UNIT-V

CHAPTER-I

COMPUTER-AIDED SOFTWARE ENGINEERING (CASE)

computer aided software engineering (CASE) and how use of CASE tools help to improve software development effort and maintenance effort. Software is becoming the costliest component in any computer installation. Even though hardware prices keep dropping like never and falling below even the most optimistic expectations, software prices are becoming costlier due to increased manpower costs.

CASE AND ITS SCOPE

We first need to define what is a CASE tool and what is a CASE environment. A CASE tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool can mean any tool used to automate some activity associated with software development.

Many CASE tools are now available. Some of these tools assist in phase-related tasks such as specification, structured analysis, design, coding, testing, etc. and others to non-phase activities such as project management and configuration management.

The primary objectives in using any CASE tool are:

- To increase productivity.
- To help produce better quality software at lower cost.

CASE ENVIRONMENT

Although individual CASE tools are useful, the true power of a tool set can be realised only when these set of tools are integrated into a common framework or environment. If the different CASE tools are not integrated, then the data generated by one tool would have to input to the other tools.

This may also involve format conversions as the tools developed by different vendors are likely to use different formats. This results in additional effort of exporting data from one tool and importing to another. Also, many tools do not allow exporting data and maintain the data in proprietary formats.

CASE tools are characterised by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software.

This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus, a CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast a CASE environment, a *programming environment* is an integrated collection of tools to support only the coding phase of software development.

The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger. The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements are highlighted.

Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc. A schematic representation of a CASE environment is shown in Figure 12.1.

The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc. All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

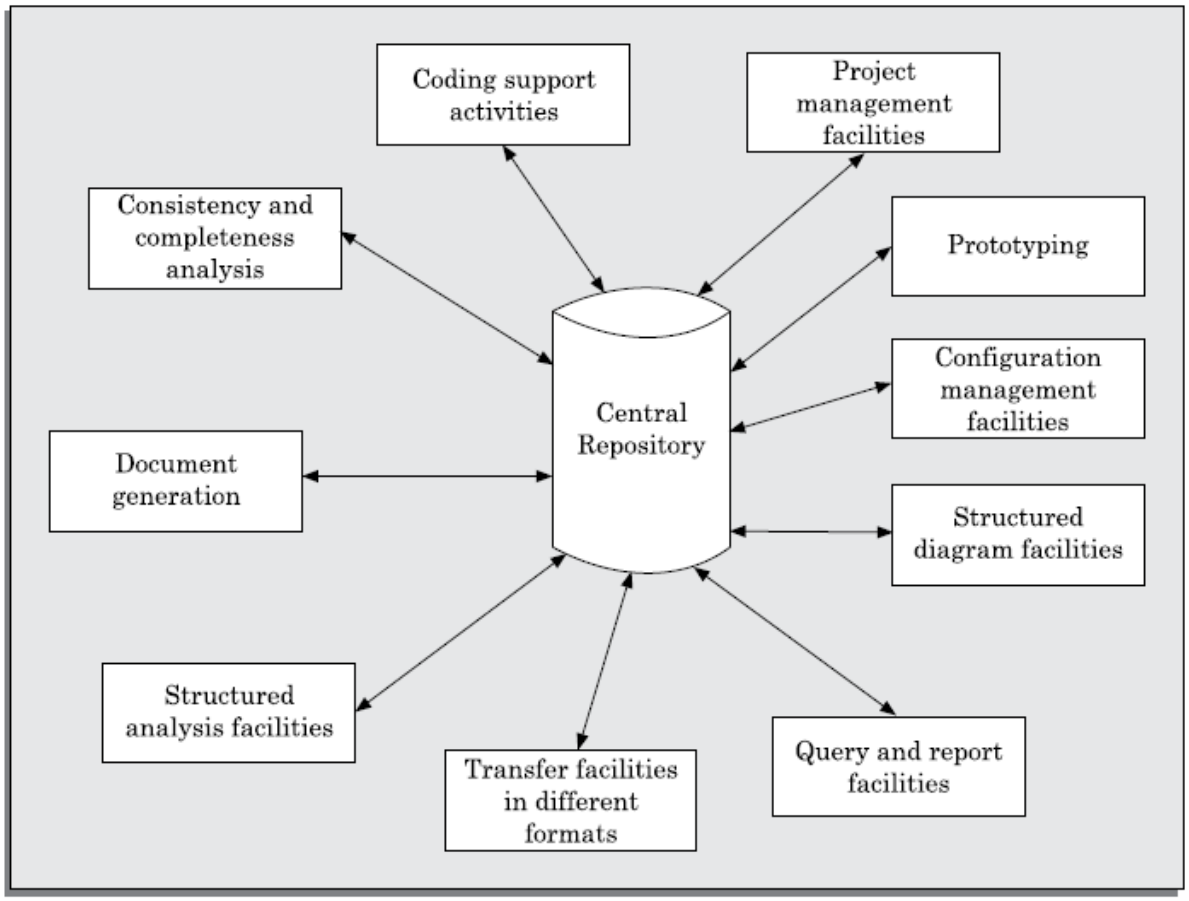


FIGURE 12.1 A CASE environment.

Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools.

Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases. Different studies carry out to measure the impact of CASE, put the effort reduction between 30 per cent and 40 per cent.
- Use of CASE tools leads to considerable improvements in quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development, and the chances of human error is considerably reduced.
- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundance in the stored data is reduced, and therefore, chances of inconsistent documentation are reduced to a great extent.
- CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs, but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

CASE SUPPORT IN SOFTWARE LIFE CYCLE

Let us examine the various types of support that CASE provides during the different phases of a software life cycle. CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases. Some of the possible support that CASE tools usually provide in the software development life cycle are discussed below.

Prototyping Support

We have already seen that prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on.

The prototyping CASE tool's requirements are as follows:

- Define user interaction.
- Define the system control flow.
- Store and retrieve data required by the system.
- Incorporate some processing logic.

There are several standalone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype. A good prototyping tool should support the following features:

 Since one of the main uses of a prototyping CASE tool is *graphical user interface* (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.

 It should integrate with the data dictionary of a CASE environment.

 If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.

 The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.

 The run time system of prototype should support mock up run of the actual system and management of the input and output data.

Structured Analysis and Design

Several diagramming techniques are used for structured analysis and structured design. CASE tool should support one or more of the structured analysis and design technique. The CASE tool should support effortlessly drawing analysis and design diagrams. The CASE tool should support drawing fairly complex diagrams and preferably through a hierarchy of levels. It should provide easy navigation through different levels and through design and analysis.

The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Wherever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there is heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

Code Generation

As far as code generation is concerned, the general expectation from a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic support expected from a CASE tool during code generation phase are the following:

 The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.

 The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular programming languages.

 It should generate database tables for relational database management systems.

 The tool should generate code for user interface from prototype definition for X window and MS window-based applications.

Test CASE Generator

The CASE tool for test case generation should have the following features:

 It should support both design and requirement testing.

 It should generate test set reports in ASCII format which can be directly imported into the test plan document.

OTHER CHARACTERISTICS OF CASE TOOLS

The characteristics listed in this section are not central to the functionality of CASE tools but significantly enhance the effectivity and usefulness of CASE tools

Hardware and Environmental Requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, we have to emphasise on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and we choose this for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients which run different modules access data dictionary through this server.

The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g., a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow backup/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

Documentation Support

The deliverable documents should be organized graphically and should be able tom incorporate text and diagrams from the central repository. This helps in producing up-to date documentation. The CASE tool should integrate with one or more of the commercially available desk-top publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

Project Management

It should support collecting, storing, and analysing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

External Interface

The tool should allow exchange of information for reusability of design. The information which is to be exported by the tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

Reverse Engineering Support

The tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

Data Dictionary Interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

Tutorial and Help

The application of CASE tool and thereby its success depends on the users' capability to effectively use all the features supported. Therefore, for the uninitiated users, a tutorial is very important. The tutorial should not be limited to teaching the user interface part only, but should comprehensively cover the following points:

- The tutorial should cover all techniques and facilities through logically classified sections.
- The tutorial should be supported by proper documentation.

TOWARDS SECOND GENERATION CASE TOOL

An important feature of the second-generation CASE tool is the direct support of any adapted methodology. This would necessitate the function of a CASE administrator for every organisation, who can tailor the CASE tool to a particular methodology. In addition, we may look forward to the following features in the second-generation CASE tool:

Intelligent diagramming support: The fact that diagramming techniques are useful for system analysis and design is well established. The future CASE tools would provide help to aesthetically and automatically layout the diagrams.

Integration with implementation environment: The CASE tools should provide integration between design and implementation.

Data dictionary standards: The user should be allowed to integrate many development tools into one environment. It is highly unlikely that any one vendor will be able to deliver a total solution. Moreover, a preferred tool would require tuning up for a particular system. Thus, the user would act as a system integrator. This is possible only if some standard on data dictionary emerges.

Customisation support: The user should be allowed to define new types of objects and connections. This facility may be used to build some special methodologies. Ideally it should be possible to specify the rules of a methodology to a *rule engine* for carrying out the necessary consistency checks.

ARCHITECTURE OF A CASE ENVIRONMENT

The architecture of a typical modern CASE environment is shown diagrammatically in Figure 12.2. The important components of a modern CASE environment are user interface, tool set, *object management system* (OMS), and a repository. We have already seen the characteristics of the tool set. Let us examine the other components of a CASE environment.

User interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

Object management system and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities into the underlying storage management system (repository).

The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entities and relation types with perhaps a few instances of each.

Thus, the object management system takes care of appropriately mapping these entities into the underlying storage management system.

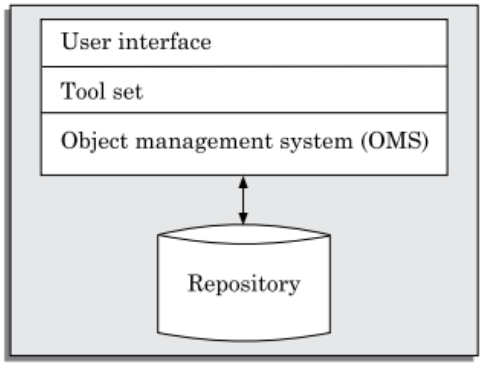


FIGURE 12.2 Architecture of a modern CASE environment.

UNIT-V

CHAPTER-II

SOFTWARE MAINTENANCE

Software maintenance denotes any changes made to a software product after it has been delivered to the customer.

Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use. For example, a car tyre wears out due to use.

On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

CHARACTERISTICS OF SOFTWARE MAINTENANCE

Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product *per se*, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.

When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface.

For instance, a software product may need to be maintained when the operating system changes or the software needs to run over hand held devices. Thus, every software product continues to evolve after its development through maintenance efforts.

Types of Software Maintenance

There are three types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary to overcome the failures observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support any new features that the users may want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Characteristics of Software Evolution

Lehman and Belady studied the characteristics of evolution of several software products [1980]. They expressed their observations in the form of laws. Their important laws are presented in the following subsection. But a word of caution here is that these are generalisations and may not be applicable to specific cases. Further, most of their observations

concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law:

A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts.

This law clearly shows that every product irrespective of how well designed must undergo maintenance.

In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law:

The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that usually maintenance activities result in patch work. It is rarely the case that members of the original development team are part of the maintenance team.

The maintenance team, therefore, often has a partial and inadequate understanding of the architecture, design, and code of the software. Therefore, any modifications tend to be ugly and more complex than they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law:

Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore, this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

Special Problems Associated with Software Maintenance

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:

Software maintenance work in organisations is mostly carried out using *ad hoc* techniques.

The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organisation often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development

work. During maintenance it is necessary to thoroughly understand someone else′s work, and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products. Though the word legacy implies ″aged″ software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain.

The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product.

Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1.

A program can be reformatted using any of the several available Pretty Printer programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important that meaningful variable names is the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible.

Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

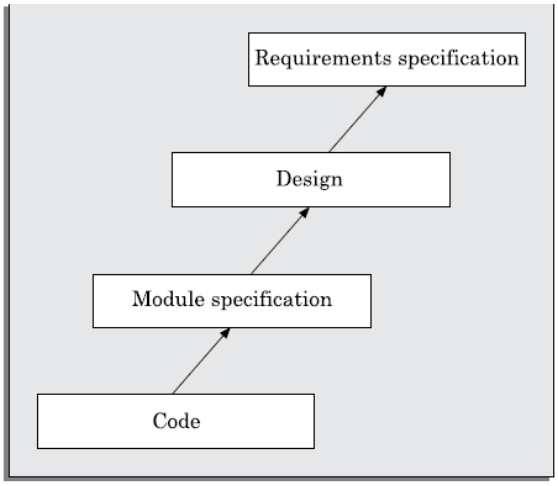


FIGURE 13.1 A process model for reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

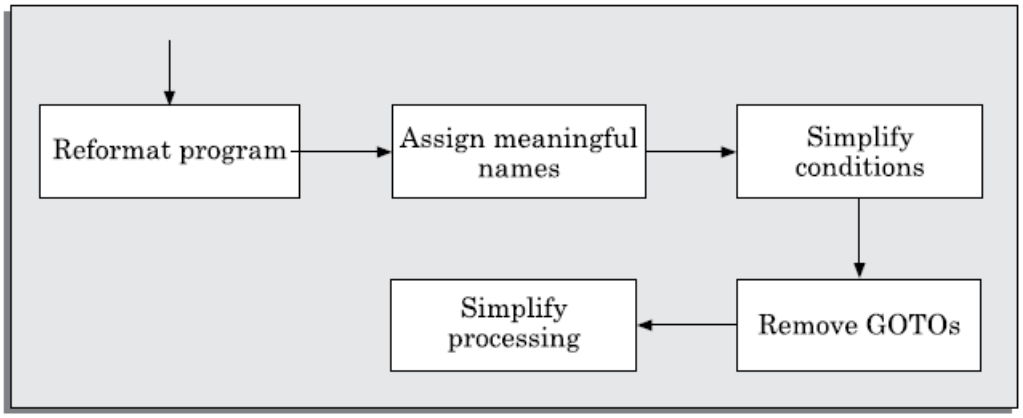


FIGURE 13.2 Cosmetic changes carried out before reverse engineering.

SOFTWARE MAINTENANCE PROCESS MODELS

Before discussing process models for software maintenance, we need to analyse various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as:

(i) the extent of modification to the product required,
(ii) (ii) the resources available to the maintenance team,
(iii) (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.), (iii) the expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope (activities required) for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed

First model

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Figure 13.3. In this approach, the project starts by gathering the requirements for changes.

The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code.

The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re-engineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.

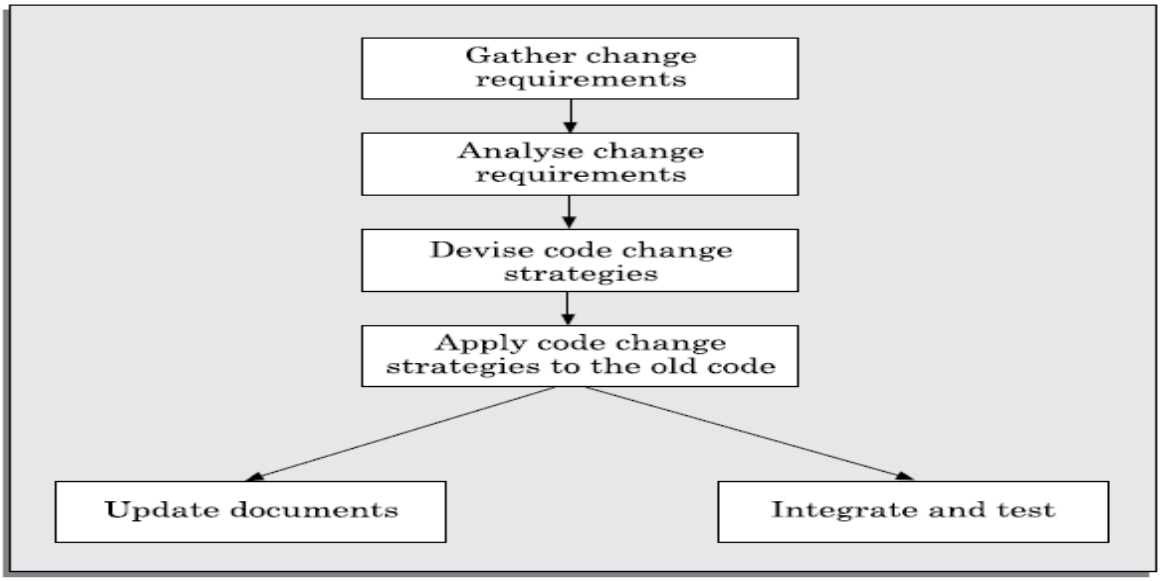


FIGURE 13.3 Maintenance process model 1.

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as *software re-engineering*. This process model is depicted in Figure 13.4.

The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications. The module

specifications are then analysed to produce the design. The design is analysed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification.

At this point a forward engineering is carried out to produce the new code. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more effi cient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15 per cent (see Figure 13.5).

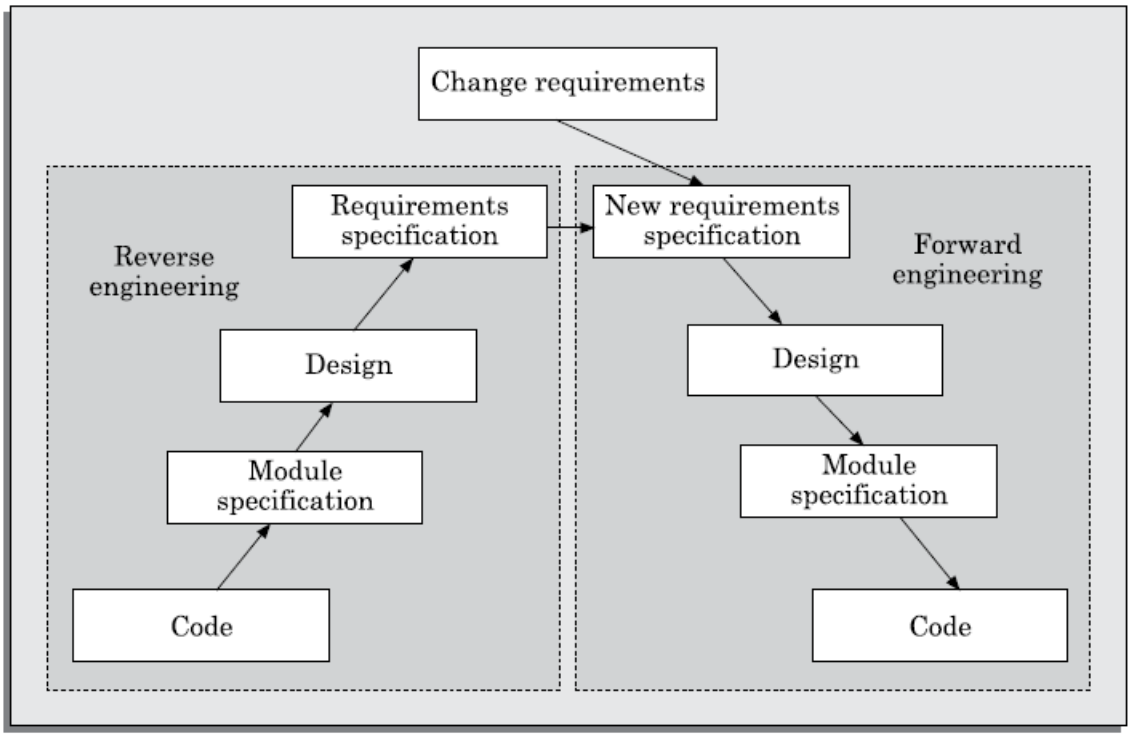
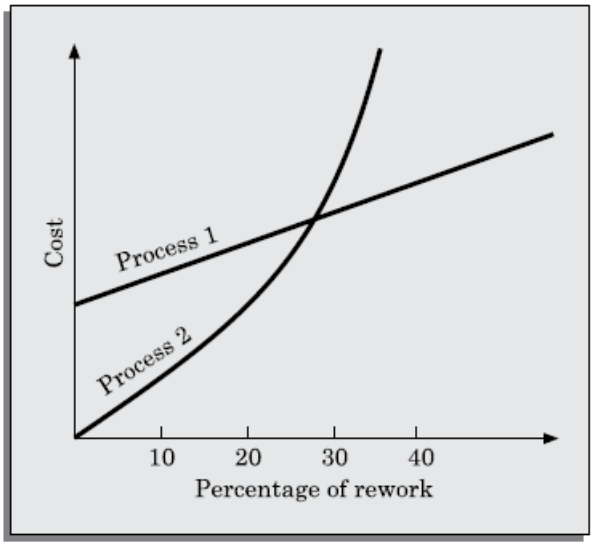


FIGURE 13.4 Maintenance process model 2.

Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

 Re-engineering might be preferable for products which exhibit a high failure rate.

 Re-engineering might also be preferable for legacy products having poor design and code structure.



Empirical estimation of maintenance cost versus percentage rework.

ESTIMATION OF MAINTENANCE COST

We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the *annual change traffic* (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, KLOC added is the total kilo lines of source code added during maintenance. KLOC deleted is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and code deleted.

The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = ACT × Development cost

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.