsystems which are able to provide related services, that can be realized by exploiting services from other layers • Elements in a given layer can only see the layer below. Callbacks used to communicate upwards
Client-Server	• Separate application logic from interaction logic. Clients may be "fat" or "thin" • The driving idea is to make unbalanced the partners' roles within a communication process.
Tiered	• Partitioning a system into sub-systems (peers), which are responsible for a class of services. • Typical tiered architecture include: presentation, logical and data tier.
Dataflow	• Data or tasks strictly flow "downstream". Pipes and filter, batch sequential
Blackboard	• Tools or applications coordinate through shared repository.

ARCHITECTURE RECOVERY

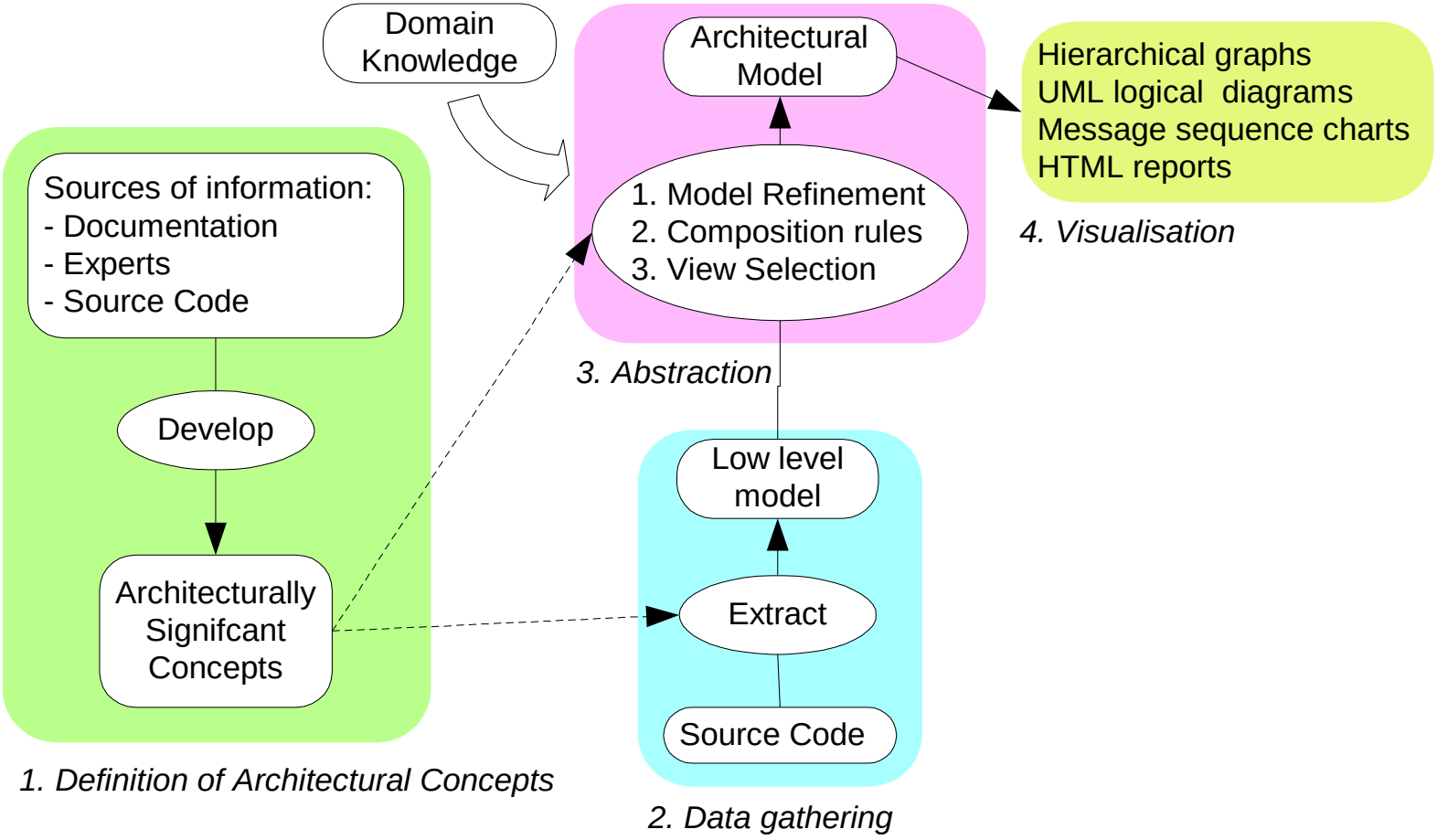
Architecture Extraction

Extract-abstract-present



ARCHITECTURE RECOVERY

Architecture Reconstruction



ARCHITECTURE RECOVERY

- Aims to recover the overall system structure in terms of its high-level components and the way they interact
- There are several techniques
  - Using human experts
  - Recognizing known patterns
  - Static and dynamic analysis
  - Clustering and data mining

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

Software Complexity and Maintenance Metrics

- Code complexity correlates with the defect rate and robustness of the application program

Code with good complexity:

- contains less errors
- is easier and faster to test
- is easier to understand
- is easier to maintain

- To obtain a high quality software with low cost of testing and maintenance, the code complexity should be measured as early as possible in coding.

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

McCabb'e Cyclomatic Complexity

- Cyclomatic Number is one of the metric based on not program size but more on information/control flow.
- It is based on specification flow graph representation developed by Thomas J Mc Cabb in 1976.
- Program graph is used to depict control flow.
- Nodes are representing processing task (one or more code statement) and edges represent control flow between nodes.
- To compute Cyclomatic Number by V(G) as following methods:

$$V(G) = E - N + 2P$$

Where,

- V (G)= Cyclomatic Complexity
- E= the number of edges in a graph
- N= the number of nodes in graph
- P= the number of connected components in graph,

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

- The problem with McCabb's Complexity is that, it fails to distinguish between different conditional statements (control flow structures).
- Also does not consider nesting level of various control flow structures. NPATH, have advantages over the McCabb's metric.

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

Halstead Software Science Complexity

- M. Halstead's introduced software science measures for software complexity product metrics.
- Halseatd's software science is based on a enhancement of measuring program size by counting lines of code.
- Halstead's metrics measure the number of operators and the number of operands and their respective occurrence in the program (code).
- These operators and operands are to be considered during calculation of Program Length, Vocabulary, Volume, Potential Volume, Estimated Program Length, Difficulty, and Effort and time by using following formulae.
  - n1 = number of unique operators,
  - n2 = number of unique operands,
  - N1 = total number of operators, and
  - N2 = total number of operands,
- Here we shall concentrate on those measures which impact on complexity: *program length* and *program effort*.

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

Program Length (N)

- The program length (N) is the sum of the total number of operators and operands in the program:
  N = N1 + N2

Program Effort(E)

- The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

- E = V * D
- V = N * log2(n)
- D = (n1 / 2) * (N2 / n2)

SOFTWARE COMPLEXITY & MAINTENANCE METRICS

Metrics for Maintenance

Software maturity index (SMI)

- Provides an indication of the stability of a software product based on changes that occur for each release
- SMI = [MT - (Fa + Fc + Fd)]/MT

  where

  MT = #modules in the current release

  Fa = #modules in the current release that have been added

  Fc = #modules in the current release that have been changed

  Fd = #modules from the preceding release that were deleted in the current release
- As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize
- The average time to produce a release of a software product can be correlated with the SMI

PROGRAM 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.

PROGRAM 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.

PROGRAM 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.

PROGRAM 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.

REFERENCE

- ✓ Penny Grub, Armstrong A Takang, Software Maintenance Concepts and Practice, 2nd edition
- ✓ Alain April, Alain Abran (2008), Software Maintenance Management Evaluation and Continuous Improvement.
- ✓ Pierre Bourque, École de technologie supérieure(2014), Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0, A Project of the IEEE Computer Society.
- ✓ https://www.verifysoft.com/en_software_complexity_metrics.pdf