UNIT-5

COMPUTER-AIDED SOFTWARE ENGINEERING (CASE): CASE and its scope, CASE environment, CASE support in the software life cycle, other characteristics of CASE tools, Towards second generation CASE Tool, and Architecture of a CASE Environment.
SOFTWARE MAINTENANCE: Characteristics of software maintenance, Software reverse engineering, Software maintenance process models and Estimation of maintenance cost.
SOFTWARE REUSE: reuse- definition, introduction, reason behind no reuse so far, Basic issues in any reuse program, A reuse approach, and Reuse at organization level.

COMPUTER AIDED SOFTWARE ENGINEERING

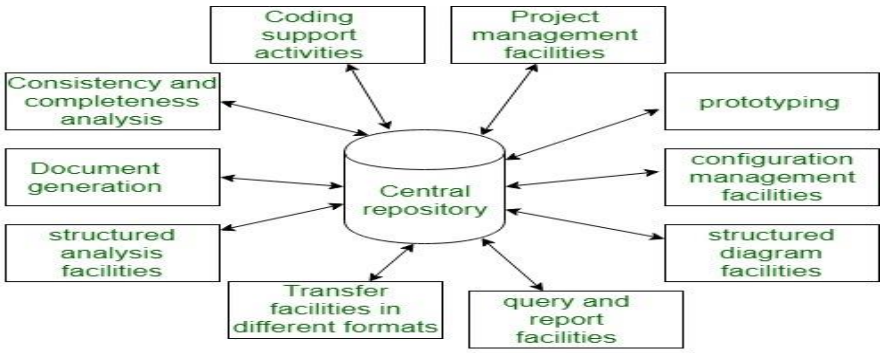
- CASE tools promise effort and cost reduction in software development and maintenance. For software engineers, CASE tools help develop better quality products more efficiently.

1. CASE AND ITS SCOPE :

- A CASE tool can mean any tool used to automate some activity associated with software development.
- 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.
- primary objectives in using any CASE tool are:
  - To increase productivity.
  - To help produce better quality software at lower cost.

2. CASE ENVIRONMENT :

- A CASE environment enhances the power of individual CASE tools by integrating them into a common framework. This integration ensures seamless data sharing between tools, eliminating the need for manual data export/import and format conversions.
- The tools in a CASE environment focus on different stages of the software development life cycle and share information through a central repository, typically a data dictionary. This integration supports automation in software development methodologies.
- This central repository is usually a data dictionary containing the definition of all composite and elementary data items.



A CASE environment

Difference from programming environments:

- The true power of a tool set can be realised only when these set of tools integrated into a common framework or environment.
- If CASE tools are not integrated, then the data generated by one tool would have to input to another, leading to inefficiencies.
- Different vendors are likely to use different formats. This results in additional effort of exporting data from one tool and importing to another.
- E.g : Turbo c environment, Visual Basic, Visual C++, etc

2.1 Benefits of CASE :

- CASE tools provide cost savings in software development, reducing effort by 30-40%.
- They improve quality through software development phases, minimizing human errors.
- High-quality and consistent documentation is achieved by maintaining important data in a central respiratory, reducing redundancy.
- CASE tools reduce manual effort, automating repetitive tasks like.
- They lead to significant cost savings in software maintenance through traceability and consistency checks.
- Adoption of CASE tools promotes a structured and orderly approach in software development.

3. CASE SUPPORT IN AND SOFTWARE LIFE CYCLE :

- CASE tools support development methodology and provide certain amount of consistency checking between different phases.

3.1 Prototyping Support :

- 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.

A good Prototyping tool features:

- 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.

3.2 Structured Analysis and Design :

- Diagramming techniques are used for structured analysis and structured design.
- A 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.
- Analysis and through all levels of analysis hierarchy.

3.3 Code Generation :

- 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.
- 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.
- The tool should generate records, structures, class definition automatically.
- It should generate database tables for relational database management systems.

3.4 Test Case Generator :

- 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.

4. OTHER CHARACTERISTICS OF CASE TOOLS :

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

4.1 Hardware and Environmental Requirements :

- Instead of defining hardware requirements for a CASE tool, select the most optimal CASE tool configuration for a given hardware configuration.
- The heterogeneous network is one instance of distributed environment. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. dictionary server may support one or more projects.
- 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.

4.2 Documentation Support :

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

4.3 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

4.4 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.

4.5 Reverse Engineering Support :

- The tool should support generation of structure charts and data.
- It should populate the data dictionary from the source code.
- It should contain fro conversion tool forms like indexed sequential file structure, hierarchical and network database to relational database systems.

4.6 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 and provide analysis reports like cross-referencing, impact analysis, etc.
- Ideally, it should support a query language to view its contents.

4.7 Tutorial and Help :

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

5. TOWARDS SECOND GENERATION CASE TOOL :

- An important feature of the second-generation CASE tool is the direct support of any adapted methodology. Features in the second-generation CASE tool are :

1. **Intelligent diagramming support:**
   - The future CASE tools would provide help to aesthetically and automatically lay out the diagrams
2. **Integration with implementation environment:**

- The CASE tools should provide integration between design and implementation.

3. **Data dictionary standards:**
   - The user should be allowed to integrate many development tools into one environment.
4. **Customisation support:**
   - The user should be allowed to define new types of objects and connections.

6. ARCHITECTURE OF A CASE ENVIRONMENT :

- The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository.

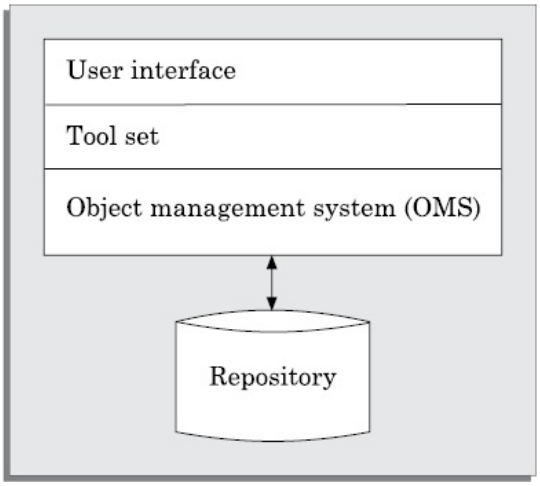


Figure 6.1: Architecture of a modern CASE environment.

- **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 under lying 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.

SOFTWARE MAINTENANCE

Definition: Software Maintenance is the process of modifying and updating software applications after they have been deployed. It ensures that the software continues to function correctly, meets new requirements, and remains secure. *On the other hand software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platform, etc.

1. CHARACTERISTICS OF SOFTWARE MAINTENANCE :

Software maintenance keeps software compatible with new hardware, OS, and tools. It ensures updates do not introduce new errors. Every software product continues to evolve after its development through maintenance efforts .

1.1 Types of software maintenance :

There are three main types of software maintenance, they are :

- **Corrective:** Corrective maintenance of software product is necessary either to rectify the bugs observed while the system is in use.
- **Adaptive**: Adaptive maintenance for updating software to work with changes in the environment.
- **Perfective**: Perfective maintenance improves performance, efficiency, or usability without changing core functionality.

1.2 Characteristics of Software Evolution :

- Lehman and Belady have studied the characteristics of evolution of several software products in 1980.They have expressed their observations in the form of three laws. They are:
- **Lehman's first law:** This law states that every product irrespective of how well designed must undergo maintenance. Infact, when a product does not need any more maintenance, it is a sign that the product is about to be discarded.
- **Lehman's second law:** This law states that as more maintenance is performed on a program, its structure tends to degrade. This happens because new functions are added on top of the existing program, often in ways the original design didn't support. Without redesign, these additions become more complex than necessary. Additionally, quick-fix solutions lead to inconsistent documentation, which becomes less helpful over time.
- **Lehman's third law:** This law states that the rate at which code is written or modified is approximately the same during development and maintenance.

1.3 Special problems associated with software maintenance :

- During maintenance it is necessary to thoroughly understand someone else's work ,and then carry out the required modifications and extensions.
- The majority of software products needing maintenance are legacy (aged) products. 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.

2. SOFTWARE REVERSE ENGINEERING :

- Software reverse engineering is the process of recovering the design and the requirements specifications 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.
- The first stage of reverse engineering 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 2.1.
  - A program can be reformatted using any of the several available prettyprinter programs.
  - Assigning meaningful variable names is important because meaningful variable names is the most helpful code documentation.
  - Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.
  - Remove GOTO statements.

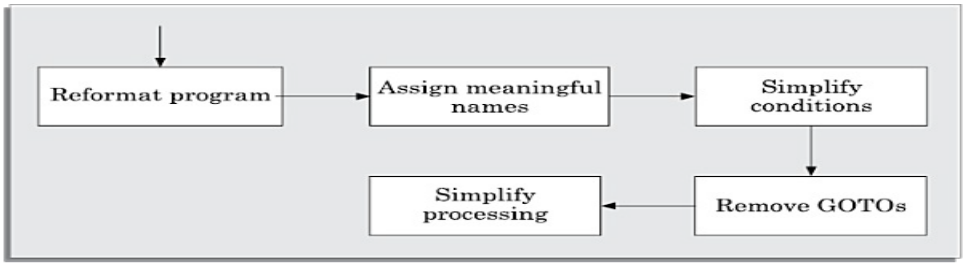


Figure 2.1: Cosmetic changes carried out before reverse engineering.

- After the cosmetic changes have been carried out on a legacy software, the proces of extracting the code, design, and the requirements specification can begin. These activities are shown in Figure 2.2 .

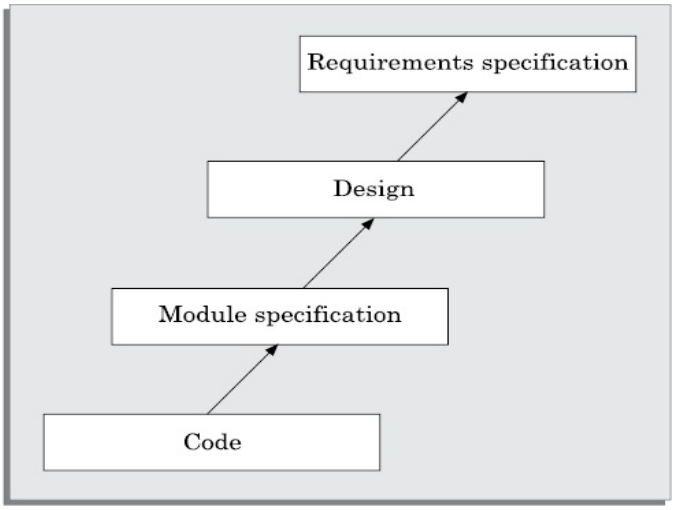


Figure 2.2: A process model for reverse engineering.

3. SOFTWARE MAINTENANCE PROCESS MODELS :

- The activities involved in a software maintenance project are not unique and depend on several factors such as:
  - The extent of modification to the product required.
  - The resources available to the maintenance team.
  - The conditions of the existing product.
  - The expected project risks, etc.
- For complex maintenance projects the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle. Two broad categories of process models are proposed, they are

MAINTENANCE PROCESS MODEL 1 :

- 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.
- 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.
- Debugging of the reengineered system becomes easier

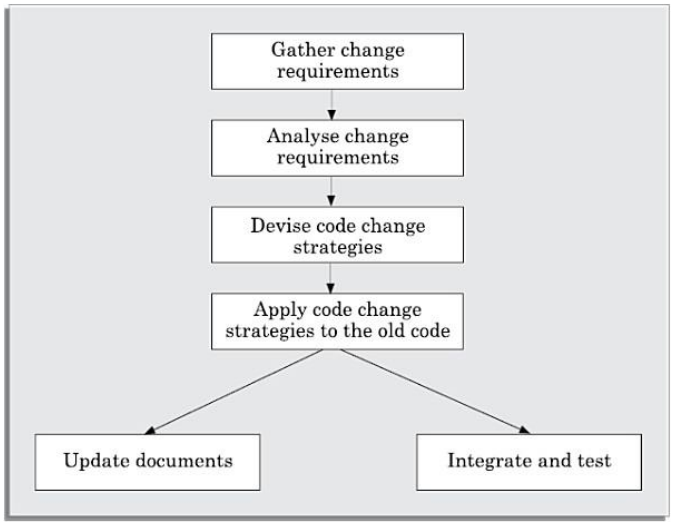


Figure 3.1 : Maintenance process model 1.

MAINTENANCE PROCESS MODEL 2 :

- 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.
- The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed to extract the module specifications.
- The module specifications then analysed to produce the design.
- The design is analysed to produce the original requirements specifications.
- This approach is more costly than the first approach.

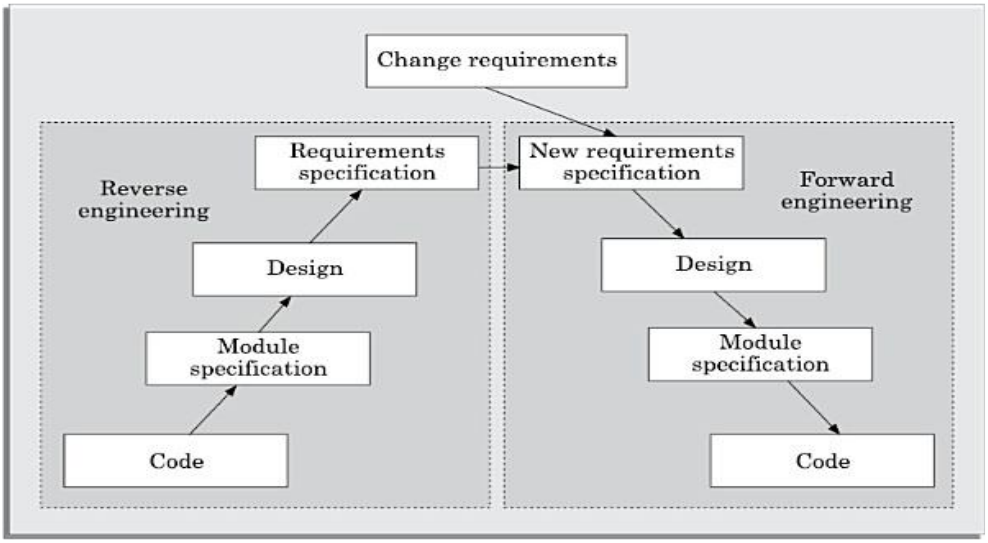


Figure 3.2: Maintenance process model 2.

- **AN EMPIRICAL STUDY** indicates that process 1 is preferable when the amount of rework is no more than 15 per cent (see Figure 3.3). 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.

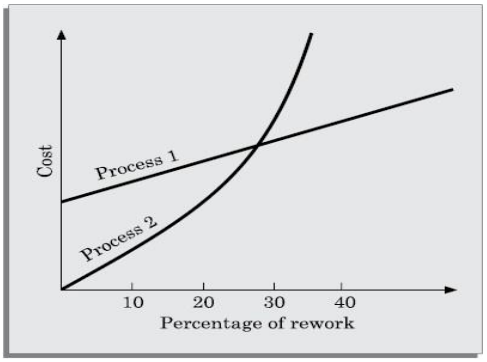


Figure 3.3: Empirical estimation of maintenance cost versus percentage rework.

4. ESTIMATION OF MAINTENANCE 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

- **KLOCadded** is the total kilo lines of source code added during maintenance.
- **KLOCdeleted** is the total KLOC deleted during maintenance.

- Maintenance cost is the multiplication of annual change traffic (ACT) with the total development cost

Maintenance cost = ACT × Development cost

- Most maintenance cost estimation models, 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.

SOFTWARE REUSE

Definition: Software products are expensive. A possible way to reduce development cost is to reuse parts from previously developed software.

1. WHAT CAN BE REUSED?

- Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:
  - Requirements specification
  - Design
  - Code
  - Test cases
  - Knowledge
- Knowledge is the most abstract development artifact that can be reused. A planned reuse of knowledge can increase the effectiveness of reuse.

2. WHY ALMOST NO REUSE SO FAR?

- In many software development industries, engineers often feel that the current system they are developing is similar to previous systems. However, no attention is paid to reusing elements from past systems. Everything is built from scratch, leading to delays, and there is little time to explore how similarities between systems could be leveraged.
- Even organizations that attempt reuse face challenges. Creating components that are reusable across different applications is difficult, as it is hard to predict which components will be reusable. Even when reusable components are available, programmers often prefer to create their own due to the complexity and difficulty in adapting the available components to new applications.
- These observations highlight fundamental issues for reuse, such as well-defined functionality and standardized interfaces, which must be addressed in any reuse program.

3. BASIC ISSUES IN ANY REUSE PROGRAM :

The following are some basic issues in program reuse

- **Component creation:** The reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important.
- **Component indexing and storing :** Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse.
- **Component searching :** The programmers need to search for right components matching their requirements in a database of components.
- **Component understanding :** The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component.
- **Component adaptation:** Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand.
- **Repository maintenance :** A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository.

4. A REUSE APPROACH :

The reusable components need to be identified after every development project is completed. Domain analysis is a promising approach to identify reusable components.

4.1 Domain Analysis :

The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse Domain :

- A reuse domain is a shared understanding of some community, characterised by concepts, techniques, and terminologies that show some coherence.
- Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.
- Analysis of the application domain is required to identify the reusable components.
- The actual construction of the reusable components for a domain is called domain engineering.

Evolution of a reuse domain :

Stage 1 : There is no clear and consistent set of notations. Obviously, no reusable components are available. All software is written from scratch.

Stage 2 : Here, only experience from similar projects are used in a development effort. This means that there is only knowledge reuse.

Stage 3: At this stage, the domain is ripe for reuse. The set of concepts are stabilised and the notations standardised. Standard solutions to standard problems are available. There is both knowledge and component reuse.

Stage 4: The domain has been fully explored. The software development for the domain can largely be automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an *application generator*.

4.2 ComponentClassification :

- Components need to be properly classified to develop an effective indexing and storage scheme. Hardware components are often classified using a multilevel hierarchy.
- At the lowest level, components are described in several forms—natural language description, logic schema, timing information, etc.
- The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's Classification Scheme :

Each component is best described using several different characteristics or facets. For e.g , objects can be classified based on:

- Actions they embody.
- Objects they manipulate.
- Data structures used.
- Systems they are part of.

Prieto-Diaz's faceted classification involves choosing an n-tuple that best fits a component.

4.3 Searching :

- A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results would do a browsing using the links provided to look up related items.

4.4 Repository maintenance :

- Repository Maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. Also, the links relating the different items may need to be modified to improve the effectiveness of search.

4.5 Reuse without Modifications :

- Reuse without modification is much more useful than the classical program libraries. These can be supported by compilers through linkage to run-time support routines (application generators).
- Application generators translate specifications into application programs. The specification usually is written using 4GL or might also be in a visual form. The programmer would create a graphical drawing using some standard available symbols.
- Application generators have significant advantages over simple parameterised programs. The other advantages include fewer errors, easier to maintain, substantially reduced development effort, and the fact that one need not bother about the implementation details.

5. REUSE AT ORGANISATION LEVEL :

- Reusability should be a standard part in all software development activities including specification, design, implementation, test, etc.
- Achieving organisation-level reuse requires adoption of the following steps:

- Assessing a product's potential for reuse.
- Refine the item for greater reusability.
- Enter the product in the reuse repository.

Assessing a product's potential for reuse:

Assessment of a components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to assess a component's reusability. Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored.

A sample questionnaire to assess a component's reusability is the following:

- Is the component's functionality required for implementation of systems in the future?
- How common is the component's function within its domain?
- Would there be a duplication of functions within the domain if the component is taken up?
- Is the component hardware dependent? Is the design of the component optimised enough?
- If the component is non-reusable, then can it be decomposed to yield some reusable components?
- Can we parametrise a non-reusable component so that it becomes reusable?

Refining products for greater reusability :

For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localised using data encapsulation techniques. The following refinements may be carried out:

- **Name generalisation:** The names should be general, rather than being directly related to a specific application.
- **Operation generalisation:** Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.
- **Exception generalisation:** This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.

Handling portability problems:

- The programs often need to call some operating system functionality and these calls may not be the same on all machines. Also, programs use some function libraries, which may not be available on all host machines. A portability solution is to overcome these problems shown in Figure 5.1.
- The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-related calls should be routed through the portability interface.
- One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.

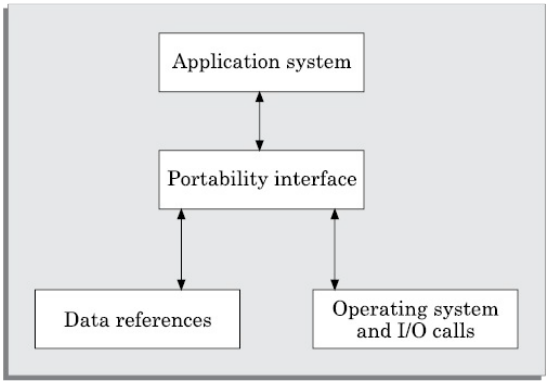


Figure 5.1: Improving reusability of a component by using a portability interface.

5.1 Current State of Reuse :

In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organisations that most of the factors inhibiting an effective reuse program are non- technical. Some of these factors are the following:

- Need for commitment from the top management.
- Adequate documentation to support reuse.
- Adequate incentive to reward those who reuse.
- Providing access and information about reusable components.