Software Re-Engineering

Lecture: 08

Dr. Imran Ali
School of Computing- Software Engineering
FAST-National University of Computer & Emerging Sciences, Karachi

Sequence [Todays Agenda]

Content of Lecture

- Source Code Reengineering Reference Model (SCORE/RM)

Source Code Reengineering Reference Model (SCORE/RM)

- The Source Code Reengineering Reference Model (SCORE/RM) is useful in understanding the process of reengineering of software.
- The model was proposed by Colbrook, Smythe, and Darlison.

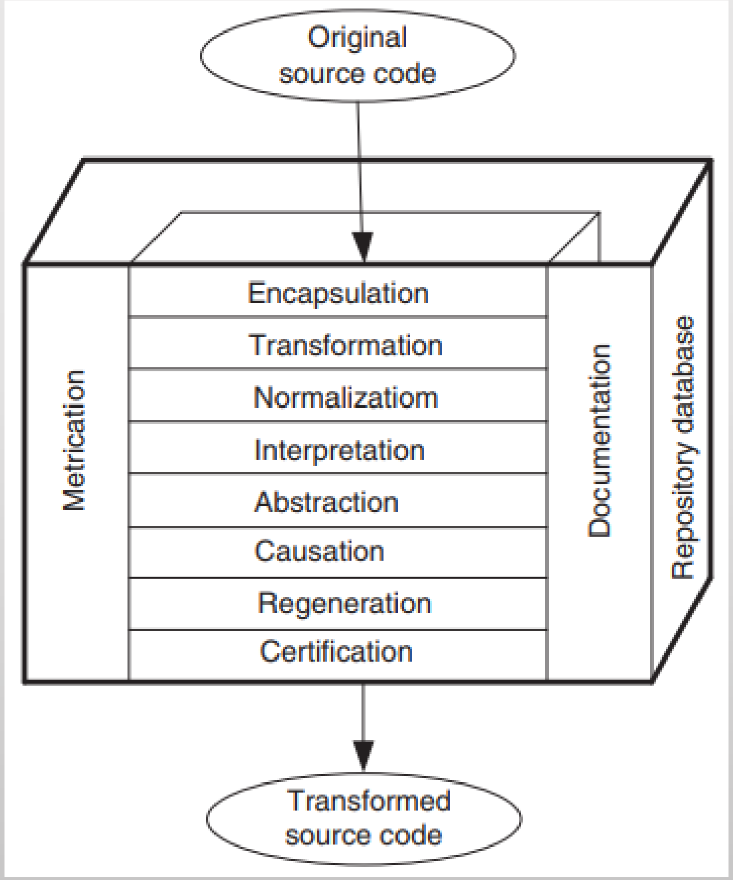
- The framework, represented in Figure, consists of four kinds of elements:

I. Function

II. Documentation

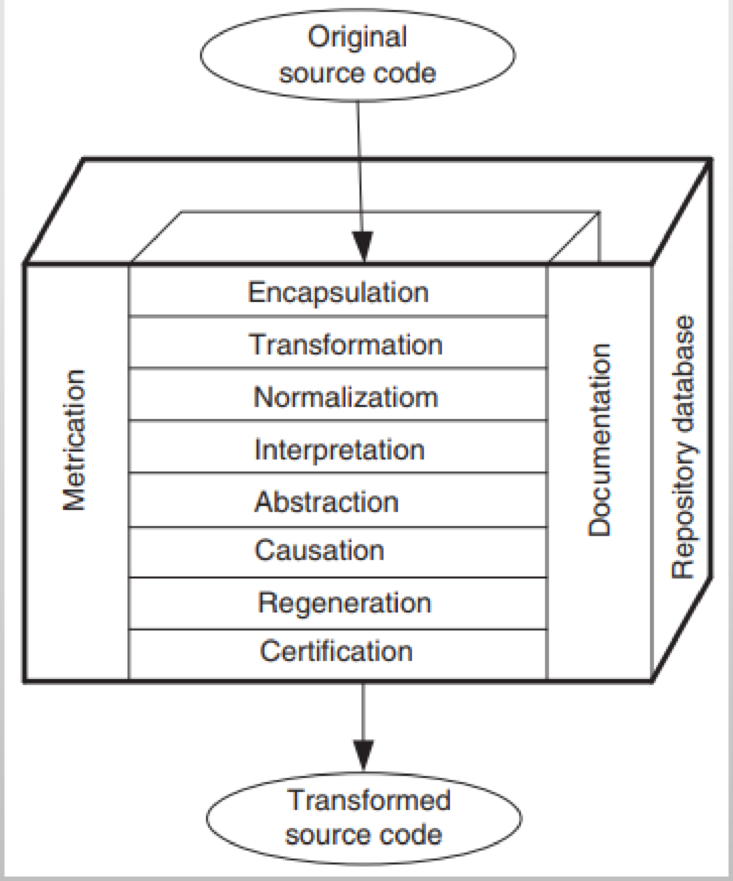
III. Repository database

IV. Metrication



- The function element is divided into eight layers, namely:

I. Encapsulation,
II. Transformation,
III. Normalization,
IV. Interpretation,
V. Abstraction,
VI. Causation,
VII. Regeneration,
VIII. Certification.



- The eight layers provide a detailed approach to:

I. Analyzing and optimizing the system to be reengineered by removing redundant data and altering the control flow;

II. Comprehending the software's requirements; and

III. Reconstructing the software according to established practices.

Metrication Element

- Improvements in the software as a result of reengineering are quantified by means of the metrication element.
- The metrication element is described in terms of the relevant software metrics before executing a layer and after executing the same layer.
- Product Metrics: LOC, Cyclomatic complexity, Defect density etc.
- Process Metrics: Lead time for changes, Deployment frequency, Change failure rate.
- Project Metrics: Used by the project managers such as schedule variance, and budget variance.

Function Element

- The top six of the eight layers shown in Figure constitute a process for reverse engineering, and the bottom three layers constitute a process for forward engineering.
- Both the processes include causation, because it represents the derivation of requirements specification for the software.

Documentation Element

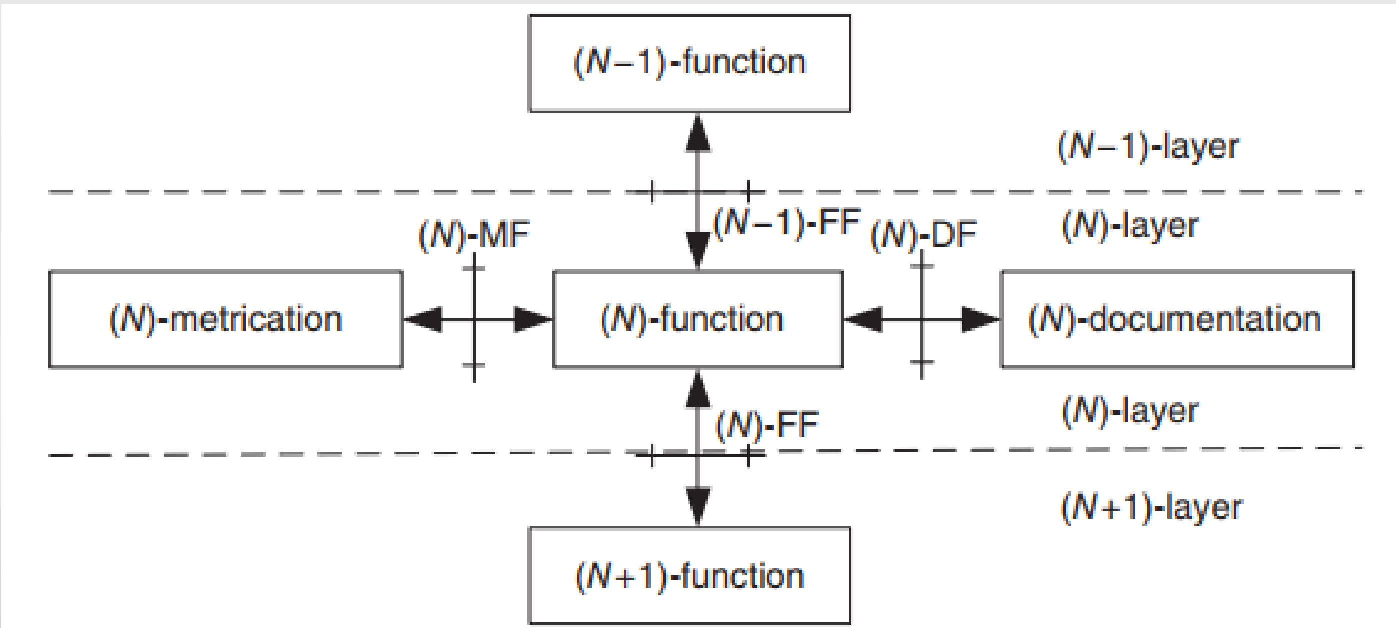
- The specification, constraints, and implementation details of both the old and the new versions of the software are described in the documentation element.

Repository Element

- The repository database is the information store for the entire reengineering process, containing information i.e. metrication, documentation, and both the old and the new source code.

- The interfaces among the elements are shown in Figure.
- For simplicity, any layer is referred to as (*N*)-layer, while its next lower and next higher layers are referred to as (*N* - 1)-layer and the (*N* + 1)-layer, respectively.

- The three types of interfaces are explained as follows:
- Metrication/function:
  - (*N*)-MF—the structures describing the metrics and their values.
- Documentation/function:
  - (*N*)- DF—the structures describing the documentation.
- Function/function:
  - (*N*)- FF—the representation structures for source code passed between the layers.



The interface nomenclature

Encapsulation

- This is the first layer of reverse engineering. In this layer, a reference baseline is created from the original source code.
- Reference baseline: The stable and well-defined version of a software product or system that serves as a fixed point of reference for future development and testing
- The goal of the reference baseline is to uniquely identify a version of a software and to facilitate its reengineering.
- The following functions are expected in this layer:
  - Configuration management
  - Analysis
  - Parsing
  - Test Generation

- Configuration management:
- A process for managing and controlling changes to a software product throughout its lifecycle, ensuring updates are tracked, implemented, and monitored to maintain system and reduce errors.
- The changes to the software undergoing maintenance are recorded by following a well-documented and defined procedure for later use in the new source code.
- This step requires strong support from upper management by allocating resources.
- Analysis:
- The portions of the software requiring reengineering are evaluated.
- In addition, cost models for the tangible benefits are put in place.

- <u>Parsing:</u>
- The source code of the system to be reengineered is translated into an intermediate language (IL).
- The IL can have several dialects, depending upon the relationship between the languages for the new code and the original code.
- All the reengineering algorithms act upon the IL representation of the source code.

- Parsing Intermediate Language (IL):
- In software engineering, parsing intermediate language (IL) serves to analyze and transform code into a machine-understandable format, ensuring syntax correctness and enabling further processing like optimization and code generation.
- Example: In Java, the Java compiler translates Java code into Java bytecode, which is an intermediate language, and then a Java Virtual Machine (JVM) interprets the bytecode and executes the program.

- <u>Test generation:</u>
- This refers to the design of certification tests and their results for the original source code. Certification tests are basically acceptance tests to be used as baseline tests.
- The “correctness” of the newly derived software will be evaluated by means of the baseline tests.

Transformation

- To make the code structured, its control flow is changed.
- This layer performs the following functions:
- Rationalization of control flow: The control flow is altered to make code structured.
- Isolation: All the external interfaces and referenced software are identified.
- Procedural granularity: This refers to the sizing of the procedures, by using the ideas of high cohesion and low coupling.

Normalization

- In this stage data and their structures are scrutinized by means of the following functions:
- Data reduction:
- Duplicate data are eliminated. To be consistent with the requirements of the program, databases are modified.
- Data representation:
- The life histories of the data entities are now generated. The life histories describe how data are changed and reveal which control structures act on the data.

Cyclomatic Complexity

- Cyclomatic complexity, developed by Thomas McCabe, is a metric that measures the complexity of a program by counting its decision points.
- It measures the number of unique paths through the code, indicating how complex the logic is.
- Lower complexity suggests simpler, more manageable code, reducing the chances of errors and making it easier to maintain and modify.
- Essentially, it helps assess the code’s readability and risk associated with changes.

Cyclomatic Complexity

- The cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it.
- It is a software metric used to indicate the complexity of a program and is computed using the control flow graph of the program.
- The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if the second command might immediately follow the first command.
- For example, if the source code contains no control flow statement then its cyclomatic complexity will be 1, and the source code contains a single path in it. Similarly, if the source code contains one if condition then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.

Formula for Calculating Cyclomatic Complexity

- Mathematically, for a structured program, the directed graph inside the control flow is the edge joining two basic blocks of the program as control may pass from first to second.
- So, cyclomatic complexity M would be defined as:

$$M = E - N + 2P$$

- *where*

  E = the number of edges in the control flow graph
  N = the number of nodes in the control flow graph
  P = the number of connected components

Formula for Calculating Cyclomatic Complexity

- In case, when exit point is directly connected back to the entry point.
- Here, the graph is strongly connected, and cyclometric complexity is defined as:

$$M = E - N + P$$

- *where*

  E = the number of edges in the control flow graph

  N = the number of nodes in the control flow graph

  P = the number of connected components

Formula for Calculating Cyclomatic Complexity

- In the case of a single method, P is equal to 1. So, for a single subroutine, the formula can be defined as:

$$M = E - N + 2$$

- *where*

  E = the number of edges in the control flow graph

  N = the number of nodes in the control flow graph

  P = the number of connected components

How to Calculate Cyclomatic Complexity

- Steps that should be followed in calculating cyclomatic complexity and test cases design are:
  - Construction of graph with nodes and edges from code.
  - Identification of independent paths.
  - Cyclomatic Complexity Calculation.
  - Design of Test Cases.

Example:

Design the Control Flow and find the cyclomatic complexity of code given below ?

```
A = 10
IF B > C THEN
        A = B
ELSE
        A = C
ENDIF
        Print A
        Print B
        Print C
```

Solution:

A = 10

IF B > C THEN

A = B

ELSE

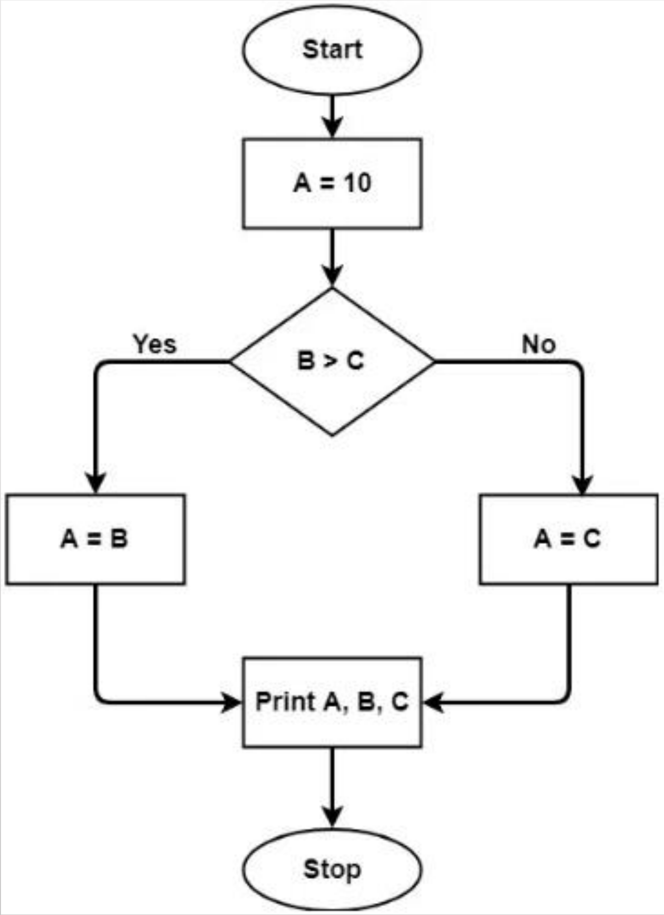
A = C

ENDIF

Print A

Print B

Print C



Solution:

- The cyclomatic complexity calculated for the previous code will be from the control flow graph.
- The graph shows seven shapes(nodes), and seven lines(edges), hence cyclomatic complexity is:

$$M = E - N + 2$$
$$= 7 - 7 + 2 = 2.$$

Interpretation:

- The process of deriving the meaning of a piece of software is started in this layer.
- The interpretation layer performs the following functions:
- Functionalization: This is additional rationalization of the data and control structure of the code, which (i) eliminates global variables (to reduce the complexity) and/or (ii) introduces recursion and polymorphic functions.
- Program reading: This means annotating the source code with logical comments.

Abstraction:

- The annotated and rationalized source code is examined by means of abstractions to identify the underlying object hierarchies.
- The abstraction layer performs the following functions:
- <u>Object identification:</u> The main idea in object identification is (i) separate the data operators and (ii) group those data operators with the data they manipulate.
- <u>Object interpretation:</u> Application domain meanings are mapped to the objects identified above. It is the different implementations of those objects that produce differences between the renovated code and the original code.

Causation:

- This layer performs the following functions:
- Specification of actions: This refers to the services provided to the user.
- Specification of constraints: This refers to the limitations within which the software correctly operates.
- Modification of specification: The specification is extended and/or reduced to accurately reflect the user's requirements.

Regeneration:

- Regeneration means reimplementing the source code using the requirements and the functional specifications.
- The layer performs the following functions:
- Generation of design: This refers to the production and documentation of the detailed design.
- Generation of code: This means generating new code by reusing portions of the original code and using standard libraries.
- Test generation: New tests are generated to perform unit and integration tests on the source code developed and reused.

Certification:

- The newly generated software is analyzed to establish that it is (i) operating correctly; (ii) performing the specified requirements; and (iii) consistent with the original code. The layer performs the following functions:
- <u>Validation and Verification:</u> The new system is tested to show its correctness.
- <u>Conformance:</u> Tests are performed to show that the renovated source code performs at the minimum all those functionalities that were performed by the original source code.


Thank You!