UNIT 12 ADVANCED TOPICS IN SOFTWARE ENGINEERING

Structure **Page Nos.**

12.0 INTRODUCTION

In the first few units we have studied the various methods for conventional software engineering. Software engineering methods can be categorised on a “formality spectrum.” The analysis and design methods discussed previously would be placed at the informal to moderately rigorous end of the spectrum. In these methods, a combination of diagrams, text, tables, and simple notation is used to create analysis and design models.

At the other end of the formality spectrum are formal methods, which are completely mathematical in form. A specification and design are described using a formal syntax and semantics that specify system function and behaviour. Other specialised models being the Cleanroom software engineering, component-based software engineering, re-engineering and reverse engineering.

Harlan Mills has developed the Cleanroom software engineering methodology, which is an approach for integrating formal methods into the lifecycle. The Cleanroom approach combines formal methods and structured programming with Statistical Process Control (SPC), the spiral lifecycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in non-software fields.

In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems). Software engineering has focused on custom development of components. To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process. There are various ways to achieve this and one of the models is the Component-based software engineering.

Let us study the various specialised software engineering models like the formal method, Cleanroom engineering, component-based engineering, re-engineering and reverse engineering in this unit.

12.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the pitfalls of the conventional software engineering;
- know various advanced methods of software engineering – their advantages and drawbacks;
- describe the methodology of formal methods and Cleanroom software engineering, and
- discuss the Re-engineering and Reverse engineering process.

12.2 EVOLUTION OF FORMAL METHODS

The seventies witnessed the structured programming revolution. After much debate, software engineers became convinced that better programs result from following certain precepts in program design. Recent imperative programming languages provide constructs supporting structured programming. Achieving this consensus did not end debate over programming methodology. On the contrary, a period of continuous change began, with views on the best methods of software development mutating frequently. Top-down development, modular decomposition, data abstraction, and, most recently, object oriented design are some of the jargon terms that have arisen to describe new concepts for developing large software systems. Both researchers and practitioners have found it difficult to keep up with this onslaught of new methodologies.

There is a set of core ideas that lies at the base of these changes. Formal methods have provided a unifying philosophy and central foundation upon which these methodologies have been built. Those who understand this underlying philosophy can more easily adopt these and other programming techniques.

One way to improve the quality of software is to change the way in which software is documented: at the design stage, during development, and after release. Existing methods of documentation offer large amounts of text, pictures, and diagrams, but these are often imprecise and ambiguous. Important information is hidden amongst irrelevant details, and design flaws are discovered too late, making them expensive or impossible to correct.

There is an alternative. *Formal* methods, based upon elementary mathematics, can be used to produce precise, unambiguous documentation, in which information is structured and presented at an appropriate level of abstraction. This documentation can be used to support the design process, and as a guide to subsequent development, testing, and maintenance.

It seems likely that the use of formal methods will become standard practice in software engineering. The mathematical basis is different from that of civil or mechanical engineering, but it has the same purpose: to add precision, to aid understanding, and to reason about the properties of a design. Whatever the discipline, the use of mathematics can be expensive, but it is our experience that it can actually reduce costs.

Existing applications of formal methods include: the use of the probability theory in performance modeling; the use of context-free grammars in compiler design; the use of the relational calculus in database theory. The formal method described in this book has been used in the specification and design of large software systems. It is intended for the description of state and state-based properties, and includes a theory of refinement that allows mathematics to be used at every stage of program development. Let us see the details of the Formal methods in the next section.

12.3 USE OF MATHEMATICS IN SOFTWARE DEVELOPMENT

Mathematics supports *abstraction* and therefore is a powerful medium for modeling. Because they are exact, mathematical specifications are *unambiguous* and can be validated to uncover *contradictions* and *incompleteness*. It allows a developer to *validate* a specification for functionality. It is possible to demonstrate that a design matches a specification, and that some program code is a correct reflection of a design.

How is the mathematics of formal languages applied in software development? What engineering issues have been addressed by their application? Formal methods are of global concern in software engineering. They are directly applicable during the requirements, design, and coding phases and have important consequences for testing and maintenance. They have influenced the development and standardisation of many programming languages, the programmer's most basic tool. They are important in ongoing research that may change standard practice, particularly in the areas of specifications and design methodology. They are entwined with lifecycle models that may provide an alternative to the waterfall model, namely rapid prototyping, the Cleanroom variant on the spiral model, and "transformational" paradigms. The concept of formalism in formal methods is borrowed from certain trends in 19th and 20th century mathematics. Formal methods are merely an adoption of the axiomatic method, as developed by these trends in mathematics, for software engineering. Mastery of formal methods in software requires an understanding of this mathematics background. Mathematical topics of interest include formal logic, both the propositional calculus and predicate logic, set theory, formal languages, and automata such as finite state machines.

12.4 FORMAL METHODS

A **formal method** *in software development is a method that provides a formal language for describing a software artifact (e.g., specifications, designs, source code) such that formal proofs are possible, in principle, about properties of the artifact so expressed.*

The definition is based on two essential components. First, formal methods involve the essential use of a formal language. A formal language is a set of strings over some well-defined alphabet. Rules are given for distinguishing those strings, defined over the alphabet, that belong to the language from strings that do not. Secondly, formal methods in software support formal reasoning about formulae in the language. These methods of reasoning are exemplified by formal proofs. A proof begins with a set of axioms, which are to be taken as statements postulated to be true. Inference rules state that if certain formula, known as premises, are derivable from the axioms, then another formula, known as the consequent, is also derivable. A set of inference rules must be given in each formal method. A proof consists of a sequence of well-defined formulae in the language in which each formula is either an axiom or derivable by an inference rule from previous formulae in the sequence. The last formula in the sequence is said to be proven.

The major concerns of the formal methods are as follows:

i) The correctness of the problem
   - producing software that is "correct" is famously difficult;
   - by using rigorous mathematical techniques, it may be possible to make probably correct software.

ii) Programs are mathematical objects
   - they are expressed in a **formal** language;
   - they have a **formal** semantics;
   - programs can be treated as mathematical theories.

Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in the language. Since defining what a system should do, and understanding the implications of these decisions, are the most troublesome problems in software engineering, this use of formal methods has major benefits. In fact, practitioners of formal methods frequently use formal methods solely for recording precise specifications, not for formal verifications.

Formal methods were originally developed to support verifications, but higher interest currently exists in specification methods. Several methods and languages can be used for specifying the functionality of computer systems. No single language, of those now available, is equally appropriate for all methods, application domains, and aspects of a system. Thus, users of formal specification techniques need to understand the strength and weaknesses of different methods and languages before deciding on which to adopt. The distinction between a specification method and a language is fundamental. A method states what a specification must say. A language determines in detail how the concepts in a specification can be expressed. Some languages support more than one method, while most methods can be used in several specification languages.

Some of the most well-known formal methods consist of or include specification languages for recording a system's functionality. These methods include:

- Z (pronounced "Zed")
- Communicating Sequential Processes (CSP)
- Vienna Development Method (VDM)
- Larch
- Formal Development Methodology (FDM)

12.4.1 What can be Formally Specified?

Formal methods can include graphical languages. Data Flow Diagrams (DFDs) are the most well-known graphical technique for specifying the function of a system. DFDs can be considered a semi-formal method, and researchers have explored techniques for treating DFDs in a completely formal manner. Petri nets provide another well-known graphical technique, often used in distributed systems. Petri nets are a fully formal technique.

Finally, finite state machines are often presented in tabular form. This does not decrease the formalism in the use of finite state machines. So the definition of formal methods provided earlier is quite encompassing.

Software engineers produce models and define the properties of systems at several levels of abstraction. Formal methods can be employed at each level. A specification should describe what a system should do, but not how it is done. More details are provided in designs, with the source code providing the most detailed model.

For example, Abstract Data Types (ADTs) frequently are employed at intermediate levels of abstraction. ADTs, being mathematical entities, are perfect candidates for formal treatment and are often so treated in the literature.

Formal methods are not confined to the software components of large systems. System engineers frequently use formal methods. Hardware engineers also use formal methods, such as VHSIC Hardware Description Language (VHDL) descriptions, to model integrated circuits before fabricating them. The following section lists the goals of the formal specification.

12.4.2 Goals of Formal Specification

Once a formal description of a system has been produced, what can be done with it? Usable formal methods provide a variety of techniques for reasoning about specifications and drawing implications. The completeness and consistency of a specification can be explored. Does a description imply a system should be in several states simultaneously? Do all legal inputs yield one and only one output? What surprising results, perhaps unintended, can be produced by a system? Formal methods provide reasoning techniques to explore these questions. Do lower level descriptions of a system properly implement higher level descriptions? Formal methods support formal verification, the construction of formal proofs that an implementation satisfies a specification. The possibility of constructing such formal proofs was historically the principal driver in the development of formal methods. Prominent technology for formal verification includes E Dijkstra's "weakest precondition" calculus and Harlan Mills' "functional correctness" approach which we are not going to discuss here. The following are the goals of the formal specification:

- **Removal of ambiguity:** The formal syntax of a specification language enables requirements or design to be interpreted in only one way, eliminating the ambiguity that often occurs when using natural language. Removes ambiguity and encourages greater rigour in the early stages of the software engineering process.
- **Consistency:** Facts stated in one place should not be contradicted in another. This is ensured by mathematically proving that initial facts can be formally mapped (by inference) into later statements later in the specification.
- **Completeness:** This is difficult, even when using formal methods. Some aspects of a system may be left undefined by mistake or by intention. It is impossible to consider every operational scenario in a large, complex system.

12.5 APPLICATION AREAS

Formal methods can be used to specify aspects of a system other than functionality. Software safety and security are other areas where formal methods are sometimes applied in practice. The benefits of proving that unsafe states will not arise, or security will not be violated, can justify the cost of complete formal verifications of the relevant portions of a software system. Formal methods can deal with many other areas of concern to software engineers, but have not been much used, other than in research organisations, for dealing with issues unrelated to functionality, safety, and security. Areas in which researchers are exploring formal methods include fault tolerance, response time, space efficiency, reliability, human factors, and software structure dependencies. The following are the Formal specifications application areas:

- safety critical systems
- security systems
- The definition of standards
- Hardware development
- Operating systems
- Transaction processing systems
- Anything that is hard, complex, or critical.

Let us see the drawbacks of the formal specifications in the next section.

12.6 LIMITATIONS OF FORMAL SPECIFICATION USING FORMAL METHODS

Some problems exist, however formal specification focuses primarily on function and data. Timing, control, and behavioural aspects of a problem are more difficult to represent. In addition, there are elements of a problem (e.g., HCI, the human-machine interface) that are better specified using graphical techniques. Finally, a specification using formal methods is more difficult to learn than other analysis methods. For this reason, it is likely that formal mathematical specification techniques will be incorporated into a future generation of CASE tools. When this occurs, mathematically-based specification may be adopted by a wider segment of the software engineering community. The following are the major drawbacks:

Cost: It can be almost as costly to do a full formal specification as to do all the coding. The solution for this is, don't formally specify everything, just the subsystems you need to (hard, complex, or critical).

Complexity: Not everybody can read formal specifications (especially customers and users). The solution is to provide a very good and detailed documentation. No one wants to read pure formal specifications–formal specifications should always be interspersed with natural language (e.g. English) explanation.

Deficiencies of Less Formal Approaches: The methods of structured and object-oriented design discussed previously make heavy use of natural language and a variety of graphical notations. Although careful application of these analysis and design methods can and does lead to high–quality software, sloppiness in the application of these methods can create a variety of problems which are:

- **Contradictions**: statements that are at variance with one another.
- **Ambiguities** : statements that can be interpreted in a number of ways
- **Vagueness** : statements that lack precision, and contribute little information.
- **Incomplete statements** : a description is not functionally complete.
- **Mixed levels of abstraction** : statements with high and low levels of detail are interspersed, which can make it difficult to comprehend the functional architecture.

This method gained popularity among the software developers who must build safety-critical software like aircraft avionics and medical devices. But the applicability in the business environment has not clicked.

Let us study the Cleanroom Software Engineering in the next section.

☞ **Check Your Progress 1**

1) What are Formal methods?

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the main parts of the Formal methods?

……………………………………………………………………………………

……………………………………………………………………………………

12.7 CLEANROOM SOFTWARE ENGINEERING

Cleanroom software engineering is an engineering and managerial process for the development of high-quality software with certified reliability. Cleanroom was originally developed by Dr. Harlan Mills. The name "Cleanroom" was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication. It reflects the same emphasis on defect prevention rather than defect removal, as well as Certification of reliability for the intended environment of use. It combines many of the formal methods and software quality methods we have studied so far. The focus of Cleanroom involves moving from traditional software development practices to rigorous, engineering-based practices.

Cleanroom software engineering yields software that:

- Is correct by mathematically sound design
- Is certified by statistically-valid testing
- Reduces the cycle time results from an incremental development strategy and the avoidance of rework
- Is well-documented
- Detects errors as early as possible and reduces the cost of errors during development and the incidence of failures during operation; thus the overall life cycle cost of software developed under Cleanroom can be expected to be far lower than the industry average.

The following principles are the foundation for the Cleanroom-based software development:

Incremental development under statistical quality control (SQC): Incremental development as practiced in Cleanroom provides a basis for statistical quality control of the development process. Each increment is a complete iteration of the process, and measures of performance in each increment (feedback) are compared with pre-established standards to determine whether or not the process is "in control." If quality standards are not met, testing of the increment ceases and developers return to the design stage.

Software development based on mathematical principles: In Cleanroom software engineering development, the key principle is that, a computer program is an expression of a mathematical function. The Box Structure Method is used for specification and design, and functional verification is used to confirm that the design is a correct implementation of the specification. Therefore, the specification must define that function before design and functional verification can begin. Verification of program correctness is performed through team review based on correctness questions. There is no execution of code prior to its submission for independent testing.

Software testing based on statistical principles: In Cleanroom, software testing is viewed as a statistical experiment. A representative subset of all possible uses of the software is generated, and performance of the subset is used as a basis for conclusions about general operational performance. In other words, a "sample" is used to draw conclusions about a "population." Under a testing protocol that is faithful to the principles of applied statistics, a scientifically valid statement can be made about the expected operational performance of the software in terms of reliability and confidence.

12.8 CONVENTIONAL SOFTWARE ENGINEERING MODELS Vs. CLEANROOM SOFTWARE ENGINEERING MODEL

Cleanroom has been documented to be very effective in new development and reengineering like whole systems or major subunits. The following points highlight the areas where Cleanroom affects or differs from more conventional software development engineering practice:

Small team of software engineers: A Cleanroom project team is small, typically six to eight qualified professionals, and works in an organised way to ensure the intellectual control of work in progress. The composition of the Cleanroom Process Teams are as follows:

- **Specification team** develops and maintains the system specification
- **Development team** develops and verifies software
- **Certification team** develops set of statistical tests to exercise software after development and reliability growth models used to assess reliability.

Time allocation across the life cycle phases: Because one of the major objectives of Cleanroom is to prevent errors from occurring, the amount of time spent in the design phase of a Cleanroom development is likely to be greater than the amount of time traditionally devoted to design.

Existing organisational practices: Cleanroom does not preclude using other software engineering techniques as long as they are not incompatible with Cleanroom principles. Implementation of the Cleanroom method can take place in a gradual manner. A pilot project can provide an opportunity to "tune" Cleanroom practices to the local culture, and the new practices can be introduced as pilot results to build confidence among software staff.

The following are some of the examples of the projects implemented using the Cleanroom engineering method:

- IBM COBOL/SF product
- Ericsson OS-32 operating system project
- USAF Space Command and Control Architectural Infrastructure (SCAI) STARS Demonstration Project at Peterson Air Force Base in Colorado Springs, Colorado.
- US Army Cleanroom project in the Tank-automotive and Armaments Command at the U.S. Army Picatinny Arsenal.

12.9 CLEANROOM SOFTWARE ENGINEERING PRINCIPLES, STRATEGY AND PROCESS OVERVIEW

This software development is based on mathematical principles. It follows the box principle for specification and design. Formal verification is used to confirm correctness of implementation of specification. Program correctness is verified by team reviews using questionnaires. Testing is based on statistical principles. The test cases are randomly generated from the usage model –failure data is interpreted using statistical models. The following is the phase-wise strategy followed for Cleanroom software development.

Increment planning: The project plan is built around the incremental strategy.

Requirements gathering: Customer requirements are elicited and refined for each increment using traditional methods.

Box structure specification: Box structures isolate and separate the definition of behaviour, data, and procedures at each level of refinement.

Formal design: Specifications (black-boxes) are iteratively refined to become architectural designs (state-boxes) and component-level designs (clear boxes).

Correctness verification: Correctness questions are asked and answered, formal mathematical verification is used as required.

Code generation, inspection, verification: Box structures are translated into program language; inspections are used to ensure conformance of code and boxes, as well as syntactic correctness of code; followed by correctness verification of the code.

Statistical test planning: A suite of test cases is created to match the probability distribution of the projected product usage pattern.

Statistical use testing: A statistical sample of all possible test cases is used rather than exhaustive testing.

Certification: Once verification, inspection, and usage testing are complete and all defects removed, the increment is certified as ready for integration.

Figure 12.1 depicts the Cleanroom software engineering development overview:

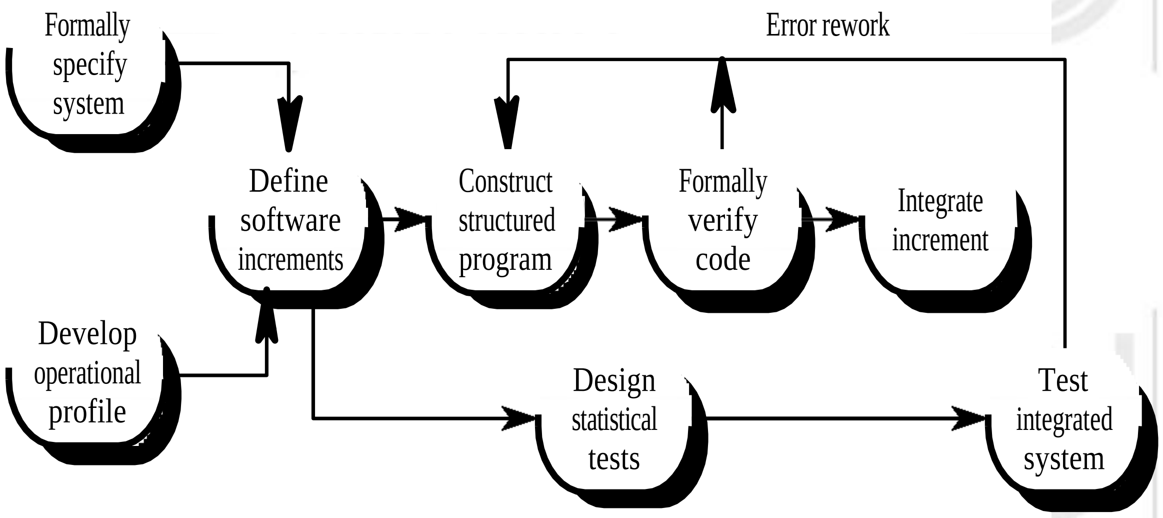

Figure 12.1: Overview of Cleanroom Software Engineering Development

12.10 LIMITATIONS OF CLEANROOM ENGINEERING

The following are some of the limitations of the Cleanroom software engineering method.

- Some people believe cleanroom techniques are too theoretical, too mathematical, and too radical for use in real software development.
- Relies on correctness verification and statistical quality control rather than unit testing (a major departure from traditional software development).
- Organisations operating at the ad hoc level of the Capability Maturity Model do not make rigorous use of the defined processes needed in all phases of the software lifecycle.

12.11 SIMILARITIES AND DIFFERENCES BETWEEN CLEANROOM AND OO PARADIGM

The following are the similarities and the differences between the Cleanroom software engineering development and OO software engineering paradigm.

Similarities

- Lifecycle - both rely on incremental development
- Usage - cleanroom usage model similar to OO use case
- State Machine Use - cleanroom state box and OO transition diagram
- Reuse - explicit objective in both process models

Key Differences

- Cleanroom relies on decomposition and OO relies on composition
- Cleanroom relies on formal methods while OO allows informal use case definition and testing
- OO inheritance hierarchy is a design resource whereas cleanroom usage hierarchy is system itself

- OO practitioners prefer graphical representations while cleanroom practitioners prefer tabular representations
- Tool support is good for most OO processes, but usually tool support is only found in cleanroom testing, not design.

Let us study the component based software engineering in the next section.

☞ **Check Your Progress 2**

1) What is Cleanroom Engineering and what are its objectives?

………………………………………………………………………………………

………………………………………………………………………………………

2) What is Statistical Process Control?

………………………………………………………………………………………

………………………………………………………………………………………

3) What are the benefits of Cleanroom software engineering?

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

4) What are the limitations of Cleanroom software engineering?

……………………………………………………………………………………….

……………………………………………………………………………………….

12.12 SOFTWARE REUSE AND ITS TYPES

In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems). Software engineering has focused on custom development of components. To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process. The following are the types of software reuse:

- **Application System Reuse**
  - reusing an entire application by incorporation of one application inside another (COTS reuse)
  - development of application families (e.g., MS Office)
- **Component Reuse**
  - components (e.g., subsystems or single objects) of one application reused in another application
- **Function Reuse**
  - reusing software components that implement a single well-defined function.

Let us take into consideration the ***Component Reuse*** type and the development process of the components in the following sections:

"Suppose you have purchased a Television. Each component has been designed to fit a specific architectural style – connections are standardised, a communication protocol has been pre-established. Assembly is easy because you don't have to build the system from hundreds of discrete parts. Component-based software engineering (CBSE) strives to achieve the same thing. A set of pre-built, standardised software components are made available to fit a specific architectural style for some application domain. The application is then assembled using these components, rather than the "discrete parts" of a conventional programming language. Components may be constructed with the explicit goal to allow them to be generalised and reused. Also, component reusability should strive to reflect stable domain abstractions, hide state representations, be independent (low coupling) and propagate exceptions via the component interface".

12.13 WHY COMPONENT BASED SOFTWARE ENGINEERING?

The goal of component-based software engineering is to increase the productivity, quality, and decrease time-to-market in software development. One important paradigm shift implied here is to build software systems from standard components rather than "reinventing the wheel" each time. This requires thinking in terms of system families rather than single systems.

CBSE uses Software Engineering principles to apply the same idea as OOP to the whole process of designing and constructing software systems. It focuses on *reusing* and *adapting* existing components, as opposed to just coding in a particular style. CBSE encourages the composition of software systems, as opposed to programming them. Two important aspects of the question are:

- Firstly, several underlying technologies have matured that permit building components and assembling applications from sets of those components. Object oriented and Component technology are examples – especially standards such as CORBA.
- Secondly, the business and organisational context within which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires technology that will allow easy additions.

Software Component and its types

A software component is a nontrivial, independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. In some ways, this is similar to the description of an object in OOP. Components have an interface. They employ inheritance rules. Components can be characterised based on their use in the CBSE process. One category is *commercial off-the-shelf* (or COTS) components. These are components that can be purchased, pre-built, with the disadvantage that there is (usually) no source code available, and so the definition of the use of the component given by the manufacturer, and the services which it offers, must be relied upon or thoroughly tested, as they may or may not be accurate. The advantage, though, is that these types of components should (in theory) be more robust and adaptable, as they have been used and tested (and reused and retested) in many different applications.

In addition to COTS components, the CBSE process yields:

- qualified components
- adapted components
- assembled components
- updated components.

12.14 COMPONENT BASED SOFTWARE ENGINEERING (CBSE) PROCESS

CBSE is in many ways similar to conventional or object-oriented software engineering. A software team establishes requirements for the system to be built using conventional requirements elicitation techniques. An architectural design is established. Here though, the process differs. Rather than a more detailed design task, the team now examines the requirements to determine what subset is directly amenable to *composition*, rather than *construction*.

For each requirement, we should question:

- Whether any commercial off-the-shelf (COTS) components available to implement the requirement?
- Whether internally developed reusable components available to implement the requirement?
- Whether interfaces for available components compatible within the architecture of the system to be built?

The team will attempt to modify or remove those system requirements that cannot be implemented with COTS or in-house components. This is not always possible or practical, but reduces the overall system cost and improves the time to market of the software system. It can often be useful to prioritise the requirements, or else developers may find themselves coding components that are no longer necessary as they have been eliminated from the requirements already.

The CBSE process identifies not only candidate components but also qualifies each component's interface, adapts components to remove architectural mismatches, assembles components into selected architectural style, and updates components as requirements for the system change.

Two processes occur in parallel during the CBSE process. These are:

- Domain Engineering
- Component Based Development.

12.14.1 Domain Engineering

This aims to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. An application domain is like a *product family* – applications with similar functionality or intended functionality. The goal is to establish a mechanism by which software engineers can share these components in order to reuse them in future systems. As defined by Paul Clements, *domain engineering is about finding commonalities among systems to identify components that can be applied to many systems and to identify program families that are positioned to take fullest advantage of those components.*

Some examples of application domains are as follows:

- Air traffic control systems
- Defence systems
- Financial market systems.

Domain engineering begins by identifying the domain to be analysed. This is achieved by examining existing applications and by consulting experts of the type of application you are aiming to develop. A *domain model* is then realised by identifying operations and relationships that recur across the domain and therefore being candidates for reuse. This model guides the software engineer to identify and categorise components, which will be subsequently implemented.

One particular approach to domain engineering is *Structural Modeling*. This is a pattern-based approach that works under the assumption that every application domain has repeating patterns. These patterns may be in function, data, or behaviour that have reuse potential. This is similar to the pattern-based approach in OOP, where a particular style of coding is reapplied in different contexts. An example of *Structural Modeling* is in Aircraft avionics: The systems differ greatly in specifics, but all modern software in this domain has the same structural model.

12.14.2 Component Based Development

There are three stages in this process. These are:

- Qualification,
- Adaptation (also known as wrapping),
- Composition (all in the context of components).

Component qualification: examines reusable components. These are identified by characteristics in their interfaces, i.e., the services provided, and the means by which consumers access these services. This does not always provide the whole picture of whether a component will fit the requirements and the architectural style. This is a process of discovery by the Software Engineer. This ensures a candidate component will perform the function required, and whether it is compatible or adaptable to the architectural style of the system. The three important characteristics looked at are *performance, reliability and usability*.

Component Adaptation is required because very rarely will components integrate immediately with the system. Depending on the component type(e.g., COTS or in-house), different strategies are used for adaptation (also known as *wrapping*). The most common approaches are:

- **White box wrapping:** The implementation of the component is directly modified in order to resolve any incompatibilities. This is, obviously, only possible if the source code is available for a component, which is extremely unlikely in the case of COTS.
- **Grey box wrapping:** This relies on the component library providing a component extension language or API that enables conflicts to be removed or masked.
- **Black box wrapping:** This is the most common case, where access to source code is not available, and the only way the component can be adapted is by pre / post-processing at the interface level.

It is the job of the software engineer to determine whether the effort required to wrap a component adequately is justified, or whether it would be "cheaper" (from a software engineering perspective) to engineer a custom component which removes these conflicts. Also, once a component has been adapted it is necessary to check for compatibility for integration and to test for any unexpected behaviour which has emerged due to the modifications made.

Component Composition: integrated the components (whether they are qualified, adapted or engineered) into a working system. This is accomplished by way of an infrastructure which is established to bind the components into an operational system. This infrastructure is usually a library of specialised components itself. It provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

There are many mechanisms for creating an effective infrastructure, in order to achieve component composition, but in particular there is the *Data Exchange Model*, which allows users and applications to interact and transfer data (an example of which is *drag-and-drop* and *cut-and-paste*). These types of mechanisms should be defined for all reusable components. Also important is the *Underlying object model*. This allows the interoperation of components developed in different programming languages that reside on different platforms.

12.15 COMPONENT TECHNOLOGIES AVAILABLE

Common methods of achieving this is by the use of technologies such as *Corba* which allows components to communicate remotely and transparently over a network, *Java*

Beans and Microsoft COM which allows components from different vendors to work together within Windows. Sometimes it is necessary to bridge these technologies (e.g., COM/Corba) to facilitate integration of software developed by different vendors, or for integration with legacy systems.

It must be noted that as the requirements for a system change, components may need to be updated. Usually, some sort of *change control procedure* should be in place to facilitate configuration management, and to ensure updates to the components follow the architectural style of the system being developed.

12.16 CHALLENGES FOR CBSE

CBSE is facing many challenges today, some of these are summarised as follows:

- **Dependable systems and CBSE:** The use of CBD in safety-critical domains, real-time systems, and different process-control systems, in which the reliability requirements are more rigorous, is particularly challenging. A major problem with CBD is the limited possibility of ensuring the quality and other non-functional attributes of the components and thus our inability to guarantee specific system attributes.
- **Tool support:** The purpose of Software Engineering is to provide practical solutions to practical problems, and the existence of appropriate tools is essential for a successful CBSE performance. Development tools, such as Visual Basic, have proved to be extremely successful, but many other tools are yet to appear – component selection and evaluation tools, component repositories and tools for managing the repositories, component test tools, component-based design tools, run-time system analysis tools, component configuration tools, etc. The objective of CBSE is to build systems from components simply and efficiently, and this can only be achieved with extensive tool support.
- **Trusted components:** Because the trend is to deliver components in binary form and the component development process is outside the control of component users, questions related to component trustworthiness become of great importance.
- **Component certification:** One way of classifying components is to certificate them. In spite of the common belief that certification means absolute trustworthiness, it is in fact only gives the results of tests performed and a description of the environment in which the tests were performed. While certification is a standard procedure in many domains, it is not yet established in software in general and especially not for software components.
- **Composition predictability:** Even if we assume that we can specify all the relevant attributes of components, it is not known how these attributes determine the corresponding attributes of systems of which they are composed. The ideal approach, to derive system attributes from component attributes is still a subject of research. A question remains - "Is such derivation at all possible? Or should we not concentrate on the measurement of the attributes of component composites?"
- **Requirements management and component selection:** Requirements management is a complex process. A problem of requirements management is that requirements in general are incomplete, imprecise and contradictory. In an in-house development, the main objective is to implement a system which will satisfy the requirements as far as possible within a specified framework of different constraints. In component-based development, the fundamental approach is the reuse of existing components. The process of engineering requirements is much more complex as the possible candidate components are usually lacking one or more features which meet the system requirements exactly. In addition, even if some components are individually well suited to the

system, it is not necessary that they do not function optimally in combination with others in the system- or perhaps not at all. These constraints may require another approach in requirements engineering – an analysis of the feasibility of requirements in relation to the components available and the consequent modification of requirements. As there are many uncertainties in the process of component selection there is a need for a strategy for managing risks in the components selection and evolution process.

- **Long-term management of component-based systems:** As component-based systems include sub-systems and components with independent lifecycles, the problem of system evolution becomes significantly more complex. CBSE is a new approach and there is little experience as yet of the maintainability of such systems. There is a risk that many such systems will be troublesome to maintain.
- **Development models:** Although existing development models demonstrate powerful technologies, they have many ambiguous characteristics, they are incomplete, and they are difficult to use.
- **Component configurations:** Complex systems may include many components which, in turn, include other components. In many cases compositions of components will be treated as components. As soon as we begin to work with complex structures, the problems involved with structure configuration pop up.

☞ **Check Your Progress 3**

1) What are the benefits of reuse of software?

……………………………………………………………………………………

……………………………………………………………………………………

2) What is Commercial off the Shelf Software (COTS)?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

12.17 REENGINEERING – AN INTRODUCTION

Over the past few years, legacy system reengineering has emerged as a business critical activity. Software technology is evolving by leaps and bounds, and in most cases, legacy software systems need to operate on new computing platforms, be enhanced with new functionality, or be adapted to meet new user requirements. The following are some of the reasons to reengineer the legacy system:

- Allow legacy software to quickly adapt to changing requirements
- Comply to new organisational standards
- Upgrade to newer technologies/platforms/paradigms
- Extend the software's life expectancy
- Identify candidates for reuse
- Improve software maintainability
- Increase productivity per maintenance programmer
- Reduce reliance on programmers who have specialised in a given software system
- Reduce maintenance errors and costs.

Reengineering applies reverse engineering to existing system code to extract design and requirements. Forward engineering is then used to develop the replacement system.

Let us elaborate the definition. Reengineering is the examination, analysis, and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The process typically encompasses a combination of reverse engineering, re-documentation, restructuring, and forward engineering. The goal is to understand the existing software system components (specification, design, implementation) and then to re-do them to improve the system's functionality, performance, or implementation.

The reengineering process is depicted in *Figure 12.2*. Re-engineering starts with the code and comprehensively reverse engineers by increasing the level of abstraction as far as needed toward the conceptual level, rethinking and re-evaluating the engineering and requirements of the current code, then forward engineers using a waterfall software development life-cycle to the target system. In re-engineering, industry reports indicate that approximately 60% of the time is spent on the reverse engineering process, while 40% of the time is spent on forward engineering. Upon completion of the target system, most projects must justify their effort, showing that the necessary functionality has been maintained while quality has improved (implying improved reliability and decreased maintenance costs).

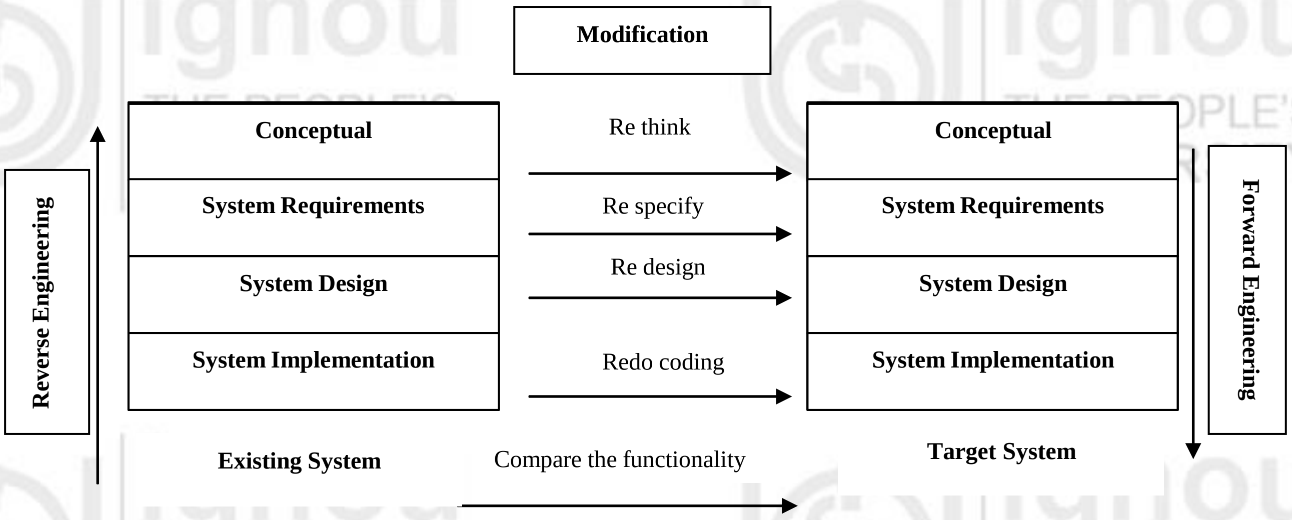

Figure 12.2: Process of Software Reengineering

12.18 OBJECTIVES OF REENGINEERING

The objectives of a specific software reengineering activity are determined by the goals of the clients and users of the system. However, there are two general reengineering objectives as shown below:

- **Improve quality:** Typically, the existing software system is of low quality, due to many modifications. User and system documentation is often out-of-date or no longer in existence. Re-engineering is intended to improve software quality and to produce current documentation. Improved quality is needed to increase reliability, to improve maintainability, to reduce the cost of maintenance, and to prepare for functional enhancement. Object-oriented technology may be applied as a means for improving maintainability and reducing costs.
- **Migration:** Old working software may still meet users' needs, but it may be based on hardware platforms, operating systems, or languages that have become obsolete and thus may need to be re-engineered, transporting the software to a newer platform or language. Migration may involve extensive redesign if the new supporting platforms and operating systems are very different from the original, such as the move from a mainframe to a network-based computing environment.

12.19 SOFTWARE REENGINEERING LIFE CYCLE

In this section, we present an evolutionary view of the software reengineering life-cycle:

Requirements analysis phase: It refers to the identification of concrete reengineering goals for a given software. The specification of the criteria should be specified and illustrated in the new reengineered system (for example, faster performance). Violations that need to be repaired are also identified in this phase.

Model analysis phase: This refers to documenting and understanding the architecture and the functionality of the legacy system being reengineered. In order to understand and to transform a legacy system, it is necessary to capture its design, its architecture and the relationships between different elements of its implementation. As a consequence, a preliminary model is required in order to document the system and the rationale behind its design. This requires reverse engineering the legacy system in order to extract design information from its code.

Source code analysis phase: This phase refers to the identification of the parts of the code that are responsible for violations of requirements originally specified in the system's analysis phase. This task encompasses the design of methods and tools to inspect, measure, rank, and visualize software structures. Detecting error prone code that deviates from its initial requirement specifications requires a way to measure where and by how much these requirements are violated. Problem detection can be based on a static analysis of the legacy system (i.e., analysing its source code or its design structure), but it can also rely on a dynamic usage analysis of the system (i.e., an investigation of how programs behave at run-time).

Remediation phase: It refers to the selection of a target software structure that aims to repair a design or a source code defect with respect to a target quality requirement. Because legacy applications have been evolved in such a way that classes, objects, and methods may heavily depend on each other, a detected problem may have to be decomposed into simpler sub-problems.

Transformation phase: This consists of physically transforming software structures according to the remediation strategies selected previously. This requires methods and tools to manipulate and edit software systems, re-organise and re-compile them automatically, debug and check their consistency, and manage different versions of the software system being reengineered.

Evaluation phase refers to the process of assessing the new system as well as establishing and integrating the revised system throughout the corporate operating environment. This might involve the need for training and possibly the need for adopting a new improved business process model.

Such a reengineering life-cycle yields a reengineering process. First, the source code is represented as an Abstract Syntax Tree. The tree is further decorated with annotations that provide linkage, scope, and type information. Once software artifacts have been understood, classified and stored during the reverse engineering phase, their behaviour can be readily available to the system during the forward engineering phase. Then, the forward engineering phase aims to produce a new version of a legacy system that operates on the target architecture and aims to address specific maintainability or performance enhancements. Finally, we use an iterative procedure to obtain the new migrant source code by selecting and applying a transformation which leads to performance or maintainability enhancements. The transformation is selected from the soft-goal interdependency graphs. The resulting migrant system is then evaluated and the step is repeated until the specific quality requirements are met.

☞ Check Your Progress 4

1) What is Software Reengineering and what are its objectives?

……………………………………………………………………………………

……………………………………………………………………………………

2) What is Reverse Engineering?

……………………………………………………………………………………………

……………………………………………………………………………………………

12.20 SUMMARY

In this unit, we have gone through the Formal Methods, Cleanroom software engineering, Component Based Software Engineering and the Software Reengineering process.

Formal methods promise to yield benefits in quality and productivity. They provide an exciting paradigm for understanding software and its development, as well as a set of techniques for use by software engineers. Formal methods can provide more precise specifications, better internal communication, and an ability to verify designs before executing them during test, higher quality and productivity. To get their full advantages, formal methods should be incorporated into a software organisation's standard procedures. Software development is a social process, and the techniques employed need to support that process. How to fully fit formal methods into the lifecycle is not fully understood. Perhaps there is no universal answer, but only solutions that vary from organisation to organisation.

Harlan Mills has developed the Cleanroom methodology, which is one approach for integrating formal methods into the lifecycle. The Cleanroom approach combines formal methods and structured programming with Statistical Process Control (SPC), the spiral lifecycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in non-software fields.

Component Based Software Engineering introduced major changes into design and development practices, which introduces extra cost. Software engineers need to employ new processes and ways of thinking – this, too, can introduce extra cost in training and education. However, initial studies into the impact of CBSE on product quality, development quality and cost, show an overall gain, and so it seems likely that continuing these practices in the future will improve software development. It is still not clear how exactly CBSE will mature, but it is clear that it aids in the development of today's large-scale systems, and will continue to aid in the development of future systems, and is the perfect platform for addressing the requirements of modern businesses**.**

Software reengineering is a very wide-ranging term that encompasses a great many activities. As the name suggests, software reengineering is applied to existing pieces of software, in an after-the-fact fashion, via the reapplication of an engineering process.

12.21 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Formal methods is that area of computer science that is concerned with the application of mathematical techniques to the design and implementation of computer software.

2) Following are the main parts to formal methods:

   i) **Formal specification:** Using mathematics to specify the desired properties of a computer system.

   ii) **Formal verification:** Using mathematics to prove that a computer system satisfies its specification.

iii) **Automated programming:** Automating the process of program generation.

Check Your Progress 2

1) The Cleanroom methodology is an iterative, life-cycle approach focused on software quality, especially reliability. Begun by Harlan Mills, it combines formal specifications, structured programming, formal verifications, formal inspections, functional testing based on random selection of test data, software reliability measurement, and Statistical Process Control (SPC) in an integrated whole. The Cleanroom approach fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in fields other than software.

2) Statistical Process Control (SPC) is commonly used in manufacturing, involves continuous process monitoring and improvement to reduce the variance of outputs and to ensure that the process remains under control.

3) Benefits of Cleanroom software engineering include significant improvements in *correctness*, *reliability*, and *understandability*. These benefits usually translate into a reduction in field-experienced product failures, reduced cycle time, ease of maintenance, and longer product life.

4) Cleanroom techniques are too theoretical, too mathematical, and too radical for use in real software development. They rely on correctness verification and statistical quality control rather than unit testing (a major departure from traditional software development). Organisations operating at the ad hoc level of the Capability Maturity Model do not make rigorous use of the defined processes needed in all phases of the software life cycle.

Check Your Progress 3

1) Benefits of software reuse are:

- **Increased Reliability**: components already exercised in working systems
- **Reduced Process Risk**: Less uncertainty in development costs
- **Effective Use of components**: Reuse components instead of people
- **Standards Compliance**: Embed standards in reusable components
- **Accelerated Development**: Avoid custom development and speed up delivery.

2) COTS systems usually have a complete applications library and also offers an applications programming interface (API). These are helpful in building large systems by integrating COTS components is a viable development strategy for some types of systems (e.g., E-commerce or video games).

Check Your Progress 4

1) Software reengineering is a process that aims to either

   i) Improve understanding of a piece of software, or
   ii) Prepare for an improvement in the software itself (e.g., increasing its maintainability or reusability).

As the name suggests software reengineering is applied to existing pieces of software, in an after-the-fact fashion, via the reapplication of an engineering process. Software reengineering is a very wide-ranging term that encompasses a great many activities. For example, software reengineering could involve refactoring a piece of software, redocumenting it, reverse engineering it or changing its implementation language.

2) Reverse engineering is a process which is used to improve the understanding of a program. So-called program comprehension is crucial to maintenance. Approximately 50% of the development time of programmers is spent understanding the code that they are working on. Reverse engineering often involves the production of diagrams. It can be used to abstract away irrelevant

implementation and design details to recover the original requirements of the program.

12.22 FURTHER READINGS

1) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.

2) *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

3) *Component Software –Beyond Object-Oriented Programming,1998,* Szyperski C; Addison-Wesley.

4) Software Engineering, Schaum's Outlines, 2003,David Gustafson,Tata

Mc Graw-Hill.

Reference websites

- **http://www.rspa.com**
- **http://www.ieee.org**

12.23 GLOSSARY

Software Reengineering: The examination and alteration of an existing subject system to reconstitute it in a new form. This process encompasses a combination of sub-processes such as reverse engineering, restructuring, redocumentation, forward engineering, and retargeting.

Reverse Engineering: The engineering process of understanding, analysing, and abstracting the system to a new form at a higher abstraction level.

Forward Engineering: Forward engineering is the set of engineering activities that consume the products and artifacts derived from legacy software and new requirements to produce a new target system.

Data Reengineering: Tools that perform all the reengineeering functions associated with source code (reverse engineering, forward engineering, translation, redocumentation, restructuring / normalisation, and retargeting) but act upon data files.

Redocumentation: The process of analysing the system to produce support documentation in various forms including users manuals and reformatting the systems' source code listings.

Restructuring: The engineering process of transforming the system from one representation form to another at the same relative abstraction level, while preserving the subject system's external functional behavior.

Retargeting: The engineering process of transforming and hosting / porting the existing system with a new configuration.

Source Code Translation: Transformation of source code from one language to another or from one version of a language to another version of the same language (e.g., going from COBOL-74 to COBOL-85).

Business Process Reengineering (BPR): The fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service, and speed.