TED (10)-4072 Reg. No. …….

(REVISION-2010) Signature ……..……

FIFTH SEMESTER DIPLOMA EXAMINATION IN COMPUTER ENGINEERING AND INFORMATIONTECHNOLIGY- MARCH, 2013

SOFTWARE ENGINEERING

[*Time:* 3 hours

(Maximum marks: 100)

Marks

PART –A
(Maximum marks: 10)

I. Answer all questions in two sentences. Each question carries 2 marks.

1. Define software life cycle.

   A software life cycle is a series of identifiable stages that a software product undergoes during its lifetime.

2. List any two software design methods

   - Function- Oriented Design
   - Object- Oriented Design

3. Write the outcome of requirement analysis.

The main purpose of the requirements analysis activity is to analyze the collected information to obtain a clear understanding of the product to be developed, with a view to removing all ambiguities, incompleteness, and inconsistencies from the initial customer perception of the problem.

4. Give a sample test suite for solution to quadratic equation.

```
classdef SolverTest < matlab.unittest.TestCase
   % SolverTest tests solutions to the quadratic equation
   % a*x^2 + b*x + c = 0

   methods (Test)
      function testRealSolution(testCase)
         actSolution = quadraticSolver(1,-3,2);
```

```
        expSolution = [2,1];
        testCase.verifyEqual(actSolution,expSolution);
    end
    function testImaginarySolution(testCase)
        actSolution = quadraticSolver(1,2,10);
        expSolution = [-1+3i, -1-3i];
        testCase.verifyEqual(actSolution,expSolution);
    end
  end
end
```

5. State the need of CASE

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. The primary reasons for using a CASE tool are:

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

PART – B

II. Answer *any five* questions. Each question carries 6 marks

1. Describe empirical estimation technique.

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, over the years different activities involved in estimation have been formalized to certain extent. There are two popular empirical estimation techniques are using

- Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this technique, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example ,

he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

- Delphi estimation Technique

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds.

2. Explain the steps involved in Risk Management

A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway. If a risk becomes, true the anticipated problem becomes a reality and is no more a risk. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project.. Risk management consists of three essential activities: risk identification, risk assessment and risk containment.

Risk identification:

The project manager needs to anticipate the risks in the project as early as possible so that the impact of the risks can be minimized by making effective risk management plans. So, early risk identification is important .Risk identification is somewhat listing down your nightmares.

Project risks:

Project risks concern varies forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage.

Technical risks:

Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence.

Business risks:

This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

Risk assessment:-

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

- The likelihood of a risk coming true (denoted as r). • The consequence of the problems associated with that risk (denoted as s).

Based on these two factors, the priority of each risk can be computed: p = r * s

Where, p is the priority with which the risk must be handled, r is the probability of the risk becoming true, and s is the severity of damage caused due to the risk becoming true. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

Risk containment:-

After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks require different containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling the risk.

There are three main strategies to plan for risk containment:

- Avoid the risk: - This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.
- Transfer the risk:- This strategy involves getting the risky component developed by a third party, buying insurance cover, etc.
- Risk reduction:- This involves planning ways to contain the damage due to a risk.

Risk leverage :Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

Risk leverage = (risk exposure before reduction – risk exposure after reduction) / (cost of reduction)

3. Write down the characteristics of a good SRS

The important properties of a good SRS document are the following:

- Concise:-

The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

- Structured:-

  It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time.

- Black-box view:-

  It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system

- Conceptual integrity:-

  It should show conceptual integrity so that the reader can easily understand it.

- Response to undesired events:-

  It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

- Traceable:-

  It should be possible to trace a specific requirement to the design elements that implement it and vice versa. Similarly, it should be possible to trace requirement to the code segments that implement it and the test cases that test the requirement and vice versa. Traceability is important to verify the results of a phase with the previous phase, to analyze the impact of a change, etc.

- Verifiable:-

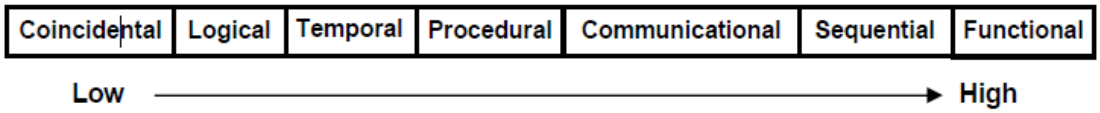
  All requirements of the system as documented in the SRS document should verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation. Requirements such as the system should be user-friendly are not verifiable. On the other hand, when the name of a book is entered, the software should display the location of the book, if the book is available is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

4. Distinguish between cohesion and coupling

- **Cohesion:-**

  Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules.

The different classes of cohesion that a module may possess are depicted in fig.



Coincidental cohesion:-A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

Logical cohesion:-A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc.

Temporal cohesion:-When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion.

Procedural cohesion:-A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

Communicational cohesion:- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

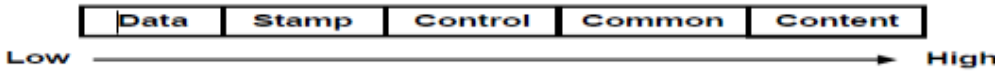
Sequential cohesion:-A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

Functional cohesion:-Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function.

- **Coupling:-**

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules.

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown in fig.



Data coupling:-Two modules are data coupled, if they communicate through a parameter.

Stamp coupling:-Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling:-Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.

Common coupling:-Two modules are common coupled, if they share data through some global data items.

Content coupling:-Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

5. Explain different coding standards.
    - well-documented.
6. Summarize the reliability metrics

Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has.

• Rate of occurrence of failure (ROCOF). ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures).

• Mean Time To Failure (MTTF). MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t1, t2, …, tn. Then, MTTF can be calculated as $1 \; 1 \; (\; 1) \; n \; ii \; itt \; n \; +- = = - \sum$.

• Mean Time To Repair (MTTR). Once failure occurs, some timeis required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

• Mean Time Between Failure (MTBR). MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours.

• Probability of Failure on Demand (POFOD). Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made.

7. Explain the benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

• A key benefit arising out of the use of a CASE environment is cost saving through all development phases.

• Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.

• CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.

• CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.

• CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and

consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.

• Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

PART – C

(Answer *one* full question from each unit. Each question carries 15 marks)

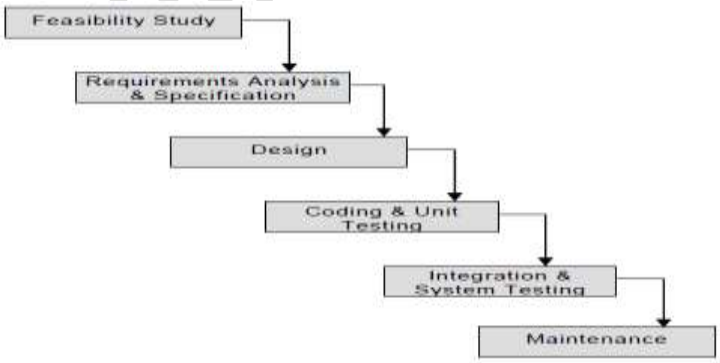
UNIT – I

III. Illustrate any three life cycle models with neat sketch.

❖ Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it can't be used in actual software development projects. Thus, this model can be considered to be a theoretical way of developing software. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases:-



• Feasibility Study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

• Requirements Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

- Design

  The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

- Coding and Unit Testing

  The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

- Integration and System Testing

  Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested anda set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out.

  - α – testing: It is the system testing performed by the development team.
  - β – testing: It is the system testing performed by a friendly set of customers.
  - acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

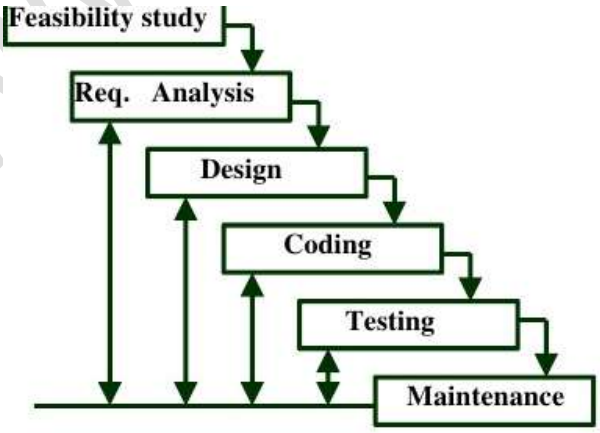
- Maintenance

  Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

❖ Iterative Waterfall Model

The classical waterfall model is not a practical model in the sense that it can't be used in actual software development projects. Thus, this model can be considered to be a theoretical way of developing software.The iterative waterfall model as making necessary changes to the classical waterfall model so that it becomes applicable to practical software development projects. Essentially the main change to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases as shown in fig. The feedback paths allow for correction of the errors committed during a phase, as and when these are detected in a later phase. For example, if during testing a design error is identified, then the feedback path allows the design to be reworked and the changes to be reflected in design documents. However, observe that there is no feedback path to the feasibility stage. This means that the feasibility study errors cannot be corrected. Though errors are inevitable in almost every phase of development, it is desirable to detect these errors in the same phase, in which they occur.



- Shortcomings of the Iterative Waterfall Model

1. The waterfall model cannot satisfactorily handle the different types of risks that a real life software project may suffer from.
2. To achieve better efficiency and higher productivity, most real life projects find it difficult to follow the rigid phase sequence prescribed by the waterfall model.

❖ Spiral Model

The Spiral model of software development is shown in fig. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. The following activities are carried out during each phase of a spiral model.

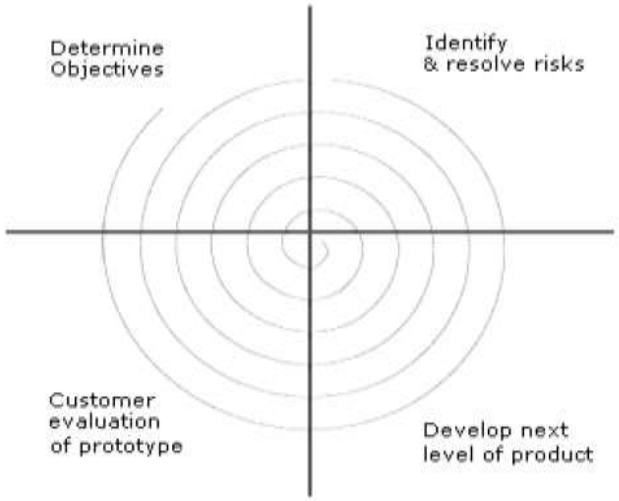


Fig. 2.2: Spiral Model

- First quadrant (Objective Setting):
- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.
- Second Quadrant (Risk Assessment and Reduction):
    - A detailed analysis is carried out for each identified project risk.
    - Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
- Third Quadrant (Development and Validation):

- Develop and validate the next level of the product after resolving the identified risks.

- Fourth Quadrant (Review and Planning):
  - Review the results achieved so far with the customer and plan the next iteration around the spiral.
  - Progressively more complete version of the software gets built with each iteration around the spiral.

OR

IV.

a) Describe the project size estimation metrics

Metrics for software project size estimation

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

- Lines of Code (LOC)

  LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored. Determining the LOC count at the end of a project is a very simple job.

  - LOC as a measure of problem size has several shortcomings:
    - LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways.
    - A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort.
    - LOC measure correlates poorly with the quality and efficiency of the code.

- LOC metric penalizes use of higher-level programming languages, code reuse, etc.
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities.

- Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data.

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

- Feature point metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

b) Explain data flow and control flow oriented design methods.

Control Flow-Oriented Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of a programs control flow structure. A programs control flow structure indicates the sequence in which the programs instructions are executed.

In order to help develop programs having good control flow structures, the flow charting technique was developed. Even today, the flow charting technique is being used to represent and design algorithms; though the popularity of flow charting represent and design programs has waned to a great extent due to emergence of more advanced techniques.

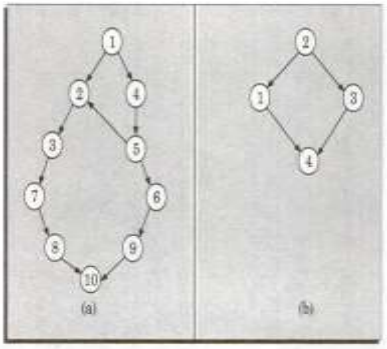


Figure 1.8: Control flow graphs of the programs in Figures 1.7(a) and (b).

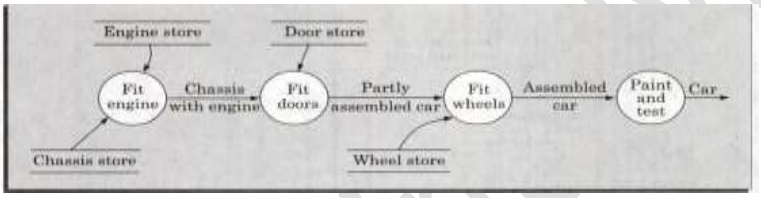
Data Flow-Oriented Design

The data flow-oriented techniques advocate that the major data items handled by a system must first be identified and then the processing required on these data items to produce the desired outputs should be determined.

The functions and the data items that are exchanged between the different functions are represented in a diagram known as a Data Flow Diagram (DFD). The program structure can be designed from the DFD representation of the problem.

DFDs:- DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example figure shows the data-flow representation of an automated car assembly plant. If you have never visited an automated car assembly plant, a brief description of an automated car assembly plant would be

necessary. In an automated car assembly plant, there are several processing stations which are located along side of a conveyor belt. Each workstation is specialized to do jobs such as fitting of wheels, fitting the engine, spray painting the car, etc. as the partially assembled product moves along the assembly line, different workstations perform their respective jobs on the partially assembled product. Each circle in the DFD model represents workstation(called a process or bubble). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be filtered from the corresponding stores, and as soon as the fitting work is complete passed on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in figure. A major advantage of DFDs is their simplicity.



UNIT – II

V.

a) Compare command language and menu based user interfaces.

- Command language based interfaces

A command language-based interface as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them in appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember. Thus, a command language-based interface can be made concise requiring minimal typing by the user.

Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

Command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them in. Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse. Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.

- Issues in designing a command language-based interface:-
  - ❖ The designer has to decide what mnemonics are to be used for the different commands.
  - ❖ The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences.
  - ❖ The designer has to decide whether it should be possible to compose primitive commands to form more complex commands.
- Menu-based interfaces

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to select from the menu. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.

- ❖ Scrolling menu:-
- ❖ Walking menu:-

❖ Hierarchical menu:-

b) List and explain the contents of SRS document.

Contents of SRS document

While documenting the user requirements, the requirements have to be properly categorized and documented in different sections of the document. The different user requirements can be categorized into the following categories:

- Functional Requirements

The functional requirements discuss the functionalities required by the users from the system. The system is considered to perform a set of high- level functions {fi}. The functional view of the system is shown in fig. Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data. The user can get some meaningful piece of work done using a high-level function.

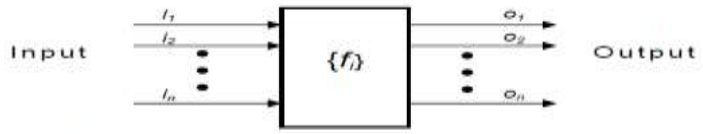


Fig. 3.1: View of a system performing a set of functions

- Nonfunctional requirements

The non-functional requirements deal with the characteristics of a system that cannot be expressed as functions. Non-functional requirements address aspects concerning maintainability, portability, usability, maximum number of concurrent users timing, and throughput. The non-functional requirements may also include reliability issues, accuracy of results, human-computer interface issues, and constraints on the system implementation.

The non-functional requirements are non-negotiable obligations that must be supported by the system. IEEE 870 standard lists four types of non-functional requirements: external interface requirements, performance requirements, constraints, and software system attributes.

- Goals of implementation

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to

the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

OR

VI.

a) Describe the steps involved in requirement gathering and analysis.

- Requirements Gathering

A requirement gathering is also popularly known as requirements elicitation. The analyst starts requirements gathering activity by collecting all information that could be useful to develop the system. Requirement gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of people and documents, and to form a clear understanding of a problem. This is especially so, if there are no working models of the problem. When the customer wants to automate some activity that is currently being done manually, then a working model of the system is available. Availability of a working model helps a great deal in gathering the requirements. If the project involves automating some existing procedures, then the task of the system analyst becomes a little a easier as he can immediately obtain the input and output data formats and the details of the operational procedures. For example while trying to automate activities of a certain office, one would have to study the input and output forms and then understand how the outputs are produced from the input data. However if the undertaken project involves developing something new for which no working model exists, then the requirements gathering and analysis activities become all the more difficult. In the absence of a working system, much more imagination and creativity is required on the part of the system analyst.

- Requirements Analysis

After requirement s gathering is complete, the analyst analyzes the gathered requirements to clearly understand the exact customer requirements and to weed out any problems in the gathered requirements.

The main purpose of the requirements analysis activity is to analyze the collected information to obtain a clear understanding of the product to be developed, with a view to

removing all ambiguities, incompleteness, and inconsistencies from the initial customer perception of the problem.

- Anomaly

An anomaly is an ambiguity in the requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in the requirements can lead to the development of incorrect systems, since an anomalous requirement can be interpreted in the several ways.

- Inconsistency

Two requirements are said to be inconsistent, if one of the requirements contradicts the other, or two-end users of the system give inconsistent description of the requirement.

- Incompleteness

An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be realized by the customer much later, possibly while using the product.

b) Explain the different types of cohesiveness.

- Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.
- Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.
- Temporal cohesion: When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion.
- Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

- Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack
- Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.
- Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function.

UNIT – III

VII.

a) Summarize code walk through and code inspection.

Code Walk Troughs:-

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

Even though a code walks through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following.

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.

- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection:-

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

b) Compare software and hardware reliability.

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase).

The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 13.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed.

Fig. 13.1: Change in failure rate of a product

OR

VIII.

a) Comment on terms

i. Software reuse

ii. Software maintenance

i. <u>Software reuse</u>

Software products are expensive. Software project managers are worried about the high cost of software development and are desperately look for ways to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality.

It is important to know about the kinds of the artifacts associated with software development that 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 and Knowledge.

The following are some of the basic issues that must be clearly understood for starting any reuse program.

- Component creation
- Component indexing and storing
- Component search
- Component understanding
- Component adaptation
- Repository maintenance

A promising approach that is being adopted by many organizations is to introduce a building block approach into the software development process. For this, the reusable components need to be identified after every development project is completed. The reusability of the identified components has to be enhanced and these have to be catalogued into component library. It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components. Domain analysis is a promising approach to identify reusable components.

ii. Software maintenance

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

There are basically three types of software maintenance. These are:

- Corrective: Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.

- Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities ofthe system according to customer demands, or to enhance the performance of the system.

b) Explain black box testing.

Black box testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class portioning:-

  In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

  1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
  2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

- Boundary value analysis:-

  A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different

equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <.

Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes. To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values no boundary value test cases can be defined. For equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then boundary value test suite is {0, 1, 10, 11}.

UNIT – IV

IX.

a) Summarize CASE support in software life cycle.

- 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 important features of a prototyping CASE tool 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 should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, prototyping CASE tool should support theuser to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.

- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.
- The run time system of prototype should support mock runs of the actual system and management of the input and output data.

- Structured analysis and design

  Several diagramming techniques are used for structured analysis and structured design. A CASE tool should support one or more of the structured analysis and design techniques. It should support effortlessly drawing analysis and design diagrams. It should support drawing for fairly complex diagrams, preferably through a hierarchy of levels.The CASE tool should provide easy navigation through the different levels and through the design and analysis.The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Whenever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there arises heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

- Code generation

  As far as code generation is concerned, the general expectation of 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, author name and the date of creation in some selectable format.The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular languages.It should generate database tables for relational database management systems.The tool should generate code for user interface from prototype definition for X window and MS window based applications.

- Test case generation

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

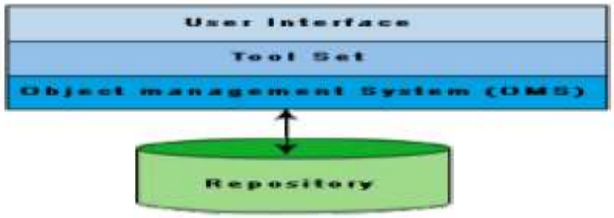
  - It should support both design and requirement testing.

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

b) Explain the architecture of a typical CASE environment.

- Architecture of a CASE environment

The architecture of a typical modern CASE environment is shown diagrammatically in fig. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository.



- 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 (OMS) 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 such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping into the underlying storage management system.

OR

X. Illustrate a case study to build a software that automates the banking system including deposit and withdrawal highlight design and testing phases.

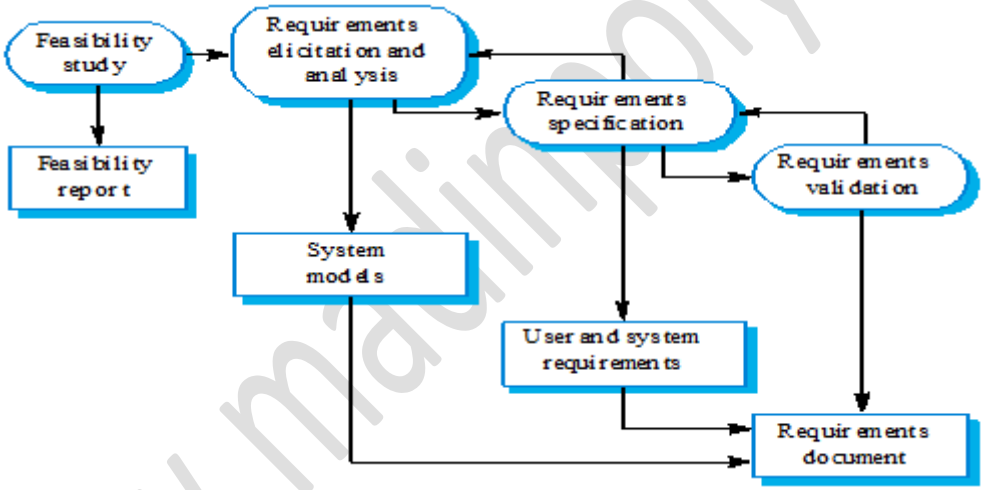
Objectives

- To describe the principal requirements engineering activities and their relationships
- To introduce techniques for requirements elicitation and analysis

- To describe requirements validation and the role of requirements reviews
- To discuss the role of requirements management in support of other requirements engineering processes

Requirements engineering processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
    - Requirements elicitation;
    - Requirements analysis;
    - Requirements validation;
    - Requirements management.



Feasibility studies

- A feasibility study decides whether or not the proposed system is worthwhile or doable.
- A short focused study that checks
    - If the system contributes to organisational objectives;
    - If the system can be engineered using current technology and within budget;
    - If the system can be integrated with other systems that are used.

Feasibility study implementation

- Based on information assessment (what is required), information collection and report writing.
- Questions for people in the organisation
    - What if the system wasn't implemented?
    - What are current process problems?
    - How will the proposed system help?
    - What will be the integration problems?
    - Is new technology needed? What skills?
    - What facilities must be supported by the proposed system?

Requirements

- Open an account for a customer (savings or chequing)
- Deposit Withdraw
- Display details of an account Change LOC
- Produce monthly statements Print a list of customers

Ambiguities

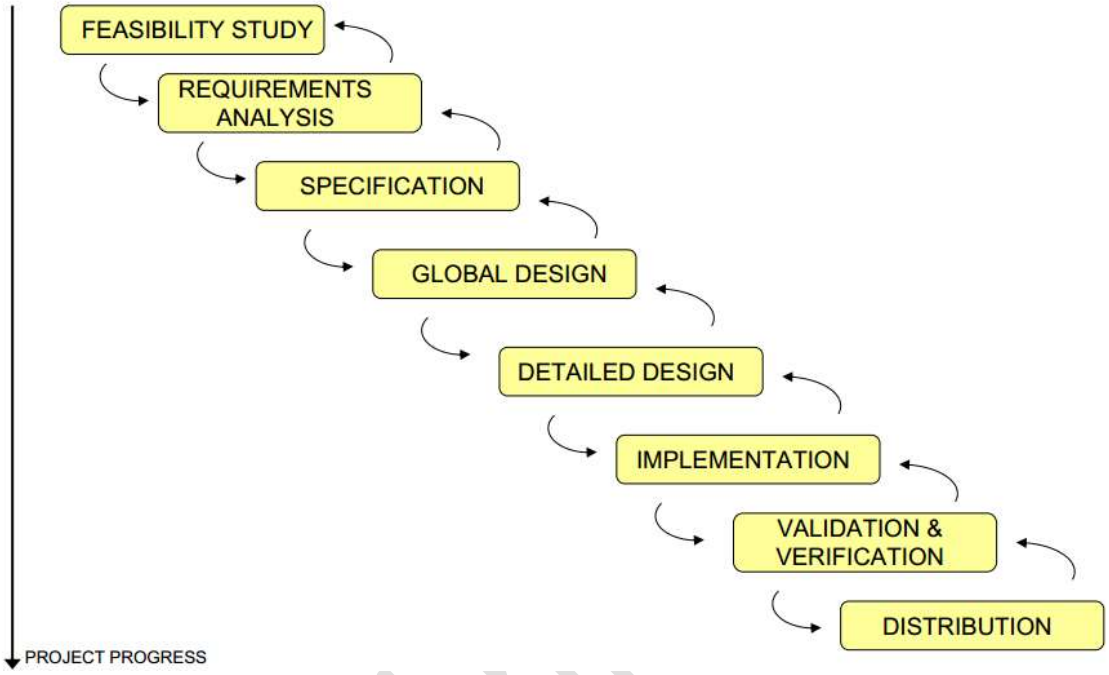
What is the difference between savings and chequing?

CASE STUDY 1

How should we go from Requirements to Code?

- Two basic approaches Plan-driven (waterfall type models)
- Agile (incremental approaches)

The waterfall model of the lifecycle



Problem for the waterfall

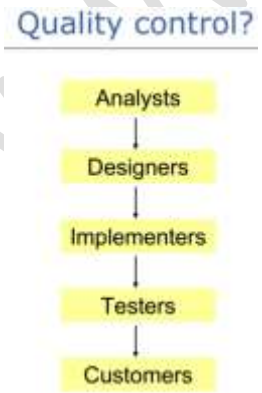
Late appearance of actual code.

Lack of support for requirements change — and more generally for extendibility and reusability. Lack of support for the maintenance activity (70% of software costs?).

Division of labor hampering Total Quality Management.

Impedance mismatches.

Highly synchronous model.



Use Case – Open Account

- Actors: Customer (initiator), Teller
- Overview: the customer applies for and is granted a new bank account with some initial approved line of credit. The teller logs on to the system, selects open account , and supplies the new account details. The system prints a copy of the contents of the new account with the date, which the teller hands to the customer.

Test Driven Development

1. Write a little Test
   a. Test will not work initially (there is no code)
   b. Might not even compile
2. Make the Test work quickly
   a. Commit whatever sins are necessary in the process (e.g. duplication in the code)
3. Refactor
   a. Eliminate the duplication created in merely getting the test to work

Open Account Use case (revisited)

- Actors: Customer (initiator), Teller
- Overview: the customer applies for and is granted a new bank account with some initial approved line of credit. The teller logs on to the system, selects open account , and supplies the new account details. The system prints a copy of the contents of the new account with the date, which the teller hands to the customer.

Withdraw Use Case

- Actors: Customer (initiator) and Teller
- Overview: The customer provides the teller with an account number and a withdrawal amount. The teller selects `withdraw-request' from the system menu, and enters the data. The System debits the account by the requested withdrawal amount, and prints a receipt with the date and the amount, which the teller gives to the customer.