# Lecture 3. Reengineering

Reengineering, model of software reengineering, reengineering process, code reverse engineering, techniques of reverse engineering, program metrics, decompilation, data reverse engineering, reverse engineering tools [1], p. 133-174

Contents

[Lecture 3. Reengineering 1](#_Toc180253008)

[1. A GENERAL MODEL FOR SOFTWARE REENGINEERING 1](#_Toc180253009)

[1.1. Types of Changes 3](#_Toc180253010)

[1.2. Software Reengineering Strategies 6](#_Toc180253011)

[1.3. Reengineering Variations 7](#_Toc180253012)

[**2.** REENGINEERING PROCESS 8](#_Toc180253013)

[2.1. Reengineering Approaches 8](#_Toc180253014)

[2.2. Source Code Reengineering Reference Model 10](#_Toc180253015)

[2.3. Phase Reengineering Model 13](#_Toc180253016)

[**3.** CODE REVERSE ENGINEERING 14](#_Toc180253017)

[**4.** TECHNIQUES USED FOR REVERSE ENGINEERING 14](#_Toc180253018)

[4.1. **Lexical analysis** 14](#_Toc180253019)

[4.2. Syntactic Analysis 15](#_Toc180253020)

[4.3. Control Flow Analysis 15](#_Toc180253021)

[4.4. Data Flow Analysis 16](#_Toc180253022)

[4.5. Program Slicing 18](#_Toc180253023)

[4.6. Visualization 19](#_Toc180253024)

[4.7. Program Metrics 20](#_Toc180253025)

[**5.** DECOMPILATION VERSUS REVERSE ENGINEERING 22](#_Toc180253026)

[**6.** DATA REVERSE ENGINEERING 23](#_Toc180253027)

[**7.** REVERSE ENGINEERING TOOLS 24](#_Toc180253028)

# A GENERAL MODEL FOR SOFTWARE REENGINEERING

The reengineering process accepts as input the existing code of a system and produces the code of the renovated system. On the one hand, the reengineering process may be as straightforward as translating with a tool the source code from the given language to source code in another language. For example, a program written in BASIC can be translated into a new program in C. On the contrary, the reengineering process may be very complex as explained below:

* recreate a design from the existing source code;
* find the requirements of the system being reengineered;
* compare the existing requirements with the new ones;
* remove those requirements that are not needed in the renovated system;
* make a new design of the desired system; and
* code the new system.

Founded on the different levels of abstractions used in the development of software, Figure 4.3 depicts the processes for all abstraction levels of reengineering. This model suggests that reengineering is a sequence of three activities—reverse engineering, re-design, and forward engineering—strongly founded in three principles, namely, abstraction, alteration, and refinement, respectively.



A visual metaphor called *horseshoe*, as depicted in Figure 4.4, was developed to describe a three-step architectural reengineering process. Three distinct segments of the horseshoe are the left side, the top part, and the right side. Those three parts denote the three steps of the reengineering process. The first step, represented on the left side, aims at extracting the architecture from the source code by using the abstraction principle. The second step, represented on the top, involves architecture transformation toward the target architecture by using the alteration principle. Finally, the third step, represented on the right side, involves the generation of the new architecture by means of refinement. One can look at the horseshoe bottom-up to notice how reengineering progresses at different levels of abstraction: source code, functional model, and architectural design.



Now, we are in a position to revisit three definitions of reengineering.

* The definition by Chikofsky and Cross II [6]: Software reengineering is the analysis and alteration of an operational system to represent it in a new form and to obtain a new implementation from the new form. Here, a new form means a representation at a higher level of abstraction.
* The definition by Byrne [5]: Reengineering of a software system is a process for creating a new software from an existing software so that the new system is better than the original system in some ways.
* The definition by Arnold [3]: Reengineering of a software system is an activity that: (i) improves the comprehension of the software system or (ii) raises the quality levels of the software, namely, performance, reusability, and maintainability.

In summary, it is evident that reengineering entails: (i) the creation of a more abstract view of the system by means of some reverse engineering activities; (ii) the restructuring of the abstract view; and (iii) implementation of the system in a new form by means of forward engineering activities. This process is formally captured by Jacobson and Lindst¨orm [8] with the following expression:

Reengineering = Reverse engineering+Δ+Forward engineering*.*

A variant of reengineering in which the transformation is driven by a major technology change is called *migration*. Another common term used by practitioners of reengineering is *rehosting*. Rehosting means reengineering of source code without addition or reduction of features in the transformed targeted source code. Rehosting is most effective when the user is satisfied with the system’s functionality, but looks for better qualities of the system.

## Types of Changes

The model in Figure 4.3 suggests that an existing system can be reengineered by following one of four paths. The selection of a specific path for reengineering depends partly on the characteristics of the system to be modified. For a given characteristic to be altered, the abstraction level of the information about that characteristics plays a key role in path selection. Based on the type of changes required, system characteristics are divided into groups: *rethink, respecify, redesign, and recode*.

* + 1. *Recode.* Implementation characteristics of the source program are changed by recoding it. Source code level changes are performed by means of *rephrasing* and program translation. In the latter approach, a program is transformed into a program in a different language. On the other hand, rephrasing keeps the program in the same language. Examples of translation scenarios are *compilation*, *decompilation*, and *migration*. By means of compilation, one transforms a program written in a high-level language into assembly or machine code. Decompilation is a form of transformation in which high-level source code is discovered from an executable program. In migration, a program is transformed into a program in another language while retaining the program’s abstraction level. The language of the new program need not be completely different than the original program’s language; rather, it can be a variation of the first language. Examples of rephrasing scenarios are *normalization*, *optimization*, *refactoring*, and *renovation*.
			1. *Normalization* reduces a program to a program in a sublanguage that is to a subset of the language, with the purpose of decreasing its syntactic complexity. Elimination of GOTO and module flattening in a program are examples of program normalization.



From Meng2024FlattenRTLAO,

[https://api.semanticscholar.org/CorpusID:271798890](https://api.semanticscholar.org/CorpusID%3A271798890)



In Listing 2, one adder\_8bit is replaced by two calls of adder\_4bit:



* + - 1. *Optimization* is a transformation that improves the execution time or space performance of a program.
			2. *Refactoring* is a transformation that improves the design of a program by means of restructuring to better understand the new program.
			3. *Renovation.* An operational system is modified into the target system in the renovation phase. Two main aspects of a system are considered in this phase: (i) representation of the system and (ii) representation of external data. In general, the former refers to source code, but it may include the design model and the requirement specification of the existing system. On the other hand, the latter refers to the database and/or data files used by the system. Often the external data are reengineered, and it is known as *data reengineering*.
		1. *Redesign.* The design characteristics of the software are altered by redesigning the system. Common changes to the software design include: (i) restructuring the architecture; (ii) modifying the data model of the system; and (iii) replacing a procedure or an algorithm with a more efficient one.
		2. *Respecify.* This means changing the requirement characteristics of the system in two ways: (i) change the form of the requirements and (ii) change the scope of the requirements. The former refers to changing only the form of existing requirements, that is, taking the informal requirements expressed in a natural language and generating a formal specification in a formal description language, such as the Specification and Description Language (SDL) or Unified Modeling Language (UML). The latter type of changes includes such changes as adding new requirements, deleting some existing requirements, and altering some existing requirements.
		3. *Rethink.* Rethinking a system means manipulating the concepts embodied in an existing system to create a system that operates in a different problem domain. It involves changing the conceptual characteristics of the system, and it can lead to the system being changed in a fundamental way. Moving from the development of an ordinary cellular phone to the development of smartphone system is an example of Rethink.

##  Software Reengineering Strategies

Three strategies that specify the basic steps of reengineering are rewrite, rework, and replace. The three strategies are founded on three fundamental principles in software engineering, namely, abstraction, alteration, and refinement. The rewrite strategy is based on the principle of alteration. The rework strategy is based on the principles of abstraction, alteration, and refinement. Finally, the replace strategy is based on the principles of abstraction and refinement.



*Rewrite strategy.* This strategy reflects the principle of alteration. By means of alteration, an operational system is transformed into a new system while preserving the abstraction level of the original system. For example, the Fortran code of a system can be rewritten in the C language. The rewrite strategy has been further explained in Figure 4.5a.

*Rework strategy.* The rework strategy applies all the three principles. First, by means of the principle of abstraction, obtain a system representation with less details than what is available at a given level. For example, one can create an abstraction of source code in the form of a high-level design. Next, the reconstructed system model is transformed into the target system representation, by means of alteration, without changing the abstraction level. Finally, by means of refinement, an appropriate new

system representation is created at a lower level of abstraction. The main ideas in rework are illustrated in Figure 4.5b. Now, let us consider an example. Let the goal of a reengineering project be to restructure the control flow of a program. Specifically, we want to replace the unstructured control flow constructs, namely GOTOs, with more commonly used structured constructs, say, a “for” loop. A classical, rework strategy-based approach to doing that is as follows:

* Application of abstraction: By parsing the code, generate a control flow graph (CFG) for the given system.
* Application of alteration: Apply a restructuring algorithm to the CFG to produce a structured CFG.
* Application of refinement: Translate the new, structured CFG back into the original programming language.

*Replace strategy.* The replace strategy applies two principles, namely, abstraction and refinement. To change a certain characteristic of a system: (i) the system is reconstructed at a higher level of abstraction by hiding the details of the characteristic and (ii) a suitable representation for the target system is generated at a lower level of abstraction by applying refinement. Figure 4.5c succinctly represents the replace strategy. Let us reconsider the GOTO example above. By means of abstraction, a program is represented at a higher level without using control flow concepts. For instance, a module’s behavior can be described by its net effect, with no mention of control flow. Next, by means of refinement, the system is represented at a lower level of abstraction with a new structured control flow. In summary, the original unstructured control flow is replaced with a structured control flow.

##  Reengineering Variations

Three reengineering strategies and four broad types of changes were discussed in the preceding sections: (i) rewrite, rework, and replace are the three reengineering strategies and (ii) rethink, respecify, redesign, and recode are the four types of changes. The reengineering strategies and the change types can be combined to create different process variations. Three process factors cause variability in reengineering processes:

* the level of abstraction of the system representation under consideration;
* the kind of change to be made: rethink, respecify, redesign, and recode; and
* the reengineering strategy to be applied: rewrite, rework, and replace.

Possible variations in reengineering processes have been identified in Table 4.1. The table is interpreted by asking questions of the following type: If *A* is the abstraction level of the representation of the system to be reengineered and the plan is to make a *B* type of change, can I use strategy *C*? The table shows 30 reengineeringprocess variations. Out of the 30 variations, 24 variations are likely to produce acceptable solutions.

**TABLE 4.1 Reengineering Process Variations**

|  |  |  |
| --- | --- | --- |
| Starting Abstraction Level | Type Change | Reengineering Strategy |
| Rewrite | Rework | Replace |
| Implementation level | Recode | Yes | Yes | Yes |
| Redesign | Bad | Yes | Yes |
| Respecify | Bad | Yes | Yes |
| Rethink | Bad | Yes\* | Yes\* |
| Design level | Recode | No | No | No |
| Redesign | Yes | Yes | Yes |
| Respecify | Bad | Yes | Yes |
| Rethink | Bad | Yes\* | Yes\* |
| Requirement level | Recode | No | No | No |
| Redesign | No | No | No |
| Respecify | Yes | Yes | Yes |
| Rethink | Bad | Yes\* | Yes\* |
| Conceptual level | Recode | No | No | No |
| Redesign | No | No | No |
| Respecify | No | No | No |
| Rethink | Yes | Yes\* | Yes\* |

*Source:* From Reference 5. ⓒ 1992 IEEE.

Yes—One can produce a target system. Yes\*—Same as Yes, but the starting degree of abstraction is lower than the uppermost degree of abstraction within the conceptual abstraction level. No—One cannot start at abstraction level *A*, make *B* type of changes by using strategy *C*, because the starting abstraction level is higher than the abstraction level required by the particular type of change. Bad—A target system can be created, but the likelihood of achieving a good result is low.

# REENGINEERING PROCESS

An ordered set of activities designed to perform a specific task is called a *process*. In this section, we discuss two process models for software reengineering. By understanding and following a process model for software reengineering, one can achieve improvements in how software is reengineered. The process of reengineering a large software system is a complex endeavor. For ease of performing reengineering, the process can be specialized in many ways by developing several variations. In a reengineering process, the concept of *approach* impacts the overall process structure. If a particular process model requires fine-tuning for certain project goals, those approaches need to be clearly understood. Five major approaches will be explained in the following subsections.

## Reengineering Approaches

There are five basic approaches to reengineering software systems. Each approach advocates a different path to perform reengineering. Several considerations are made while selecting a particular reengineering approach:

* objectives of the project;
* availability of resources;
* the present state of the system being reengineered; and
* the risks in the reengineering project.

The five approaches are different in two aspects: (i) the extent of reengineering performed and (ii) the rate of substitution of the operational system with the new one.

*Big Bang approach.* The “Big Bang” approach replaces the whole system at once. Once a reengineering effort is initiated, it is continued until all the objectives of the project are achieved and the target system is constructed. This approach is generally used if reengineering cannot be done in parts. For example, if there is a need to move to a different system architecture, then all components affected by such a move must be changed at once. The consequent advantage is that the system is brought into its new environment all at once. On the other hand, the disadvantage of Big Bang is that the reengineering project becomes a monolithic task, which may not be desirable in all situations. In addition, the Big Bang approach consumes too much resources at once for large systems and takes a long stretch of time before the new system is visible.

*Incremental approach.* As the name indicates, by means of this approach a system is reengineered gradually, one step closer to the target system at a time. Thus, for a large system, several new interim versions are produced and released. Successive interim versions satisfy increasingly more project goals than their preceding versions. The desired system is said to be generated after all the project goals are achieved. The advantages of this approach are as follows: (i) locating errors becomes easier, because one can clearly identify the newly added components and (ii) it becomes easy for the customer to notice progress, because interim versions are released. The incremental approach incurs a lower risk than the “Big Bang” approach due to the fact that as a component is reengineered, the risks associated with the corresponding code can be identified and monitored. The disadvantages of the incremental approach are as follows: (i) with multiple interim versions and their careful version controls, the incremental approach takes much longer to complete; and (ii) even if there is a need, the entire architecture of the system cannot be changed.

*Partial approach.* In this approach, only a part of the system is reengineered and then it is integrated with the non-engineered portion of the system. One must decide whether to use a “Big Bang” approach or an “Incremental” approach for the portion to be reengineered. The following three steps are followed in the partial approach:

* In the first step, the existing system is partitioned into two parts: one part is identified to be reengineered and the remaining part to be not reengineered.
* In the second step, reengineering work is performed using either the “Big Bang” or the “Incremental” approach.
* In the third step, the two parts, namely, the not-to-be-reengineered part and the reengineered part of the system, are integrated to make up the new system.

The afore-described partial approach has the advantage of reducing the scope of reengineering to a level that best matches an organization’s current need and desire to spend a certain amount of resources. A reduced scope implies that the selected portions of a system to be modified are those that are urgently in need of reengineering. A reduced scope of reengineering takes less time and costs less. In the partial approach, modifications are not performed to the interface between the portion modified and the portion not modified.

*Iterative approach.* The reengineering process is applied on the source code of a few procedures at a time, with each reengineering operation lasting for a short time. This process is repeatedly executed on different components in different stages. During the execution of the process, ensure that the four types of the components can coexist: 1) old components not reengineered, 2) components currently being reengineered, 3) components already reengineered, and 4) new components added to the system. Their coexistence is necessary for the operational continuity of the system. There are two advantages of the iterative reengineering process: (i) it guarantees the continued operation of the system during the execution of the reengineering process and (ii) the maintainers’ and the users’ familiarities with the system are preserved. The disadvantage of this approach is the need to keep track of the four types of components during the reengineering process. In addition, both the old and the newly reengineered components need to be maintained.

*Evolutionary approach.* Similar to the ”Incremental” approach, in the ”Evolutionary” approach components of the original system are substituted with reengineered components. However, in this approach, the existing components are grouped by functions and reengineered into new components. Software engineers focus their reengineering efforts on identifying functional objects irrespective of the locations of those components within the current system. As a result, the new system is built with functionally cohesive components as needed. There are two advantages of the “Evolutionary” approach: (i) the resulting design is more cohesive and (ii) the scope of individual components is reduced. As a result, the “Evolutionary” approach works well to convert an operational system into an object-oriented system. In this approach, all the functions with much similarities must be first identified throughout the operational system; and next, those functions are refined as one unit in the new system.

## Source Code Reengineering Reference Model

The Source Code Reengineering Reference Model (SCORE/RM) is useful in understanding the process of reengineering of software. The framework, depicted in Figure 4.6, consists of four kinds of elements: function, documentation, repository database, and metrication. The function element is divided into eight layers, namely, encapsulation, transformation, normalization, interpretation, abstraction, causation, regeneration, and certification. The eight layers provide a detailed approach to (i) rationalizing 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. The top six of the eight layers shown in Figure 4.6 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. 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. The specification, constraints, and implementation details of both the old and the new versions of the software are described in the documentation element. The repository database is the information store for the entire reengineering process, containing the following kinds of information: metrication, documentation, and both the old and the new source codes. The interfaces among the elements are shown in Figure 4.7.



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 functions of the individual layers are discussed in the following.

**Encapsulation:** This is the first layer of reverse engineering. In this layer, a reference baseline is created from the original source code. 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 of this layer:

* *Configuration management.* 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.
* *Parsing.* 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. IL may use such graphs as call graph, control flow graph, dependence graph, etc. <https://www.academia.edu/85382863/A_toolset_for_the_reengineering_of_complex_computer_systems>
* *Test generation.* 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.

**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 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:

* *Object identification.* The main idea in object identification is (i) separate the data operators and (ii) group those data operators with the data they manipulate.
* *Object interpretation.* 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 re-implementing 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:

* *Validation* and *Verification.* The new system is tested to show its correctness.
* *Conformance.* 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.

## Phase Reengineering Model

The model comprises five phases: analysis and planning, renovation, target system testing, redocumentation, and acceptance testing and system transition, as depicted in Figure 4.8. The labels on the arcs denote the possible information that flows from the tail entities of the arcs to the head entities. A major process activity is represented by each phase. Tasks represent a phase’s activities, and tasks can be further decomposed to reveal the detailed methodologies.



*Analysis and planning.* The first phase of the model is analysis and planning. Analysis addresses three technical and one economic issue. The first technical issue concerns the present state of the system to be reengineered and understanding its properties. The second technical issue concerns the identification of the need for the system to be reengineered. The third technical issue concerns the specification of the characteristics of the new system to be produced.



*Renovation.* An operational system is modified into the target system in the renovation phase. Two main aspects of a system are considered in this phase: (i) representation of the system and (ii) representation of external data. In general, the former refers to source code, but it may include the design model and the requirement specification of the existing system. On the other hand, the latter refers to the database and/or data files used by the system. Often the external data are reengineered, and it is known as *data reengineering*.

# CODE REVERSE ENGINEERING

The factors necessitating the need for reverse engineering are as follows:

* The original programmers have left the organization.
* The language of implementation has become obsolete, and the system needs to be migrated to a newer one.
* There is insufficient documentation of the system.
* The business relies on software, which many cannot understand.
* The company acquired the system as part of a larger acquisition and lacks access to all the source code.
* The system requires adaptations and/or enhancements.
* The software does not operate as expected

Six key steps in reverse engineering, as documented in the IEEE Standard for Software Maintenance, are:

* 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;
* describe the connected units;
* describe the system application; and
* create an internal structure of the system

# TECHNIQUES USED FOR REVERSE ENGINEERING

## **Lexical analysis**

It 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. Perhaps 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 (fast lexical analyzer).

The usual context of wildcard characters is in [globbing](https://en.wikipedia.org/wiki/Glob_%28programming%29) similar names in a list of files, whereas regexes are usually employed in applications that pattern-match text strings in general. For example, the regex ^[ \t]+|[ \t]+$ matches excess whitespace at the beginning or end of a line. An advanced regular expression that matches any numeral is [+-]?(\d+(\.\d\*)?|\.\d+)([eE][+-]?\d+)?

It is from <https://en.wikipedia.org/wiki/Regular_expression>, <https://www.geeksforgeeks.org/write-regular-expressions/>

<https://www.w3schools.com/jsref/jsref_regexp_not_0-9.asp>

## Syntactic Analysis

The next most complex form of automated program analysis is syntactic in nature. Compilers and other tools such as interpreters determine the expressions, statements, and modules of a program. Syntactic analysis is performed by a parser. Here, too, 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.

As an example, consider this possible BNF for a U.S. [postal address](https://en.wikipedia.org/wiki/Address_%28geography%29):

 <**postal-address**> ::= <**name-part**> <**street-address**> <**zip-part**>

 <**name-part**> ::= <**personal-part**> <**last-name**> <**opt-suffix-part**> <**EOL**> | <**personal-part**> <**name-part**>

 <**personal-part**> ::= <**first-name**> | <**initial**> "."

 <**street-address**> ::= <**house-num**> <**street-name**> <**opt-apt-num**> <**EOL**>

 <**zip-part**> ::= <**town-name**> "," <**state-code**> <**ZIP-code**> <**EOL**>

<**opt-suffix-part**> ::= "Sr." | "Jr." | <**roman-numeral**> | ""

 <**opt-apt-num**> ::= "Apt" <**apt-num**> | ""

<https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form>

Two types of representations are used to hold the results of syntactic analysis: *parse tree* and *abstract syntax tree*. The former is the more primitive one of the two. It is similar to the parsing diagrams used to show how a natural language sentence is broken up into its constituents. However, 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. Removal of those extraneous details produces a structure called an *Abstract Syntax Tree* (AST).

## Control Flow Analysis

After determining the structure of a program, control flow analysis (CFA) can be performed on it. The two kinds of CFA are *intraprocedural analysis* and *interprocedural analysis*. The former shows the order in which statements are executed within a subprogram, whereas the latter shows the calling relationship among program units. Intraprocedural analysis is performed by generating CFGs of subprograms. 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. A CFG shows an abstract view of the ways in which a subprogram can execute.



From <https://www.cs.toronto.edu/~david/course-notes/csc110-111/17-graphs/08-control-flow-graphs.html>

## Data Flow Analysis

Although CFA is useful, many questions cannot be answered by means of CFA. For example, CFA 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. Normally, if a variable appears on the left-hand side of an assignment statement, then the variable is said to be defined. On the contrary, if a variable appears on the right-hand side of an assignment statement, then it is said to be referenced in that statement. 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. DFA 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.

From <https://en.wikipedia.org/wiki/Data-flow_analysis>

**Forward analysis**

The [reaching definition](https://en.wikipedia.org/wiki/Reaching_definition) analysis calculates for each program point the set of definitions that may potentially reach this program point.

1. if b == 4 then
2. a = 5;
3. else
4. a = 3;
5. endif
6. if a < 4 then

...

The reaching definition of variable a at line 6 is the set of assignments a = 5 at line 2 and a = 3 at line 4.

**Backward analysis**

The [live variable analysis](https://en.wikipedia.org/wiki/Live_variable_analysis) calculates for each program point the variables that may be potentially read afterwards before their next write update.

From <https://en.wikipedia.org/wiki/Live-variable_analysis>

Consider the following program:

1. b = 3
2. c = 5
3. a = f(b \* c)

The set of live variables between lines 2 and 3 is {b, c} because both are used in the multiplication on line 3. But the set of live variables after line 1 is only {b}, since variable c is updated later, on line 2. The value of variable a is not used in this code.

The result is typically used by [dead code elimination](https://en.wikipedia.org/wiki/Dead_code_elimination) to remove statements that assign to a variable whose value is not used afterwards.

1. int main(void) {
2. int a = 5;
3. int b = 6;
4. int c;
5. c = a \* (b / 2);
6. **if** (0) { */\* DEBUG \*/*
7. printf("%d**\n**", c);
8. }
9. **return** c;
10. }

Because the expression 0 will always evaluate to [false](https://en.wikipedia.org/wiki/False_%28logic%29), the code inside the if statement can never be executed, and dead-code elimination would remove it entirely from the optimized program.

The in-state of a block is the set of variables that are live at the start of it. It initially contains all variables live (contained) in the block, before the transfer function is applied and the actual contained values are computed. The transfer function of a statement is applied by killing the variables that are written within this block (remove them from the set of live variables). The out-state of a block is the set of variables that are live at the end of the block and is computed by the union of the block's successors' in-states.

// in: {}; predecessor blocks: none

b1: a = 3;

 b = 5;

 d = 4;

 x = 100; //x is never being used later thus not in the out set {a,b,d}

 if a > b then

// out: {a,b,d} //union of all (in) successors of b1 => b2: {a,b}, and b3:{b,d}

// in: {a,b}; predecessor blocks: b1

b2: c = a + b;

 d = 2;

// out: {b,d}

// in: {b,d}; predecessor blocks: b1 and b2

b3: endif

 c = 4;

 return b \* d + c;

// out:{}

The in-state of b3 only contains *b* and *d*, since *c* has been written. The out-state of b1 is the union of the in-states of b2 and b3. The definition of *c* in b2 can be removed, since *c* is not live immediately after the statement.

Solving the data flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed in-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **processing** | **out-state** | **old in-state** | **new in-state** | **work list** |
| b3 | {} | {} | {b,d} | (b1,b2) |
| b1 | {b,d} | {} | {} | (b2) |
| b2 | {b,d} | {} | {a,b} | (b1) |
| b1 | {a,b,d} | {} | {} | () |

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

## Program Slicing

The slice of a program for a given variable at a given line of code is the portion of the program that gives a value to the variable at that point. Therefore, if one determines during debugging that the value of a variable at a specific line is incorrect, one may look at the corresponding program slice to find the faulty code. 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.

[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* ≥ 3

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 affect 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*. Note that the statements in a forward program slice execute *after* the slicing criterion. Forward slicing answers the question “What program components might be affected by a selected computation?”. As an example, let us consider the program shown in Figure 4.11 which is a block of code in C. The *backward slice*, given the slicing criterion *S <* [7]; sum *>* is shown in Figure 4.12. For the slicing criterion *S <* [3]; product *>*, the forward program slice has been shown in Figure 4.13. Besides detecting defects, program slicing is also used to extract business rules and in refactoring which is discussed in Chapter 7.

[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 *>*

[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 *>*

## 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. Essentially, graphical symbols are used to represent components. In a call graph, for example, the nodes and arcs are the representations, whereas the graph itself is the visualization. For effective software visualization, one needs to consider the properties and structure of the symbols used in software representation and visualization.

## Program Metrics

To understand and control the overall software engineering process, program metrics are applied. Table 4.3 summarizes the commonly used program metrics. The early program metric research focused on *complexity metrics*, and one of the most widely used complexity metrics is cyclomatic complexity. The concept of function point (FP) was introduced in late 1970sas an alternative metrics based on simple source line count. The aim of FP is to measure the amount of functionality delivered by a program. Intuitively, the more functionality a program has, the larger is the FP count. Based on a module’s *fan-in* and *fan-out* information flow characteristics, a complexity metric is defined: *Cp* = (*fan-in* × *fan-out*)2. Fan-in and fan-out have been explained in Table 4.3. A large fan-in and a large fan-out may be symptoms of a poor design.

**TABLE 4.3 Commonly Used Software 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.

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* =(*n*1/2)×(*N*2/*n*2); Volume: *V* =*N*× log2 *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. Function points are a measure of the size of computer applications and the projects that build them

<https://www.geeksforgeeks.org/software-engineering-calculation-of-function-point-fp/>

From <https://en.wikipedia.org/wiki/Cyclomatic_complexity>:

|  |  |
| --- | --- |
| It is proposed to use a graph in which each exit point is connected back to the entry point. In this case, the graph is [strongly connected](https://en.wikipedia.org/wiki/Strongly_connected). Here, the cyclomatic complexity of the program is equal to the [cyclomatic number](https://en.wikipedia.org/wiki/Cyclomatic_number) of its graph (also known as the [first Betti number](https://en.wikipedia.org/wiki/Betti_number#Example_2:_the_first_Betti_number_in_graph_theory)), which is defined as[[2]](https://en.wikipedia.org/wiki/Cyclomatic_complexity#cite_note-mccabe76-2)M=E−N+P.where *E* = the number of edges of the graph, *N* = the number of nodes of the graph, *P* = the number of [connected components](https://en.wikipedia.org/wiki/Component_%28graph_theory%29).This may be seen as calculating the number of [linearly independent cycles](https://en.wikipedia.org/wiki/Linearly_independent_cycle) that exist in the graph: those cycles that do not contain other cycles within themselves. Because each exit point loops back to the entry point, there is at least one such cycle for each exit point. | undefinedEach exit point is connected back to the entry point. This graph has 10 edges, eight nodes and one [connected component](https://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29), which also results in a cyclomatic complexity of 3 (10 − 8 + 1 = 3). |
| **if** (c1()) f1();**else** f2();**if** (c2()) f3();**else** f4();The cyclomatic complexity of the program is 3 (as the strongly connected graph for the program contains 9 edges, 7 nodes, and 1 connected component) (9 − 7 + 1). | undefined |

In the 1990s, large-scale adoption of object-oriented (OO) programming techniques gave rise to some OO design metrics known as Chidamber and Kemerer (CK) metric suite. Six performance metrics are found in the CK metric suite as follows:

* 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. It is the number of methods implemented within a class plus the number of methods accessible to an object class due to inheritance.
* 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%.
* Coupling between object class (CBO)—This is the number of distinct non-inheritance-related classes on which a given class is coupled. A class is said to be coupled to another class if it uses methods or attributes of the other class.
* 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.

# DECOMPILATION VERSUS 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. Actual recovery of the original source code is not really possible. The relationship between decompilation and the traditional reengineering model (see Figure 4.3) is depicted in Figure 4.14.



It should be noted that it is never going to be possible to accurately predict beforehand whether or not a particular decompilation is going to be considered legal. License agreements may also bind the user to operate the program in a certain way and to avoid using decompilation or disassembly techniques on that program. It is recommended to seek legal counsel before starting any low-level reverse engineering project.

# DATA REVERSE ENGINEERING

There has been considerable effort to develop concepts and methods to reengineer *data-oriented applications*. A data-oriented application is centered around a set of permanent files or a database. As a persistent data structure is the central part of the data-oriented applications, most approaches focus on database *schema analysis* (data reverse engineering (DRE)) and/or *schema translation and redesign* (data forward engineering). In addition, the procedural portions of *data-oriented applications* have to be adapted to the newly redesigned schema in order to complete the reengineering task. In practice, the purpose of DRE is as follows:

* *Knowledge acquisition.* Knowledge acquisition is a method of learning. It includes elicitation, collection, analysis, modelling, and validation of information for software projects. The need for knowledge acquisition is pivotal to reverse engineering of *data-oriented applications*. The data portion—be it flat files or a relational database—must be clearly understood in a reverse engineering process.
* *Tentative requirements.* DRE of an operational system can identify the tentative requirements of the replacement system. DRE ensures that the functionality of the current system is not forgotten or overlooked.
* *Documentation.* DRE improves the documentation of existing systems, especially when the original developers are no longer available for advice. Maintenance of legacy software is assisted by the new documentation.
* *Integration.* DRE facilitates integration of applications, because (i) a logical model of encompassed software is a prerequisite for integration and (ii) a logical model of encompassed software presents a plausible model of how the program will function in certain environmental conditions.
* *Data administration.* As data are increasingly used as information, the data owner must be able to perform data administration easily and pragmatically.

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. The phase of conceptual schema has no impact on reverse engineering.
* *Logical phase*. In this phase, the conceptual schema is expressed as a simple model, which is suitable for optimization reasoning. Independent of the target DBMS, the model can be optimized. Next, it is translated according to the target model. The model can be further optimized according to data management system (DMS)-dependent rules.
* *Physical phase*. Now the logical schema is described in the data description language (DDL) of the DMS and the host programming language. The views needed by the application programs are expressed partly in DDL and partly in the host language.

A database reverse engineering (DBRE) process is based on backward execution of the logical phase and the physical phase, beginning with the results of the physical phase. The process is divided into two main phases, namely, *data structure extraction* and *data structure conceptualization*. The two phases relate to the recovery of two different schemas: (i) the first one retrieves the present structure of data from their DDL/host language representation and (ii) the second one retrieves a conceptual schema that describes the semantics underlying the existing data structures. Figure 4.15 shows the general architecture of a reference DBRE methodology.



# REVERSE ENGINEERING TOOLS

The tools can provide a new view of the product, as shown in Figure 4.16.

