

Software Maintenance and Computer Aided Software Engineering (CASE)

(Lecture 12)

Introduction



- Software maintenance:
 - any modifications to a software product after it has been delivered to the customer.
- Software maintenance is an important activity for many organizations.

Introduction

- Maintenance is inevitable for almost any kind of product.
- Most products need maintenance:
 - due to wear and tear caused by use.
- Software products do not need maintenance on this count.

Introduction

- Many people think
 - only bad software products need maintenance.
- The opposite is true:
 - bad products are thrown away,
 - good products are maintained and used for a long time.

Introduction

□ Software products need maintenance for three reasons:

□ corrective

□ adaptive

□ perfective

Corrective

- Corrective maintenance of a software product:
 - to correct bugs observed while the system is in use.
 - to enhance performance of the product.

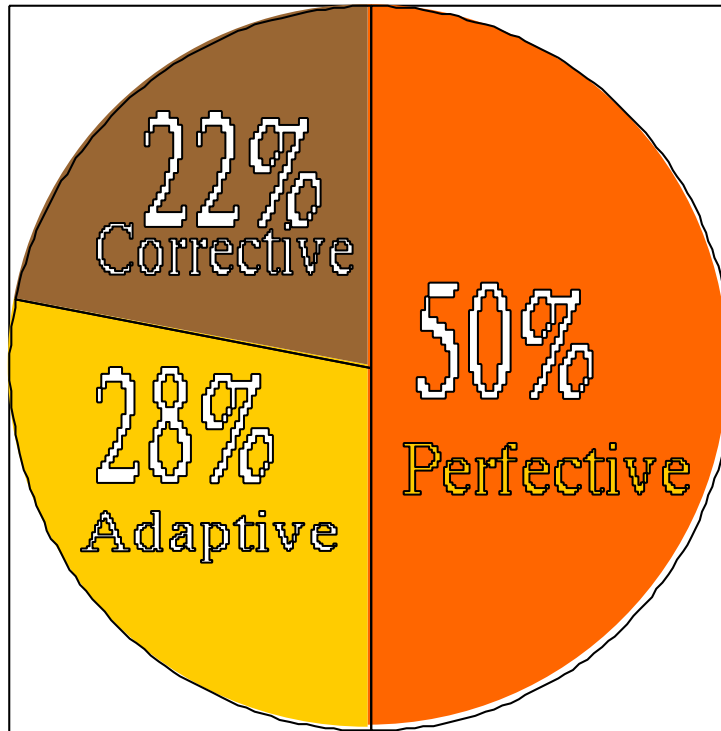
Adaptive

- A software product needs maintenance (porting) when customers:
 - need the product to run on new platforms,
 - or, on new operating systems,
 - need the product to interface with new hardware or software.

Perfective

- Perfective maintenance:
 - to support new features required by users.
 - to change some functionality of the system due to customer demands.

Maintenance Effort Distribution



Causes for maintenance

- During development:
 - Software not anticipated to last very long (e.g., Y2K problem).
- Rate of hardware obsolescence:
 - immortality of software products
 - maintenance necessary for software performing low-level functions.

Causes for maintenance



- Users want existing software to run on new platforms:
 - to run in new environments,
 - and/or with enhanced features.

Causes for maintenance





- Whenever other software it works with change:
 - maintenance is needed to cope up with the newer interface.
 - For instance, a software product may need maintenance when the operating system changes.

Software evolution

- Every software product continues to evolve after its development:
 - through maintenance efforts.
- Larger software products stay in operation for longer time:
 - because of high replacement cost.

Laws of Maintenance



-   There will always be a lot of old software needing maintenance.
-   Good products are maintained, bad products are thrown away.

Laws of Maintenance



□ Lehman's first Law:

- "Software products must change continuously, or become progressively less useful."

Laws of Maintenance



□ Lehman's Second Law

- "When software is maintained, its structure degrades,
 - unless active efforts are made to avoid this phenomenon."

Laws of Maintenance



□ Lehman's Third Law:

□ "Over a program's life time,

□ its rate of development is approximately constant."

Other Laws of Maintenance

- All large programs will undergo significant changes during operation phase of their life cycle,
 - regardless of *apriori* intentions.

Legacy code--- Major maintenance problems

- Unstructured code (bad programs)
- Maintenance programmers have:
 - insufficient knowledge of the system or the application domain.
 - Software maintenance has a bad image.
- Documentation absent, out of date, or insufficient.

Insufficient knowledge

- Maintenance team is usually different from development team.
- even after reading all documents
 - it is very difficult to understand why a thing was done in a certain way.
- Also there is a limit to the rate at which a person can study documents
 - and extract relevant information

Bad image of maintenance?

- Maintainers are skilled not only in writing code:
 - proficient in understanding others' code
 - detecting problems, modifying the design, code, and documentation
 - working with end-users

Maintenance

Nightmares

- Use of gotos
- Lengthy procedures
- Poor and inconsistent naming
- Poor module structure
- Weak cohesion and high coupling
- Deeply nested conditional statements
- Functions having side effects

How to do better maintenance?

- Program understanding
- Reverse engineering
- Design recovery
- Reengineering
- Maintenance process models

Maintenance activities

- Two types of activities:

- Productive activities:

- modification of analysis, design, coding, etc.

- Non-productive activities:

- understanding system design, code, etc.

Software Reverse Engineering



- By analyzing a program code, recover from it:
 - the design and the requirements specification.

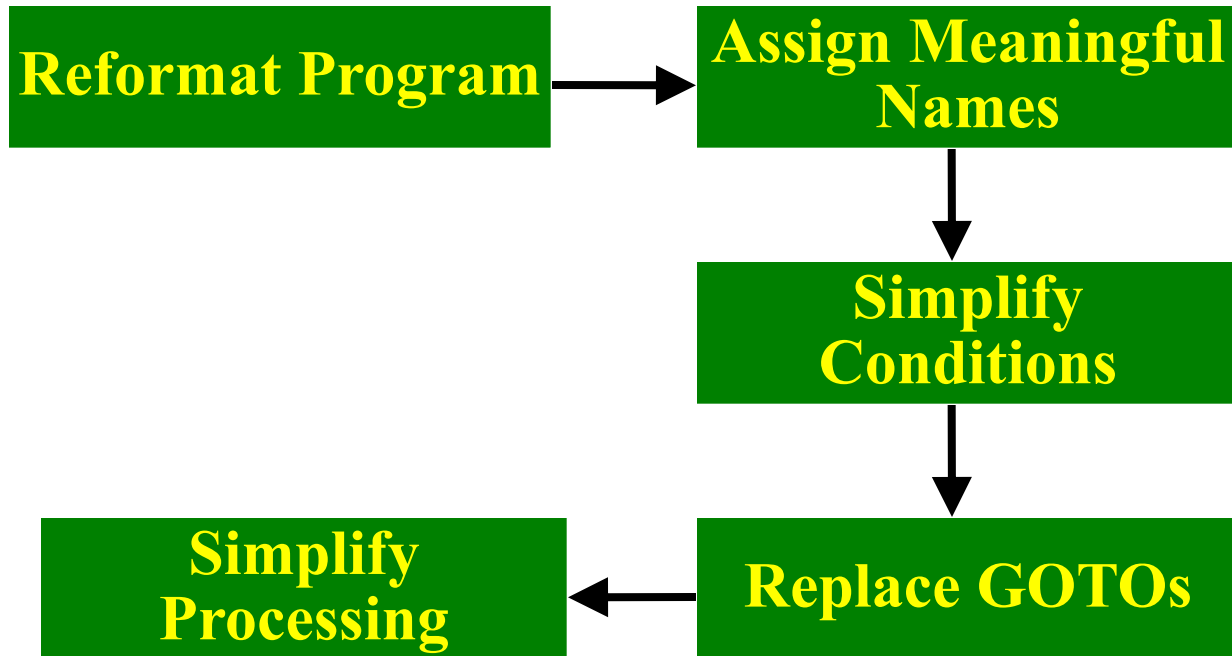
Software Reverse Engineering

- Reverse engineering is an important maintenance technique:
 - several existing software products are unstructured,
 - lack proper documentation,
 - were not developed using software engineering principles.

Software Reverse Engineering

- First carry out cosmetic changes to the code to improve:
 - readability,
 - structure,
 - understandability.

Cosmetic changes



Cosmetic Changes



- Reformat the program:
 - use any pretty printer program
 - layout the program neatly.
- Give more meaningful names to:
 - Variables, data structures, and functions.

Cosmetic Changes

- Replace complex and nested conditional expressions:
 - simpler conditional statements
 - whenever appropriate use case statements.

Software Reverse Engineering

- In order to extract the design:
 - fully understand the code.
- Automatic tools can be used to help derive:
 - data flow and control flow diagrams from the code.

Software Reverse Engineering

- Extract structure chart:
 - module invocation sequence and data interchange among modules.
- Extract requirements specification:
 - after thoroughly understanding the code.
 - design has been extracted.

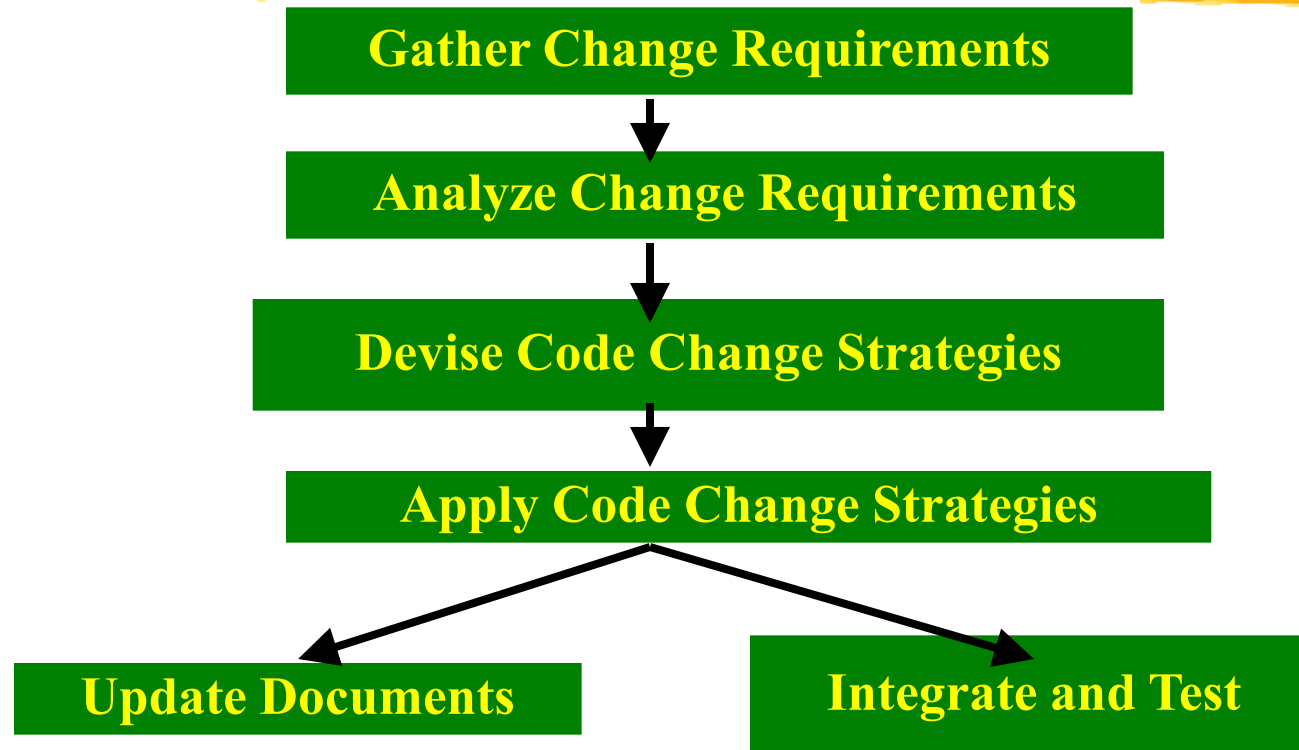
Software Maintenance Process Models

- Maintenance activities are not unique:
 - depend on the extent of modifications required,
 - also, depend on condition of the product:
 - how structured it is,
 - how well documented it is, etc.

Software Maintenance Process Model - 1

- When the required changes are small and simple:
 - the code can be directly modified
 - changes reflected in all relevant documents.
 - more elaborate activities are required when required changes are not trivial.

Software Maintenance Process Model - 1



Software Maintenance Process Model 1

- Start by gathering change requirements.
- Analyze change requirements
 - formulate strategies for code change.

Software Maintenance Process Model 1

- Formulating strategies for code change:
 - presence of few members of the original development team
 - helps in reducing cycle time,
 - especially for unstructured and inadequately documented code.

Software Maintenance Process Model 1

- Availability of a working old system at the maintenance site:
 - greatly helps the maintenance team
 - provides a good insight into the working of the old system
 - can compare the working of the modified system with the old system.

Software Maintenance Process Model 1



- Debugging the system under maintenance becomes easier:
 - program traces of both the systems can be compared to localize bugs.

Software Maintenance Process Model -2

- For complex maintenance projects, software reengineering needed:
 - a reverse engineering cycle followed by a forward engineering cycle.
 - with as much reuse as possible from existing code and other documents.

Maintenance Process Model 2

- Preferable when:
 - amount of rework is significant
 - software has poor structure.
- Can be represented by a reverse engineering cycle:
 - followed by a forward engineering cycle.

Software reengineering



- Many aging software products belong to this category.
- During the reverse engineering,
 - the old code is analyzed (abstracted) to extract the module specifications.

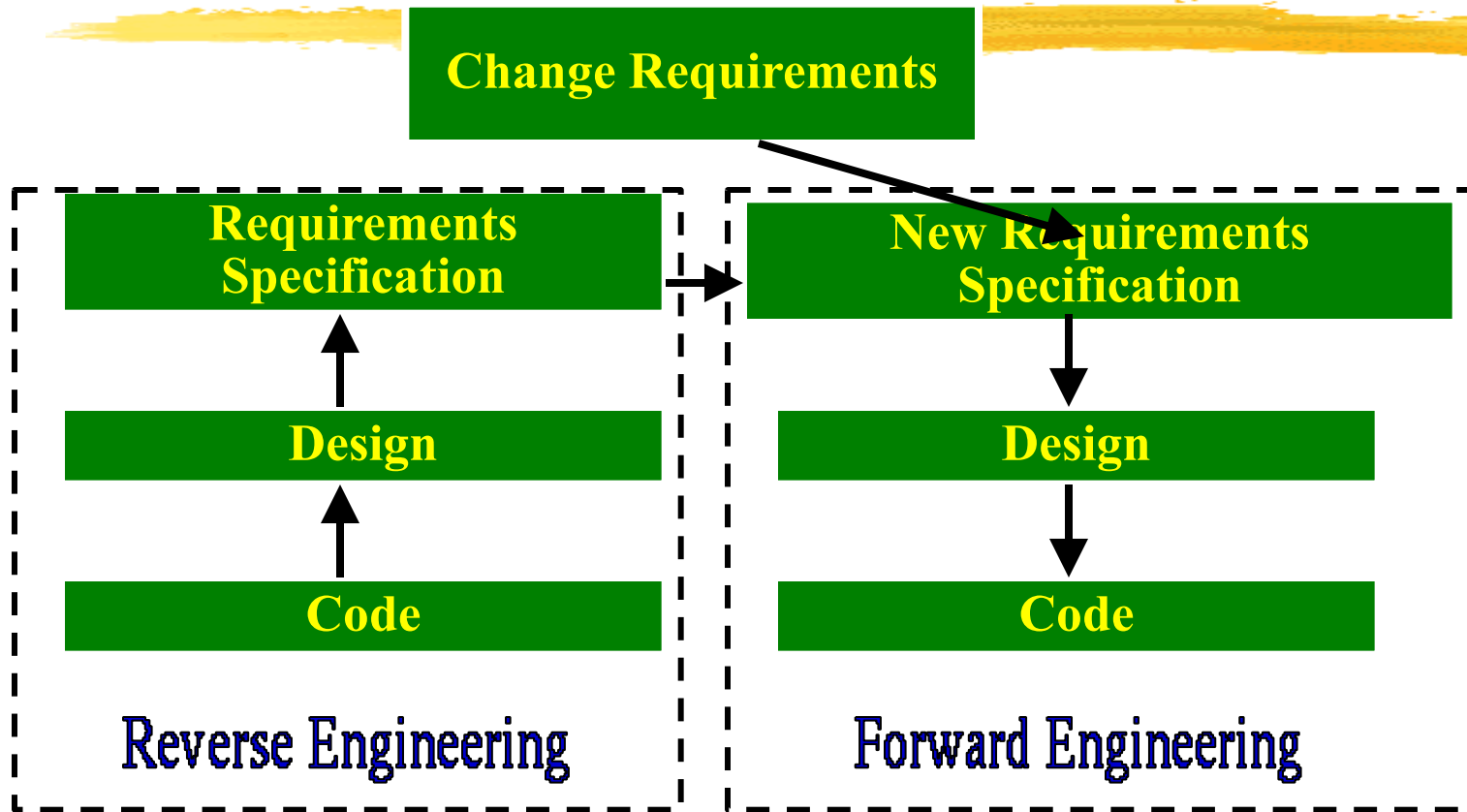
Software reengineering

- The module specifications are analyzed
 - to produce the design.
- The design is analyzed (abstracted)
 - to produce the original requirements specification.
- The change requests are then applied to the requirements specification:
 - arrive at the new requirements specification.

Software reengineering

- Forward engineering is carried out to produce the new code.
- During design, module specification, and coding:
 - substantial reuse is made from the reverse engineered products.

Process model for Software reengineering



Software reengineering

- Advantages of reengineering:
 - produces better design than the original product,
 - produces required documents,
 - often results in higher efficiency.

Software reengineering

- Efficiency improvements are brought about by better design.
 - However, this approach is more costly than the first approach.
- An empirical study indicates:
 - process 1 is preferable when amount of rework is no more than 15%.

Software reengineering



- Reengineering is preferable when:
 - amount of rework is high,
 - product exhibits high failure rate.
 - product difficult to understand.

Computer Aided Software Engineering (CASE)



- CASE tools help in software development and maintenance.
- CASE is a much talked about topic in software industries.

CASE and Its Scope

- CASE tool is a generic term:
 - denotes any form of automated support for software engineering.
- In a more restrictive sense:
 - a CASE tool automates some software development activity.

CASE and Its Scope

- Some CASE tools assist in phase-related tasks:
 - specification, structured analysis, design, coding, testing, etc.
- Other tools help non-phase activities:
 - project management and configuration management.

Objectives of CASE



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

CASE Environment



- Although individual CASE tools are useful:
 - true power of a tool set can be realized only when:
 - all CASE tools are integrated together.

CASE Environment

- Tools covering different stages of life cycle share information (data):
 - they should integrate through some central repository (store)
 - consistent view of development information.

CASE Environment

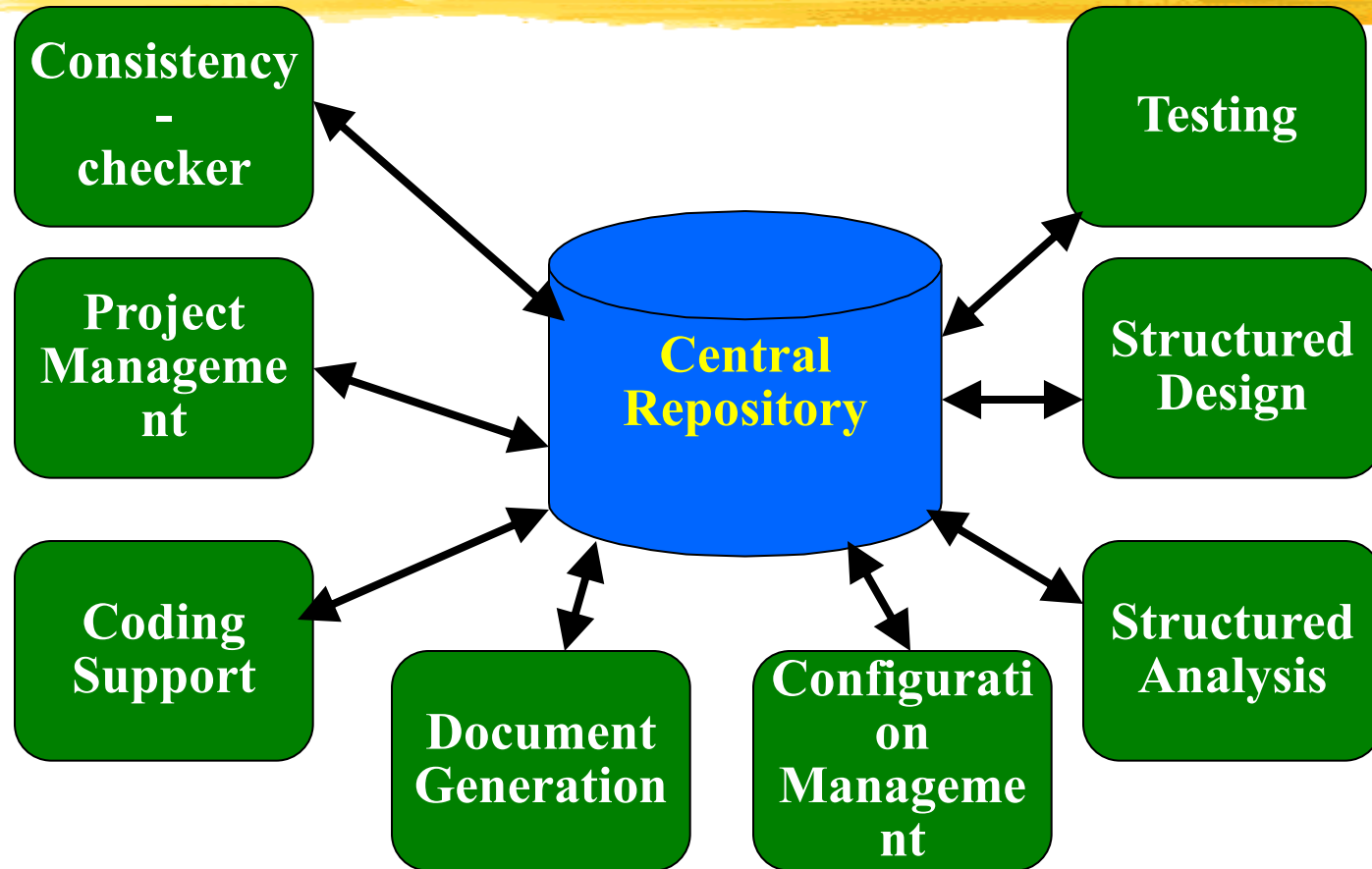
- The central repository is the data dictionary:
 - contains definition of all composite and elementary data items.
 - through this repository all CASE tools share information.

Programming Environment



- A CASE environment helps:
 - automate step-by-step methodologies.
- In contrast to CASE environment:
 - a programming environment denotes tools supporting coding phase alone.

Schematic representation of architecture of CASE environment



Benefits of CASE

- A key benefit of using CASE environment:
 - cost saving through all developmental phases.
- Studies carried out to measure the impact of CASE usage:
 - cost saving between 30% to 40%.

Benefits of CASE

- Use of CASE tools leads to improvements in quality:
 - becomes easy to iterate through different software development phases.
 - chances of human error is reduced.
 - CASE tools help produce higher quality and consistent documents.

Benefits of CASE

- Data relating to a software product are maintained in a central repository:
 - redundancy in the stored data is reduced.
 - chances of inconsistent documentation is reduced.

Benefits of CASE



- CASE tools take drudgery out from software engineers' work:
 - engineers need not manually check balancing of the DFDs
 - easily draw diagrams and produce documentation, etc.

Benefits of CASE

- CASE tools lead to cost saving in software maintenance effort:
 - traceability and consistency checks,
 - systematic information capture during various development phases.

Benefits of CASE

□ Introduction of a CASE environment:

- impacts the style of working of engineers.

- makes them oriented towards structured and orderly approach.

Prototyping Support



- Prototyping CASE tool:
 - often used in graphical user interface (GUI) development,
 - supports creating a GUI using a graphics editor.

Prototyping Support

- The user should be allowed to define:
 - data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.

Structured Analysis and Design

- A CASE tool should:
 - support some standard structured analysis and design technique.
 - support easy creation of analysis and design diagrams.
 - should provide easy navigation through different levels of design and analysis diagrams.

Structured Analysis and Design

- The tool must support completeness and consistency checking.
- The tool should disallow inconsistent operations:
 - but, it is difficult to implement such a feature.

Code Generation

- As far as code generation is concerned:
 - expectations from a CASE tool is low.
- The CASE tool should support:
 - generation of module skeletons in one or more popular languages.
 - Another reasonable requirement is traceability from source code to design.

Code Generation

- It should automatically generate header information:
 - copyright messages,
 - brief description of the module,
 - author name and date of creation, etc.

Code Generation



- The tool should generate data records or structures automatically:
 - using data dictionary definitions.
 - It should generate database tables for relational database management systems.

Code Generation



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

Testing Support

- Static and dynamic program analysis of programs.
- It should generate test reports in ASCII format:
 - which can be directly imported into the test plan document.

Desirable Features



- The tool should work satisfactorily
 - when many users work simultaneously.
- The tool should support windowing interface:
 - Enable the users to see more than one diagram at a time.
 - Facilitate navigation and switching from one part to the other.

Documentation Support



- The deliverable documents:
 - should be able to incorporate text and diagrams from the central repository.
 - help in producing up-to-date documentation.

Desirable Features

- The CASE tool should integrate
 - with commercially available desk-top publishing packages.
- It should be possible to export text, graphics, tables, data dictionary reports:
 - to DTP packages in standard formats such as PostScript.

Project Management

- It should support collecting, storing, and analyzing 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 exported by the tool should preferably be in ASCII format.
- The data dictionary should provide
 - a programming interface to access information.

Reverse Engineering Support

- The tool should support:
 - generating structure chart, DFD, and data dictionary from source code.
 - should populate the data dictionary from source code.

Data Dictionary Interface



- Data dictionary interface should provide
 - viewing and updating the data definitions.
 - print facility to obtain hard copy of the viewed screens.
 - analysis reports like cross-referencing, impact analysis, etc.
 - it should support a query language.

Tutorial and Help

- Successful use of CASE tools:
 - depends on the users' capability to effectively use all supported features.
- For the first time users:
 - a computer animated tutorial is very important.

Tutorial and Help



- The tutorial should not be limited to teaching the user interface part only:
 - The tutorial should logically classify and cover all techniques and facilities.
 - The tutorial should be supported by proper documentation and animation.

Towards Next Generation CASE Tool

- An important feature of next generation CASE tools:
 - be able to support any methodology.
- Necessity of a CASE administrator for every organization:
 - who would tailor the CASE environment to a particular methodology.

Intelligent Diagramming Support



- Future CASE tools would
 - aesthetically and automatically lay out the diagrams.

Towards Next Generation CASE Tool



- The user should be allowed to:
 - integrate many different tools into one environment.
 - It is highly unlikely that any one vendor will be able to deliver a total solution.

Towards Next Generation CASE Tool



- A preferred tool would support tune up:
 - user would act as a system integrator.
 - This is possible only if some data dictionary standard emerges.

Customization 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.

Summary

- We discussed some fundamental concepts in software maintenance.
- Maintenance is the mostly expensive phase in software life cycle:
 - during development emphasize on maintainability to reduce the maintenance costs.

Summary

- We discussed software reverse engineering:
 - extract design from code.
- Reengineering is a reverse engineering cycle:
 - followed by a forward engineering cycle

Summary

- Maintenance process models:
 - Process model for small changes
 - Process model for reengineering
- We also discussed:
 - applicability of process models to maintenance projects.

Summary

- We discussed important features of present day CASE tools:
 - and the emerging trends.
- Use of CASE tools is indispensable for large software projects:
 - where a team of software engineers work together.

Summary

- The trend is now towards:
 - distributed workstation-based CASE tools.
- We discussed some desirable features of CASE tools.