Reverse Engineering

Forward engineering - the traditional software engineering approach starting with requirements analysis and progressing to implementation of a system.
Reengineering - the process of examination and alteration whereby a system is altered by first reverse engineering and then forward engineering
Restructuring - the transformation of a system from one representational form to another.
Reverse engineering - the process of analysing 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.

Abstraction

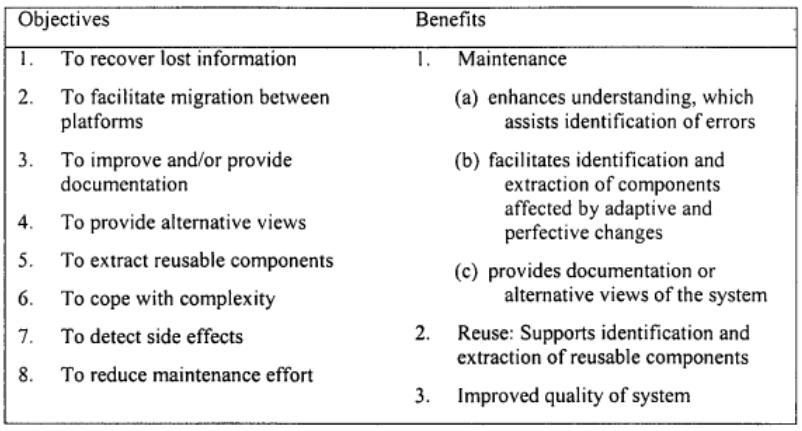
Abstraction is achieved by highlighting the important features of the subject system and ignoring the irrelevant ones. There are three types of abstraction that can be performed on software systems: function, data and process abstraction.

1. **Function Abstraction (Procedural Abstraction):**
   means eliciting functions from the target system - those aspects which operate on data objects and produce the corresponding output.
   Functions are often characterised by an input-output relation; a function takes x as input and produces f(x) as output. During the abstraction process, we are interested in what the function does and not how it operates.
2. **Data Abstraction:**
   means eliciting from the target system data objects as well as the functions that operate on them. The main focus here is on the data objects.
3. **Process Abstraction:**
   This is the abstracting from the target system of the exact order in which operations are performed. There are two classes of process that can be abstracted:
   i. Concurrent processes communicate via shared data that is stored in a designated memory space.
   ii. Distributed processes usually communicate through 'message passing' and have no shared data area.

Goal 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 to recover lost information, to facilitate migration between platforms, to improve and/or provide new documentation, to extract reusable components, to reduce maintenance effort, to cope with complexity, to detect side effects, to assist migration to a CASE environment, and to develop similar or competitive products.

Table 7.1 Summary of objectives and benefits of reverse engineering



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	
5. To extract reusable components	(c) provides documentation or alternative views of the system
6. To cope with complexity	
7. To detect side effects	2. Reuse: Supports identification and extraction of reusable components
8. To reduce maintenance effort	3. Improved quality of system

Levels Of Reverse Engineering

Reverse engineering involves performing one or more of the above types of abstraction, in a bottom-up and incremental manner. It entails detecting low-level implementation constructs and replacing them with their high-level counterparts. The process eventually results in an incremental formation of an overall architecture of the program.

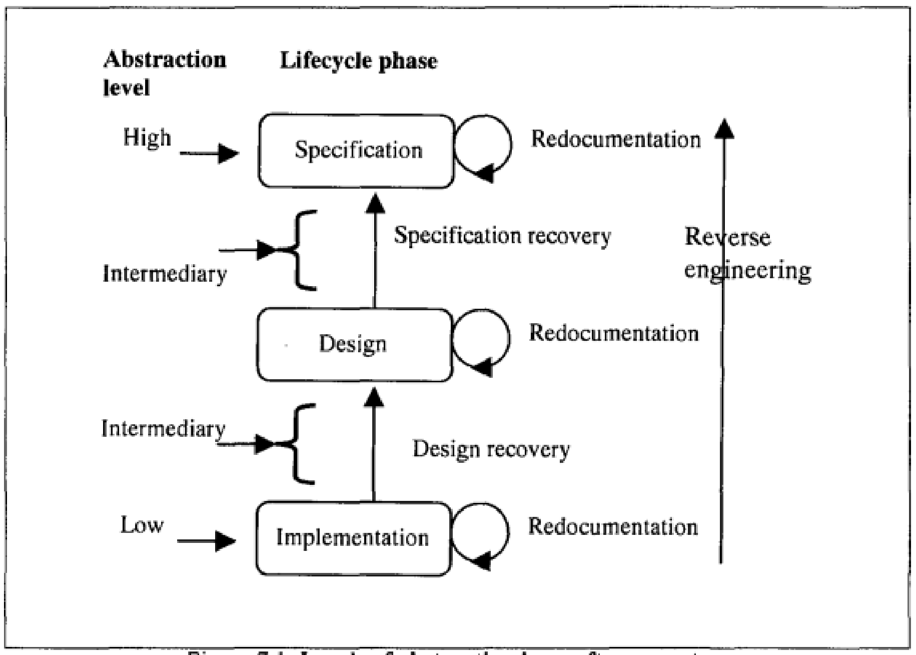


Figure 7.1 **Levels of abstraction in a software system**

1. Redocumentation:

Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level. The goals of this process are threefold.

i. To create alternative views of the system so as to enhance understanding, for example the generation of a hierarchical data flow or control flow diagram from source code.
ii. To improve current documentation. Ideally, such documentation should have been produced during the development of the system and updated as the system changed. This, unfortunately, is not usually the case.
iii. To generate documentation for a newly modified program. This is aimed at facilitating future maintenance work on the system - preventive maintenance.

2. Design Recovery:

Design recovery entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains.

3. Specification Recovery:

In some situations, reverse engineering that only leads to the recovery of the design of the system may not be of much use to an organisation or a software engineer. A typical example is where there is a paradigm shift and the design of the new paradigm has little or nothing in common with the design of the original paradigm, for instance moving from structured programming to object-oriented programming. In this case, an appropriate approach is to obtain the original specification of the system through specification recovery. This involves identifying, abstracting and representing meaningful higher levels of abstractions beyond those obtained simply by inspecting the design or source code of the software system.

Conditions for reverse engineering:

The motives for reverse engineering are usually commercial.

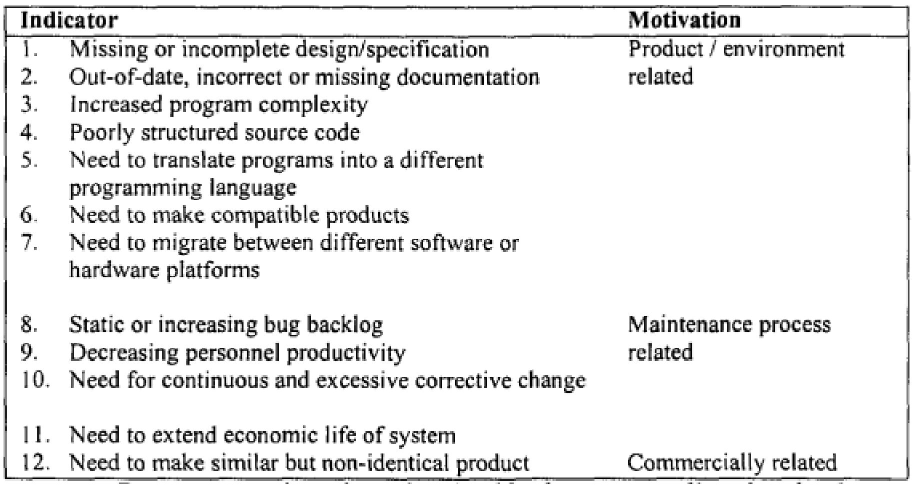


Table 7.2 Factors that motivate the application of reverse engineering

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

Reverse engineering in itself does not directly lead to modification of a system. It simply enables an understanding of a system by representing it at an equivalent or higher abstraction level. The desired change can then be effected by one or more of the following supporting techniques.

Supporting Techniques

1. **Forward engineering**
2. **Restructuring**
   This involves transforming a system from one representational form to another at the same relative level of abstraction without a change in its functionality or semantics.
   Various types of restructuring:
   Control-flow-driven restructuring: This involves the imposition of a clear control structure within the source code and can be either intermodular or intramodular in nature.
   Efficiency-driven restructuring: This involves restructuring a function or algorithm to make it more efficient.
   Adaption-driven restructuring: This involves changing the coding style in order to adapt the program to a new programming language or new operating environment, for instance changing an imperative program in Pascal into a functional program in Lisp.
3. **Reengineering**
   This is the process of examining and altering a target system to implement a desired modification.
   Reengineering consists of two steps.
   i. reverse engineering is applied to the target system so as to understand it and represent it in a new form.
   ii. forward engineering is applied, implementing and integrating any new requirements, thereby giving rise to a new and enhanced system.

Benefits of Reverse Engineering

The successful achievement of these objectives translates into a number of benefits in the areas of maintenance and software quality assurance. The output of reverse engineering can also be beneficial to software reuse - reapplication of existing software code or design.

1. **Maintenance:**

   The ability to use reverse engineering tools to recapture design history and provide documentation facilitates understanding of a system. Considering the time devoted to program understanding, reverse engineering tools offer real scope for reducing maintenance costs.

2. **Software Reuse:**
   The software components that result from a reverse engineering process can be reused. Quite often these components need to be modified in one way or another before they can be reused.

Current Problems:

Reverse engineering promises to be particularly useful in addressing the problems of understanding legacy systems. However, a number of problem areas that still need to be addressed:

The Automation Problem:
It is not yet feasible to automate fully at a very high level. Technology is not yet mature enough to provide the level of automation that software engineers would like. Complete automation may never be feasible because the process of understanding a system - in which reverse engineering plays a part - requires the use of domain-specific information. This may always be reliant upon domain experts.

The Naming Problem:
Naming would still pose a problem. Take as an example the source code for a binary sort algorithm. Extracting the specification is one thing, but automatically naming it (with a meaningful name such as BinarySort as opposed to the less meaningful identifier, p3, say) is quite another.

#Extra
Boldyreff and Zhang have done some work to address this problem. Their approach, called the transformational approach, involves functional decomposition of the program. Code segments are transformed into recursive procedures. The programmer then names and comments each procedure interactively. A collection of these comments then forms a higher-level abstraction of the program.

Next Section: Reuse and Reusability

Software reuse involves the redeployment of software products from one system to maintain or develop another system to reduce effort and to improve quality and productivity. Reusability is the ease with which this goal can be achieved.

Objectives and benefits of Reuse:

3 Objectives:

1. **To increase Productivity:**
   By reusing product, process and personnel knowledge to implement changes rather than writing code from scratch, the software engineer's productivity can be greatly increased because of the reduction in the time and effort that would have been spent on specification, design, implementation and testing the changes. Reuse can, accordingly, reduce the time and costs required to maintain software products.
2. **To increase Quality:**
   Since reusable programs have usually been well tested and already shown to satisfy the desired requirements, they tend to have fewer residual errors. This leads to greater reliability and robustness. It is this feature which makes software reuse attractive to software engineers interested in improving (or at least maintaining) the quality of software products.
3. **To facilitate code transportation:**
   The aim of code transportation is to produce code that can easily be transported across machines or software environments with little or no modification. This excludes activities which are aimed at adapting the same product to changes in its software or hardware operating environment. Producing machine independent components can be achieved using inhouse, national or international standards. Portability reduces the time and resource required to adapt the components to a different machine or software environment.

Benefits of Reuse:

1. **Reduction in maintenance time and effort:**
   Owing to the generality, manageable size and consistency in style of reusable components, it is much easier to read, understand and modify them when effecting a software change, there is also a reduction in the learning time as a result of the increasing familiarity with the reused code that the user gains with time.
2. **To improve maintainability:**
   Reusable components tend to exhibit such characteristics as generality, high cohesion and low coupling, consistency of programming style, modularity and standards.

Approaches to Reuse:

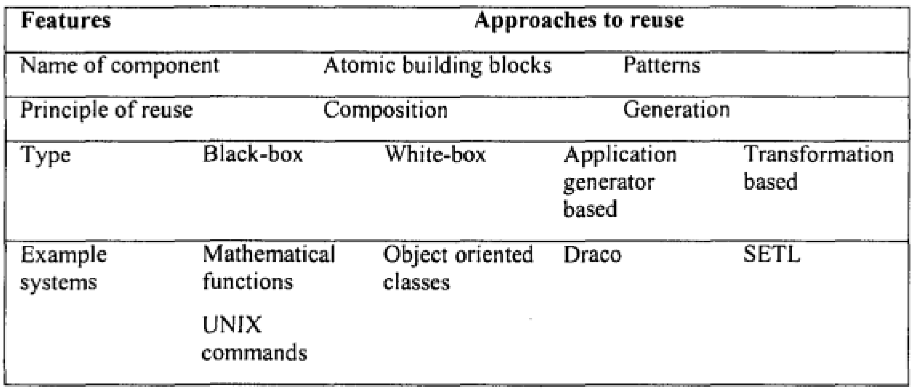


Table 8.1 Approaches to reuse

Features	Approaches to reuse			
Name of component	Atomic building blocks		Patterns	
Principle of reuse	Composition		Generation	
Type	Black-box	White-box	Application generator based	Transformation based
Example systems	Mathematical functions UNIX commands	Object oriented classes	Draco	SETL

1. **Composition Based Reuse:**
   The components being reused are atomic building blocks that are assembled to compose the target system. The components retain their basic characteristics even after they have been reused. Examples of such building blocks are program modules, routines, functions and objects.
   Example: Unix PIPE (|)
2. **Generation Based Reuse:**
   The reusable components are active entities that are used to generate the target system. Here, the reused component is the program that generates the target product. Unlike the composition approach, the output generated does not necessarily bear any resemblance to the generator program. Examples are application generators, transformation-based systems and language-based systems.

Domain Analysis

There are two categories of component that can be reused:

1. **Horizontal reuse** is reuse of components that can be used in a wide variety of domains, for example algorithms and data structures.
2. **Vertical reuse** is reuse of components that are targeted at applications within a given problem area. Due to the application domain-oriented nature of vertical reuse, there is often a need to identify common problems within the domain and attempt to produce 'standard' solutions to these problems. This can be achieved through domain analysis.

Domain analysis is a process by which information used in developing and maintaining software systems is identified, captured, and organised with the purpose of making it reusable when maintaining existing systems. This is achieved by studying the needs and requirements of a collection of applications from a given domain. The objects and operations pertinent to this domain are identified and described.

The constraints on the interactions between these objects and the operations are also described. These descriptions are then implemented as code modules in a given programming language ready to be reused.

Advantages of Domain Analysis

The repository of information produced serves, as an invaluable asset to an organisation. The information can be used for training other personnel about the development and maintenance of software systems belonging to the domain in question.

One of the problems with the data processing industry is the high turnover of personnel - especially in maintenance departments - thus depriving organisations of the valuable expertise gained from previous projects. With domain analysis, the impact of such turnover can be minimised.

Disadvantages of Domain Analysis:

It requires a substantial upfront investment. This can be a risky venture for the organisation because there is no absolute guarantee that the results of the domain analysis will justify its cost.

It is a long-term investment whose benefit will not be realised until the organisation observes some increase in productivity and a reduction in the cost of maintenance because of reuse.

Components Engineering (Read More from Book if Needed)

The composition-based approach to reuse involves composing a new system partly from existing components. There are two main ways in which these components can be obtained. The first is through a process known as design for reuse. This involves developing components with the intention of using them on more than one occasion. The second way is through reverse engineering.

Reuse Process Model

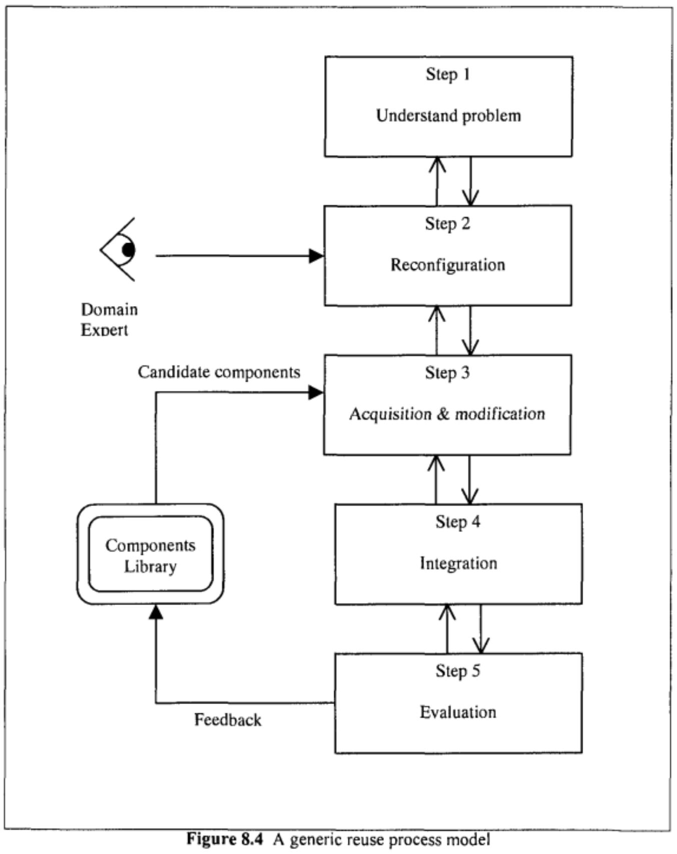


Figure 8.4 A generic reuse process model

Step 1: This step involves understanding the problem to be solved and then identifying a solution structure based on predefined components available.

Step 2: The solution structure is then reconfigured in order to maximise the potential of reuse at the current and the next phase. This means involving the domain experts of the next phase who will study the proposed solution of the current phase and identify reusable components available at the next phase. Several techniques can be used to identify reusable components. The criteria for selection are based on factors such as functionality, logic and preand post-conditions.

Step 3: The major task at this stage is preparing the reusable components identified in the solution structure in readiness for integration. This involves acquiring reusable components, modifying and/or instantiating them for the problem being solved. For those components that cannot be acquired or that are uneconomic to adapt, new ones are developed.

Step 4: The main aim at this stage is integrating the completed components into the product(s) required for the next phase of the software life cycle.

Step 5: In this step, the experience from the preceding steps is used to evaluate the reusability prospects of two categories of component. The first category is those components that need to be developed for the sub-problems for which no reusable components exist. The second category is those components that have been obtained from the adaptation of predefined components. The result of this evaluation exercise is then used to update the current library of reusable components.

Advantage of this model:

It takes a multi-project view of development and maintenance whereby products from one project are expected to be used for other projects.

Factors That Impact upon Reuse:

1. **Technical Factors:**
   i. **Programming Languages**
   The use of several programming languages to develop software within a company hinders attempts to develop reusable components. One way to address this problem is to choose a programming language and mandate its use for projects within an organisation.
   ii. **Representation of information**
   Another impediment to reuse is the representation of design information in a form that does not promote reuse. The reuse of design can be automated.
   one way to achieve this is by representing knowledge about the implementation structures in a form that allows a designer to separate individual design factors.
   iii. **Reuse Library**
   For ease of reuse, large libraries of reusable components must be populated and easily accessible. There is a need for adequate retrieval techniques for managing the libraries. This is an expensive and time-consuming venture upon which some companies are not willing to embark.
   iv. **Reuse-Maintenance Vicious Cycle**
   While applying reuse techniques to alleviate maintenance, it is essential to populate the component library with as many reusable fragments as possible. However, as the library gets bigger, managing it can present another maintenance problem. This can lead to a 'reuse maintenance vicious cycle'. One way to minimise the effect of this is to use good design principles when designing the components as well as the library in which they will be stored.
2. **Non-Technical Factors**
   i. **Education**
   It is sometimes the case that managers, who can play a significant role in reuse programmes, lack an adequate level of software engineering education. This may impede their ability to recognise the potential financial and productivity benefits of reuse.

ii. **Project Co-ordination**
In some companies there is little or no co-ordination between projects. This leads to duplication. Putting in place a reuse programme would facilitate the reuse of knowledge gained and should yield cost and productivity benefits. One way of doing this is for the company to take a multi-project view of development and maintenance. This entails treating the deliverables - artefacts and other forms of knowledge - from every project as potential input to other projects.

iii. **Commercial Interest**
One way to encourage software reuse is to give software components a wide circulation, for instance making them available in the public domain. However, a product that is worth reusing by other individuals or companies may well be marketed commercially for profit. This can prevent authors of software from putting products in the public domain for reuse by others, especially where they would be reused for commercial gain.

Next Section: Maintenance Measures:

Objectives of Software Measurement

Software measurement can be undertaken for several reasons, notably for evaluation, control, assessment, improvement and prediction.

1. **Evaluation:**
   There is a need for maintainers to evaluate different methods, program libraries and tools before arriving at a decision as to which is best suited to a given task.
2. **Control:**
   There is a need to control the process of software change to ensure that change requests are dealt with promptly and within budget.
3. **Assessment:**
   In order to control a process or product, it is important to be able to assess or to characterise it first.
4. **Improvement:**
   There is a need to improve various characteristics of the software system or process such as quality and productivity. It is difficult to assess and monitor such improvements without an objective means of measuring the characteristics.
5. **Prediction:**
   There is a need to make predictions about various aspects of the software product, process and cost.

Examples Of measures:

There are several measures that maintainers may need in order do their job.

1. **Size:**
   One of the commonest ways of measuring the size of a program is by counting the number of lines of code. Moller and Paulish define lines of code (LOC) as "the count of program lines of code excluding comment or blank lines". This measure is usually expressed in thousands of lines of code (KLOC). During maintenance, the focus is on the 'delta' lines of code: the number of lines of code that have been added or modified during a maintenance process.
   PROS
   The advantage of this measure is that it is easy to determine and also correlates strongly with other measures such as effort and error density.
   CONS
   It has, nonetheless, been criticised. There are no standards for LOC measurement and it is dependent on the programming language in question. Also it is too simplistic and does not reflect cost or productivity. Despite these criticisms, this measure is still widely used.

2. Complexity:

Zuse defines it as "the difficulty of maintaining, changing and understanding programs". One of the major problems that software maintainers face is dealing with the increasing complexity of the source code that they must modify.
Program complexity embraces several notions such as program structure, semantic content, control flow, data flow and algorithmic complexity. As such, it can be argued that computing a single complexity value is misleading; other vital information is hidden in this value. However, there is sometimes a need to compute a single value for complexity which in turn is used as an indicator for other attributes such as understandability, maintainability and the effort required for implementation and testing.
The more complex a program is, the more likely it is for the maintainer to make an error when implementing a change, The higher the complexity value, the more difficult it is to understand the program, hence making it less maintainable. Based on the argument that inherently complex programs require more time to understand and modify than simple programs, complexity can be used to estimate the effort required to make a change to a program.

Popular ways to code complexity measures:

1. McCabe's Cyclomatic Complexity

McCabe views a program as a directed graph in which lines of program statements are represented by nodes and the flow of control between the statements is represented by the edges.

$$v(F) = e - n + 2$$

where n = total number of nodes; e = total number of edges or arcs; and v(F) is the cyclomatic number.
This measure is used as an indicator of the psychological complexity of a program. During maintenance, a program with a very high cyclomatic number (usually above 10) is very complex.

PROS

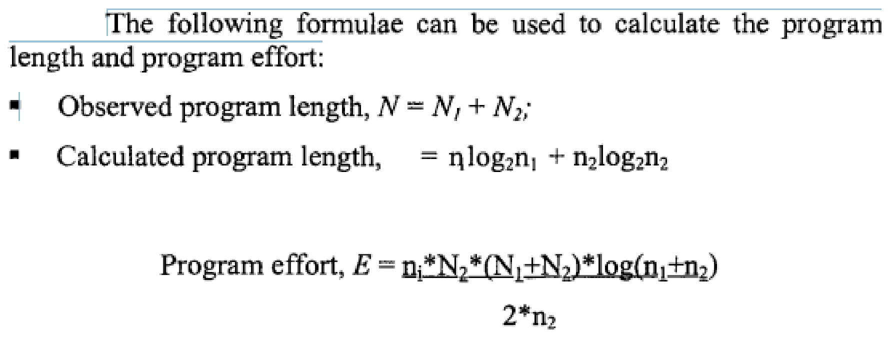
It helps to identify highly complex programs that may need to be modified to reduce complexity.
The cyclomatic number can be used as an estimate of the amount of time required to understand and modify a program.
The flow graph generated can be used to identify the possible test paths during testing.

CONS

It takes no account of the complexity of the conditions in a program, for example multiple use of Boolean expressions, and over-use of flags.
In its original form, it failed to take account of the degree of nesting in a program. As such, two programs may have been equally complex based on cyclomatic number whereas in fact, one had a greater level of nesting than the other. There have been several improvements to take into consideration the level of nesting.

2. Halstead's Measures

Halstead proposed several equations to calculate program attributes such as program length, volume and level, potential volume, language level clarity, implementation time and error rates. Here we shall concentrate on those measures which impact on complexity: program length and program effort. The measures for these attributes can be computed from four basic counts:
$n1$ = number of unique operators used
$n2$ = number of unique operands used
$N1$ = total number of operators used
$N2$ = total number of operands used
Operators include arithmetic operators (for example, *, /, + and -), keywords (for example, PROCEDURE, WHILE, REPEAT and DO), logical operators (for example, greater than, equal to and less than), and delimiters.



The following formulae can be used to calculate the program length and program effort:

- Observed program length, $N = N_1 + N_2$;
- Calculated program length, $= n_1\log_2 n_1 + n_2\log_2 n_2$

$$\text{Program effort, } E = \frac{n_1 * N_2 * (N_1+N_2) * \log(n_1+n_2)}{2 * n_2}$$

PROS

They are easy to calculate and do not require an in-depth analysis of programming features and control flow.
The measures can be applied to any language but yet are programming language sensitive.
There exists empirical evidence from both industry and academia that these measures can be used as good predictors of programming effort and number of bugs in a program.

CONS

The counting rules involved in the design of the measures were not fully defined and it is not clear what should be counted.
There was failure to consider declarations and input/output statements as a unique operator for each unique label.
The measures are code-based; it is assumed that 'software = programs'. Although this may be true for some systems, others, especially those developed using modern software engineering techniques and tools, are likely to have automatic documentation support. The measures fail to capture the contribution that this documentation makes to the programming effort or program understanding.

3. Quality:

A quality maintenance process is one which enables the maintainer to implement the desired change. Different measures can be used to characterise product and process quality

i. Product quality:

One way of measuring the quality of a software system is by keeping track of the number of change requests received from the users after the system becomes operational. This measure is computed by "dividing the number of unique change requests made by customers for the first year of field use of a given release, by the number of thousand lines of code for that release". This measure only includes requests which pertain to faults detected by customers. The measure excludes feature enhancement change requests which are not contained in the software requirements specification. The number of change requests from users can serve as an indicator of customer satisfaction. It can also serve as an indicator of the amount of maintenance effort that may be required for the system. The other measure of product quality is the number of faults that are detected after the software system becomes operational, usually after the first year of shipment. The same type of fault pointed out by more than one user is counted as a single fault. The number of users reporting the same fault may be used as a measure of the significance of the fault and therefore the priority that should be attached to fixing it.

ii. Process Quality:

This describes the degree to which the maintenance process being used is assisting personnel in satisfying change requests. Two measures of process quality are schedule and productivity.
The schedule is calculated as "the difference between the planned and actual work time to achieve the milestone of first customer delivery, divided by the planned work time". This measure is expressed as a percentage. A negative number signifies a slip and a positive number signifies early delivery. The productivity is computed by dividing the number of lines of code that have been added or modified by the effort in staff days required to make the addition or

modification. Effort is the total time from analysing the change requests to a successful implementation of the change.

4. **Understandability:**

   Program understandability is the ease with which the program can be understood, that is, the ability to determine what a program does and how it works by reading its source code and accompanying documentation. This attribute depends not just on the program source code, but also on other external factors such as the available documentation, the maintenance process and maintenance personnel. Some of the measures that can be used as an estimate of understandability are complexity, quality of documentation, consistency, and conciseness.

   Understandability usually has an inverse relation to complexity; as the complexity of a program increases, the understandability tends to decrease. From this perspective, understandability can be computed indirectly from McCabe's cyclomatic complexity and Halstead's program effort measure. Understandability can also be estimated from subjective ratings of the quality of documentation, consistency of programming style and the conciseness of the program text.

5. **Maintainability:**

   Software maintainability is "the ease with which the software can be understood, corrected, adapted, and/or enhanced". Maintainability is an external attribute since its computation requires knowledge from the software product as well as external factors such as the maintenance process and the maintenance personnel. An example of a maintainability measure that depends on an external factor is the Mean Time to Repair (MTTR): the mean time required to effect a change. Depending on the circumstances, the calculation of MTTR may require information on the problem recognition time, administrative delay time, maintenance tools collection time, problem analysis time, change specification time and change time. Maintainability can also be perceived as an internal attribute if it is derived solely from the software system. Several internal attributes of program source code can impact on maintainability, for example modularity. Thus, measures for these other internal attributes would need to be obtained to compute maintainability. Unfortunately, there is yet to be an exact model for determining maintainability. At present, measures such as complexity and readability are used as indicators or predictors of maintainability.

6. **Cost Estimation:**

   The cost of a maintenance project is the resources - personnel, machines, time and money - expended on effecting change. One way of estimating the cost of a maintenance task is from historical data collected for a similar task. The major difficulty with this approach to cost estimation is that there may be new variables impacting upon the current task which were not considered in the past. However, the more that is collected the more accurate will be the estimate.
   A second way of estimating cost is through mathematical models. One of these was Boehm's COCOMO model adapted for maintenance. The updated COCOMO II model, instead of being based on a single process model such as the waterfall model, has been interpreted to cover the waterfall model, MBASE/RUP (Model Based Architecting and Software Engineering / Rational Unified Process), and incremental development. According to Boehm, the cost of maintenance is affected by attributes of factors called cost drivers. Examples of cost drivers are database size, program complexity, use of modern programming practices and applications experience of the maintenance personnel.
   A third measure of cost is time in person-months required to modify a program.

Guidelines for Selecting Maintenance Measures

Some guidelines that can be used in selecting suitable maintenance measures. These guidelines include well defined objectives, fitness for purpose, ease of use, low implementation cost and sensitivity.

1. **Clearly defined objectives:** Prior to deciding on the use of a measurement for maintenance-related purposes, it is essential to define clearly and unambiguously what objectives need to be achieved. These objectives will determine the measures to be used and the data to be collected.

2. **Personnel involvement:** The purpose of measurement in an organisation needs to be made clear to those involved in the programme. And the measures obtained should be used for that purpose and nothing else. For instance, it needs to be made clear whether the measurement is to improve productivity, to set and monitor targets, etc. Without such clear expression of the purpose of measurement, personnel may feel that the measures will be used for punitive purposes, and this can impede the programme.
3. **Ease of use:** The measures that are finally selected to be used need to be easy to use, take not too much time to administer, be unobtrusive, and possibly subject to automation. As indicated earlier in this chapter, one of the reasons why source code-based measures are very popular is because they can be easily automated and collected in an unobtrusive way.