Module: 02

Introduction

Lecture: 1

Motivation:

The following are the motivational factors in requirement, analysis and design workflow

- The system specification should be understandable to the clients.
- The analysis model should be unambiguous such that the developers can easily interpret.
- The system design should address the design goals.

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Requirement Elicitation, Software requirement specification (SRS)	2	1
2	Data Flow Diagram(DFD), Feasibility Analysis, Cost- Benefit Analysis	2	1
3	Developing Use Cases (UML), Requirement Model – Scenario-based model	2	1
4	Class-based model, Behavioral model.	2	1

Learning Objective:

To get knowledge about the activities in different phases (requirements, analysis, and design) of software life cycle.

Abbreviations:

UML - Unified Modeling Language
JAD - Joint Application Development

8. Key Definitions:

1. **Workflow:** It is the organization of activities and artifacts. Each workflow detail represents a key skill that needs to be applied to perform effective requirements/analysis/design management.

2. **Reusability:** It is the likelihood a segment of source code that can be used again to add new functionalities with slight or no modification.

3. **Abstraction:** An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
4. **Extensibility:** Extensibility is the capacity to extend or stretch the functionality of the development environment — to add something to it that didn't exist there before. E.g, Inheritance
5. **Encapsulation:** It is the ability of an object to hide its data and methods from the rest of the world.
6. **Coupling:** The degree to which components depend on one another. There are two types of coupling, "tight" and "loose". Loose coupling is desirable for good software engineering but tight coupling may be necessary for maximum performance. Coupling is increased when the data exchanged between components becomes larger or more complex.
7. **Cohesion:** Cohesion is a measure of how strongly-related the functionality expressed by the source code of a software module is. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.
8. **Use case:** A use case in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in question.
9. **Actor:** An actor specifies a role played by a user or any other system that interacts with the subject.
10. **Use case diagram:** A use case diagram displays the relationship among actors and use cases. It is helpful in exposing requirements and planning the project. During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.
11. **Class diagram:** Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation.
12. **Sequence diagram:** A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart.
13. **Collaboration diagram:** A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.
14. **State chart diagram:** State chart diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.
15. **Activity diagram:** Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of

doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

Theoretical Background:
Theory

○ 3.1 Requirement Workflow:

This defines the set of activities during the requirements of software life cycle. Requirements can be classified as functional and non-functional. The requirements are gathered from customers. There are different techniques for eliciting the requirements which are explained below. When the requirements are agreed upon they are documented.

3.1.1 Functional and Non-functional Requirements:

Functional requirements describe what the system should do, i.e. the services provided for the users and for other systems. The functional requirements should include 1) everything that a user of the system would need to know regarding what the system does, and 2) everything that would concern any other system that has to interface to this system.
The functional requirements can be categorized as follows:

- What *inputs* the system should accept
- What *outputs* the system should produce
- What data the system should *store* that other systems might use
- What *computations* the system should perform
- The *timing and synchronization* of the above

Non-functional requirements are constraints that must be adhered to during development. One of the most important things about non-functional requirements is to make them verifiable. The categories of non-functional requirements can be divided into three groups. The first group of categories reflects the following quality attributes. These requirements constrain the design to meet specified levels of quality.

- Response time
- Throughput
- Resource usage
- Reliability
- Availability
- Recovery from failure
- Allowances for maintainability and enhancement
- Allowances for reusability

The second group of non-functional requirements categories constrains the environment and technology of the system. These requirements are as follows:

- Platform
- Technology to be used

○

The third group of non-functional requirements categories constrains the project plan and development methods. These requirements are as follows:

- Development process (methodology) to be used
- Cost and delivery date

3.1.2 Characteristics of requirements:

Requirements must be:

1. Correct
2. Complete
3. Consistent
4. Nonambiguous
5. Verifiable
6. Traceable

Each of these attributes is explained below:

Correct:
A requirement is correct if it describes something that a system must do or a constraint on the way it must be done.

Complete:
*A requirement is complete to the extent that all parts are present and each part is fully developed. The requirement set is complete if it contains all of the complete requirements***.**
It is not unusual in the early stages of analysis to mark some part of a requirement as either:

- to be done (TBD).
- do not know (DNK)

For example, in an early iteration we might identify and name an event including names for the stimulus and responses. But we might wait until later to fill in the content and structure of the stimulus and response messages. Eventually, all of the TBD's and DNK's must be replaced with the actual requirements.
It may be difficult to determine what is missing by looking at what is present, especially if the user inadvertently fails to mention a requirement. It will be easy for a user to describe their day to day needs but they may honestly forget about some event that only takes place once or twice a year.
Consistent:
A requirement is consistent if it does not conflict with another requirement.
Nonambiguous:
A requirement is non-ambiguous if there is only one interpretation of its meaning.
Some examples of ambiguities are:

- Invoices must be generated in a timely fashion.
- The software must be easy to transport to any operating system.

Since we often use English to describe requirements to system owners and users, it can be difficult to remove all ambiguities. A glossary that defines the precise meanings of terms may be helpful.
Verifiable:
A requirement is verifiable if and only if *there is a finite cost effective process that a person or machine can use to check that the as built system meets the requirement.*

Ambiguous requirements are not verifiable. Most qualitative (subjective) requirements are not verifiable. But most quantitative (objective) requirements are verifiable. Consider the following examples for requirements that are stated qualitatively and quantitatively:

Example 1:
- Qualitative (Not Verifiable): The new system must reduce the cost of producing client invoices.
- Quantitative (Verifiable): The new system must reduce the cost of producing client invoices by at least twenty percent.

Example 2:
- Qualitative (Not Verifiable): The new system must be easy to use.
- Quantitative (Verifiable): After one day of instruction and one week of use, users will be able to perform their tasks in the same amount of time as they currently use. After one month of use, system users must show a twenty percent improvement in their productivity.

Traceable:
An analysis requirement is traceable if it can be traced backward to a feasibility study, a white paper, meeting notes, or an interview. A design requirement is traceable if it can be traced backward to one or more essential requirements.
In reality, requirements should be traced in both directions. A backward trace can be used to ensure that each requirement is correct. A forward trace can be used to ensure that the set of requirements is complete.

3.1.3 Requirement Elicitation Techniques:

Requirements can be gathered from the various stakeholders, from other software systems, and from any documentation that might be available. The following are some of the techniques for gathering (also known as eliciting) and analyzing requirements.
i) Observation
In this technique documents are read and discussed with users. In its simplest form, observation means taking a notebook and 'shadowing' important potential users as they do their work, writing down everything they do. Users can also be asked to talk as they do their work, explaining what they are doing. Session videotaping can be done so that the requirements can be analyzed in more detail later. This technique is followed only in large systems since it is time consuming.
ii) Interviewing:
This is a widely used technique. A series of interviews should be conducted. The software engineering team should have as many members as possible for the interview. And the interview should be conducted for as many stakeholders as possible. Several hours should be spent for each interview and gap should be there between interviews to analyze the things heard and to think of additional questions. An extensive list of questions should be prepared and the following are some types of questions to be remembered:
_ Specific details such as maximums, minimums and possible changes anticipated

– Stakeholder's vision for the future

– Users or customers are asked to give alternative ideas if they give concrete ideas for their view of the system

– Minimally acceptable solution to the problem

– Other sources of information (E.g. Stakeholders may have interesting documents or may know someone with useful knowledge)

– The interviewee is asked to draw diagrams. Diagrams can be a focal point to stimulate improved information gathering.

An interviewer should cultivate listening skills and empathy for users and customers. Following the interview process, the information found should be summarized and shared among all the customers along with a request for comments.

iii) Brainstorming:

This is an effective way to gather information from a group of people. The general idea here is that the attendees are arranged around a table. An experienced moderator should be appointed to lead the session.

A key step in the process is to decide on a 'trigger question'. A trigger question is one for which the participants can provide simple one-line answers that are more than just numbers or yes/no responses. An example is 'What features are important in the system?' The trigger question can be determined by the person who called the meeting, by the moderator, by a quick discussion, or by brainstorming followed by a vote. Each participant is asked to write an answer for the trigger question and pass paper to his/her neighbor. i.e. the ideas are passed clockwise round the table to stimulate new ideas. This is shown by the following figure:

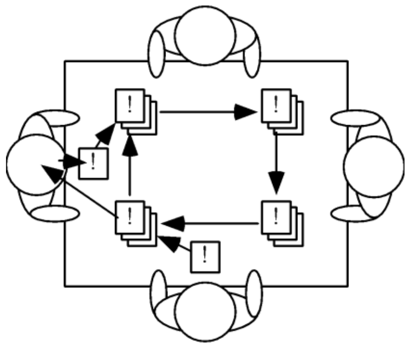


Figure 3.1: Passing ideas around the table to stimulate new ideas

Joint Application Development (JAD) is another technique based on intensive brainstorming sessions. In JAD, the developers meet with customers and users in a secluded location for a period of three to five days. The entire time is spent working together to define the requirements. The final output should be a written requirements document.

iv) Prototyping

A prototype is a program that is rapidly implemented and only contains a small part of the anticipated functionality of a complete system. This is done to obtain early feedback about a software engineer's ideas.

The simplest kind of prototype is a paper prototype of the user interface. This is a set of pictures of the system that are shown to users in sequence, to explain what would happen when the system runs.

The most common type of prototype is a 'mock-up' of the system's user interface, created

using a rapid prototyping language. Rapid prototyping languages allow one to create code very quickly to display the important parts of a user interface. This has weaknesses which include inefficiency and inflexible designs. So this should be used only as a requirements gathering tool instead of directly turning into a final system.
v) Informal use case analysis
Use case analysis is a systematic approach to working out what users should be able to do with the software being developed.
The first step in use case analysis is to determine the classes of users (or other systems) that will use the facilities of the particular system. These are called actors.
The second step in use case analysis is to determine the tasks that each actor will need to do with the system. Each task is called a use case.

3.1.4 Requirement Documentation- Use case Specification:

To perform good software engineering, it is always appropriate to write down requirements. Choosing the right level of detail requires careful balancing, because too little detail can result in a system that does not adequately solve the problem, whereas too much detail can be a waste of time and can make the development process too inflexible.
Requirements documents for large systems are normally arranged in a hierarchy. There is a top-level document describing the overall system, its subsystems and how the subsystems interact. Then there are separate documents describing each subsystem, and sometimes each sub-subsystem. The hierarchy of requirements documents is shown below:

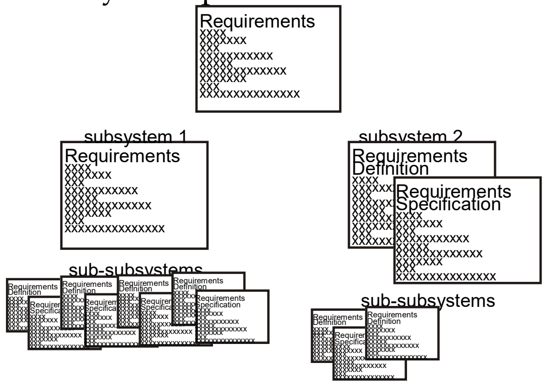


Figure 3.2: Hierarchy of requirements documents

The following are the decision factors for what type of document to produce and how much detail should be provided:

- The size of the system
- The need to interface to other systems
- The target audience
- The contractual arrangements for development
- The stage in requirements gathering
- The level of experience with the domain and the technology
- The cost incurred if the requirements are faulty

Requirements documents are given different names in different organizations. The term ‘requirements definition’ normally refers to a less detailed and the term ‘requirements specification’ refers to a more detailed and precise document.
Use case Specification:

A use case is a typical sequence of actions that an actor performs in order to complete a given task. The objective of use case analysis is to model the system from the point of view of how users or other systems interact with this system when trying to achieve their objectives. A use case model consists of a set of use cases, and an optional description or diagram indicating how they are related.
Description of a single use case:
The following is the description of a complete use case

A. **Name.** A short descriptive name should be given to the use case

B. **Actors.** Actor or actors who can perform this use case are listed

C. **Goals.** Explanation about what the actor or actors are trying to achieve

D. **Preconditions.** Listing of conditions that must be true before an actor can initiate this use case

E. **Summary.** Listing the events that occur as the actor or actors perform the use case

F. **Related use cases.** Listing the use cases that may be generalizations, specializations, extensions or inclusions of this use case.

G. **Steps.** Description about each step of the use case using a two-column format, with the left column showing the actions taken by the actor, and the right column showing the system's responses.

H. **Postconditions.** State of the system following the completion of this use case

In general, a use case should cover the full sequence of steps from the beginning of a task until the end.

Activity Diagram:
An activity diagram is used to understand the flow of work that an object or component performs. It can also be used to visualize the interaction between different use cases. An activity diagram is similar to state diagram. The difference is that in a state diagram, most transitions are caused by external events, whereas in an activity diagram, most transitions are caused by internal events, such as the completion of an activity.
One of the strengths of activity diagram is the representation of concurrent activities. Concurrency is shown using forks, joins, and rendezvous, all three of which are represented as short lines, at which transitions can start and end.

- A *fork* has one incoming transition and multiple outgoing transitions. The result is that execution splits into two concurrent threads.

- A *join* has multiple incoming transitions and one outgoing transition. The outgoing transition will be taken when all incoming transitions have been triggered. The incoming transitions must be triggered in separate threads.

- A *rendezvous* has multiple incoming and multiple outgoing transitions. Once all the incoming transitions are triggered, the system takes all the outgoing transitions, each in a separate thread.

The following figure shows an example activity diagram for the process of registering in a course section.

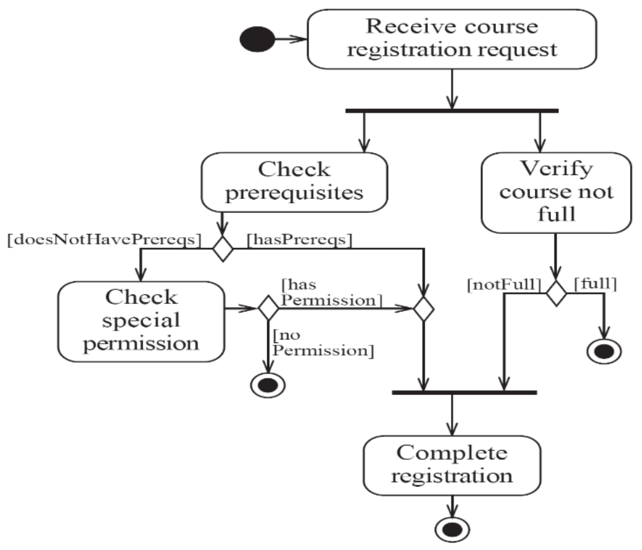


Figure 3.3: Activity diagram of the registration process

Here the black circle represents the start state and the black circle with a ring around represents an end state. The example activity diagram depicts fork and join.
While state diagrams typically show states and events concerning only one class, activity diagrams are most often associated with several classes. The partition of activities among the existing classes can be explicitly shown in an activity diagram by the introduction of swimlanes. These are boxes that form columns, each containing activities associated with one class. The following figure shows how the activities of the above diagram can be allocated to two different classes.

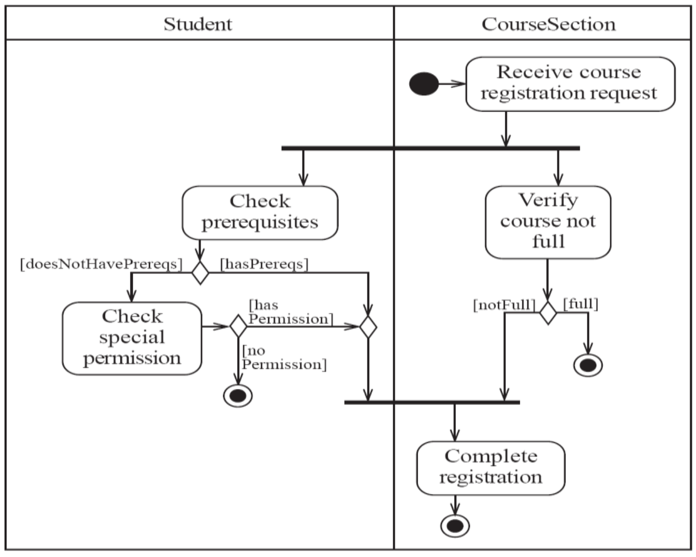


Figure 3.4: Activity diagram with swimlanes

3.2 Analysis Workflow

This defines the set of activities involved in the analysis phase of software life cycle. Analysis is done in two stages: static analysis and dynamic analysis. The activities involved in static and dynamic analysis are explained below.

3.2.1 Static Analysis:

Static analysis represents the system under development from the user's point of view. This focuses on the individual concepts that are manipulated by the system, their properties and their relationships. Static analysis depicted with UML class diagrams, includes classes, attributes, and operations. This analysis model is a visual dictionary of the main concepts visible to the user. The following is the illustration of the class diagram.

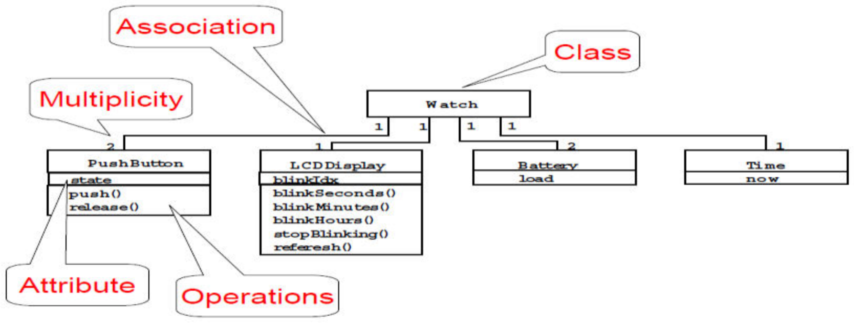


Figure 3.5: Illustration of class diagram

3.2.1.1 Identifying Objects:

Object identification is the process of determining the set of peer subsystems/objects whose behavioral interactions will solve the problem posed by the system requirements. Since it is known that many real systems cannot be accurately modeled computationally (Rosen, 1985, *Life Itself*), it is clear that there may be no 'right' set of objects that works perfectly.

The following are the points to be considered while identifying objects

- Look at the system as a whole, and ask what data indicates the state of the system.
- Those data items are the natural objects of the system. (They might *not* be the objects of your final design. They may actually be subsystems or even virtual objects created by the interaction of a number of real objects.)
- Avoid complex interdependencies. Instead, encapsulate.
- Avoid giving too much responsibility to an object—delegate!
- Try not to have a large number of top-level objects or subsystems. A list of objects you can paper your office with is too long. Instead, increase the level of abstraction.
- Try not to have too few top-level objects as well. Aim for 5 to 9, with a minimum of 3 and maximum of 15.

Methods of identifying objects:

There are basically three approaches to initial object identification:

1. Model the system requirements—define the system requirements in terms of objects and object interactions and use those in the design.

2. Transform the system requirements—define the system requirements in terms of objects and object interactions and *transform* those to reflect architectural and design constraints.

3. Use a pattern language—define the design in terms of 'patterns' known to solve system requirements previously defined in some traditional fashion.

Modeling the system requirements

This approach is common, since most systems automate some existing process. The objects in the requirements are directly modeled using the features of the language, and the design may be automatically generated from the requirements.

Pros:

- Easy to envision and hence manage
- Minimizes design effort
- Usually easy to test

Cons:

- The system requirements may not lend themselves to object-orientation
- The resulting design may not solve the real problem because it fails to consider implementation constraints
- Testing may be infeasible

Transforming the system requirements

This approach is also standard. There are two ways to do it:

1. Ignore implementation issues in defining the system requirements. The design is corrected incrementally by refactoring in response to problems while directly implementing the requirements.

2. Write the system requirements with implementation in mind. In this case, the transformation begins early, and is continued as the requirements are implemented. (Historically, this has worked better.)

Pros:

1. Addresses the problems inherent in modeling the system requirements.
2. Potentially less refactoring.

Cons:

1. Incremental correction may not converge on a working system.
2. Requires expertise in both system requirements and solutions during system requirements definition (i.e., early). This can be risky, particularly if the solution space is not understood.
3. May not address user requirements and so fails user evaluation.

Using a Pattern Language

This is invented by Christopher Alexander, a noted architect. In his dissertation research, he investigated formal techniques for identifying a design from the relationships between requirements and design constraints. This failed for reasons already familiar, so he began to investigate pattern languages—ways of describing a design directly in terms of patterns:

- problems,
- general solutions to those problems, and

– the underlying trade-offs.

Pros:

– The resulting system actually solves a problem.

– Expert system architects seem to think that way.

Cons:

– Pattern-based designs tend to be rigid.

– Non-experts tend to produce rambling architectures without unifying themes. That makes refactoring and maintenance very difficult.

Boundary, Control, Entity Objects:

Entity objects represent the persistent information tracked by the system. Boundary objects represent the interactions between the actors and the system. Control objects are in charge of realizing use cases.

Identifying Boundary, Control, Entity Objects:

The following are the points to be considered while identifying boundary objects

- Identify user interface controls that the user needs to initiate the use case
- Identify forms the user needs to enter data into the system
- Identify notices and messages the system uses to respond to the user
- When multiple actors are involved in a use case, identify actor terminals to refer to the user interface under consideration
- Do not model the visual aspects of the interface with boundary objects
- Always use the end user's terms for describing interfaces; do not use terms from the solution or implementation domains

The following are the points to be considered while identifying entity objects

- Terms that developers or users need to clarify in order to understand the use case
- Recurring nouns in the use cases
- Real-world entities that the system needs to track
- Real-world activities that the system needs to track
- Data sources or sinks

The following are the points to be considered while identifying control objects

- Identify one control object per use case
- Identify one control object per actor in the use case
- The life span of a control object should cover the extent of the use case or the extent of a user session. If it is difficult to identify the beginning and the end of a control object activation, the corresponding use case probably does not have well-defined entry and exit conditions

For explaining the above said objects, ATM can be taken as an example. The following is the requirements statement for ATM (given in box)

The automated teller machine (ATM) has a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash (in multiples of $20), a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned - except as noted below.

The ATM must be able to provide the following services to the customer:

A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of $20.00. Approval must be obtained from the bank before cash is dispensed.

A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.

A customer must be able to make a transfer of money between any two accounts linked to the card.

A customer must be able to make a balance inquiry of any account linked to the card.

A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.
If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.
The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).
The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.
The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and

The following is the use case diagram for the ATM example

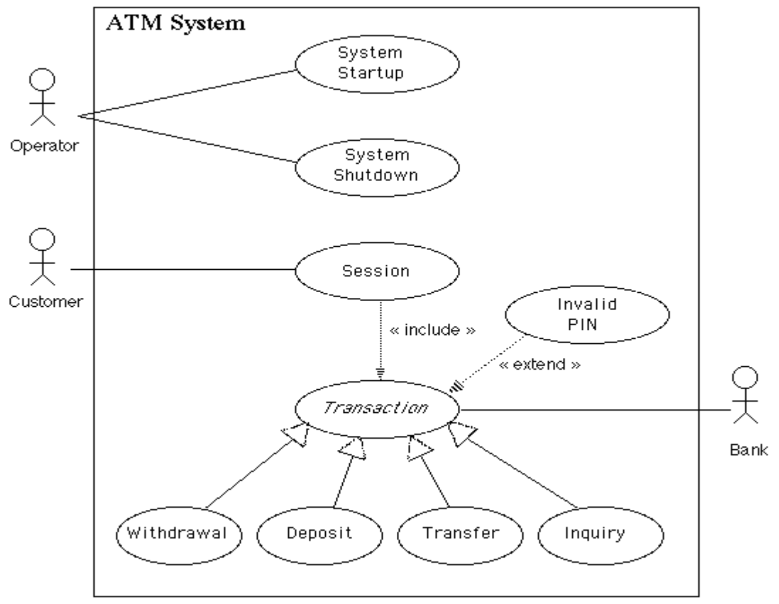


Figure 3.6: Use case diagram for ATM example

From the above use cases, the following objects can be identified

- A controller object representing the ATM itself (managing the boundary objects listed below)
- Boundary objects representing the individual component parts of the ATM:
  - Operator panel.
  - Card reader.
  - Customer console, consisting of a display and keyboard.
  - Network connection to the bank.
  - Cash dispenser.
  - Envelope acceptor.
  - Receipt printer.
- Controller objects corresponding to use cases. (Note: class ATM can handle the Startup and Shutdown use cases itself, so these do not give rise to separate objects)
  - Session
  - Transaction (abstract generalization, responsible for common features, with concrete specializations responsible for type-specific portions)
- An entity object representing the information encoded on the ATM card inserted by customer.
- An entity object representing the log of transactions maintained by the machine.

This leads to the following diagram of analysis classes

○

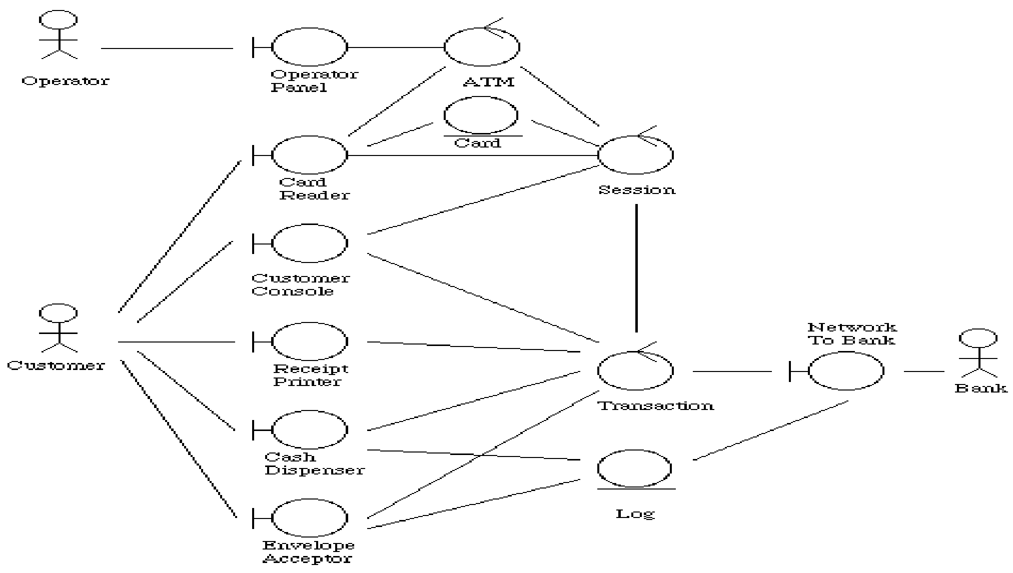


Figure 3.7: Use case diagram illustrating the analysis classes for ATM example

3.2.2 Dynamic Analysis:

The dynamic analysis or dynamic model focuses on the behavior of the system. It is depicted with sequence diagrams and with state charts. The dynamic model serves to assign responsibilities to individual classes and, in the process, to identify new classes, associations, and attributes to be added to the analysis object model i.e. static analysis.

3.2.2.1 Identifying Interaction:

Interaction is identified from the particular use case. The steps of a use case are performed by the communication between a set of actors and objects. Interaction diagrams (i.e. sequence diagram, collaboration diagram) and state charts help to visualize this.

Sequence diagram:

A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task. The objects are arranged horizontally across the diagram. An actor that initiates the interaction is often shown on the left. The vertical dimension represents time. A vertical line, called a *lifeline*, is attached to each object or actor. The lifeline becomes a broad box, called an *activation box* during the *live activation* period. During the live activation period the object is said to be performing computations. This live activation is shown by cross sign in the diagram. A message is represented as an arrow between activation boxes of the sender and receiver. A message is labelled and can have an argument list and a return value. The following sequence diagram depicts the transaction use case.

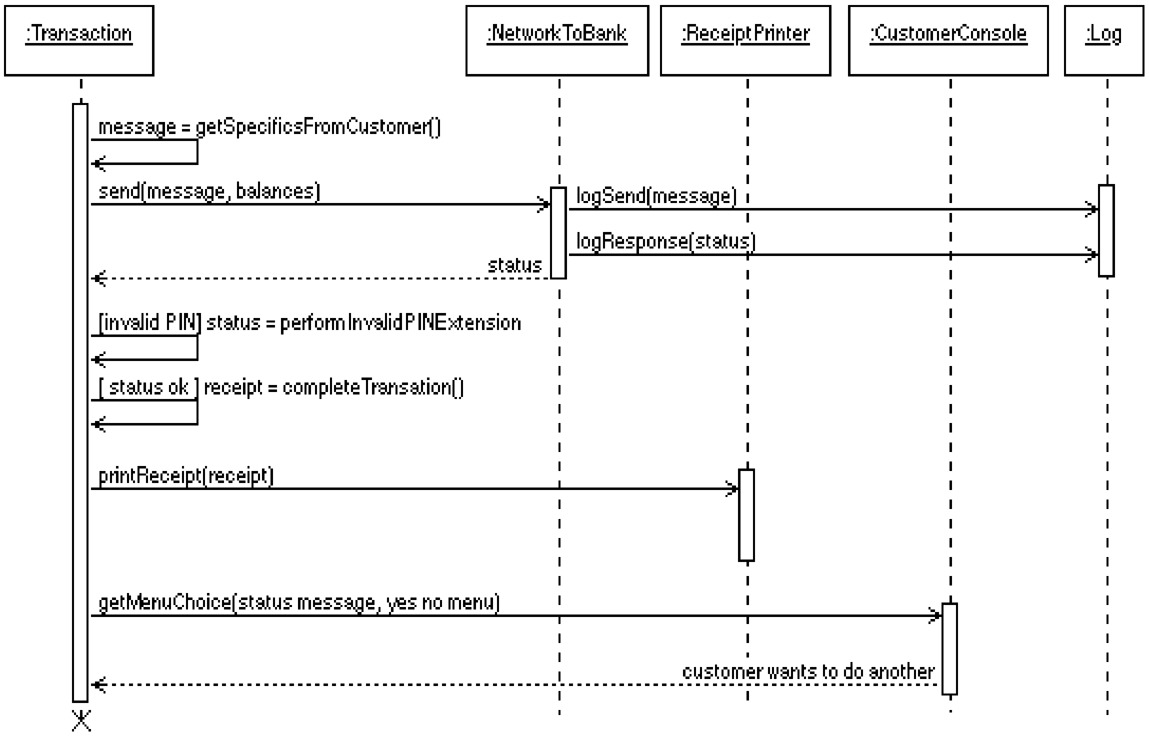

:Transaction
:NetworkToBank
:ReceiptPrinter
:CustomerConsole
:Log
message = getSpecificsFromCustomer()
send(message, balances)
logSend(message)
logResponse(status)
status
[invalid PIN] status = performInvalidPINExtension
[status ok] receipt = completeTransation()
printReceipt(receipt)
getMenuChoice(status message, yes no menu)
customer wants to do another

Figure 3.8: Example sequence diagram

Collaboration diagram:

Collaboration diagrams emphasise how the objects collaborate in order to realize an interaction. A collaboration diagram is a graph with a set of objects and actors as the vertices. Communication links are added between objects and messages are attached to these links. Messages are shown as arrows labelled with the message name. Time ordering is indicated by prefixing the message with some numbering scheme. Following are the collaboration diagrams for withdrawal and deposit use case.

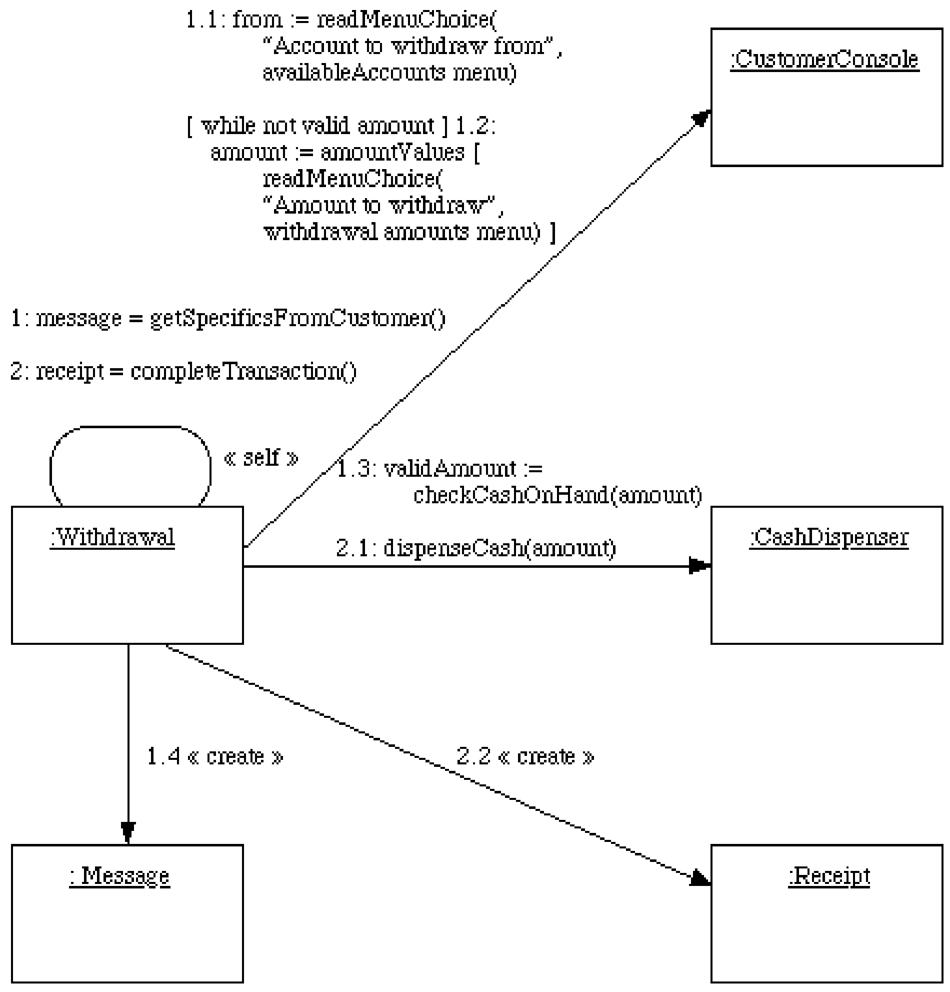


Figure 3.9: Collaboration diagram for withdrawal use case

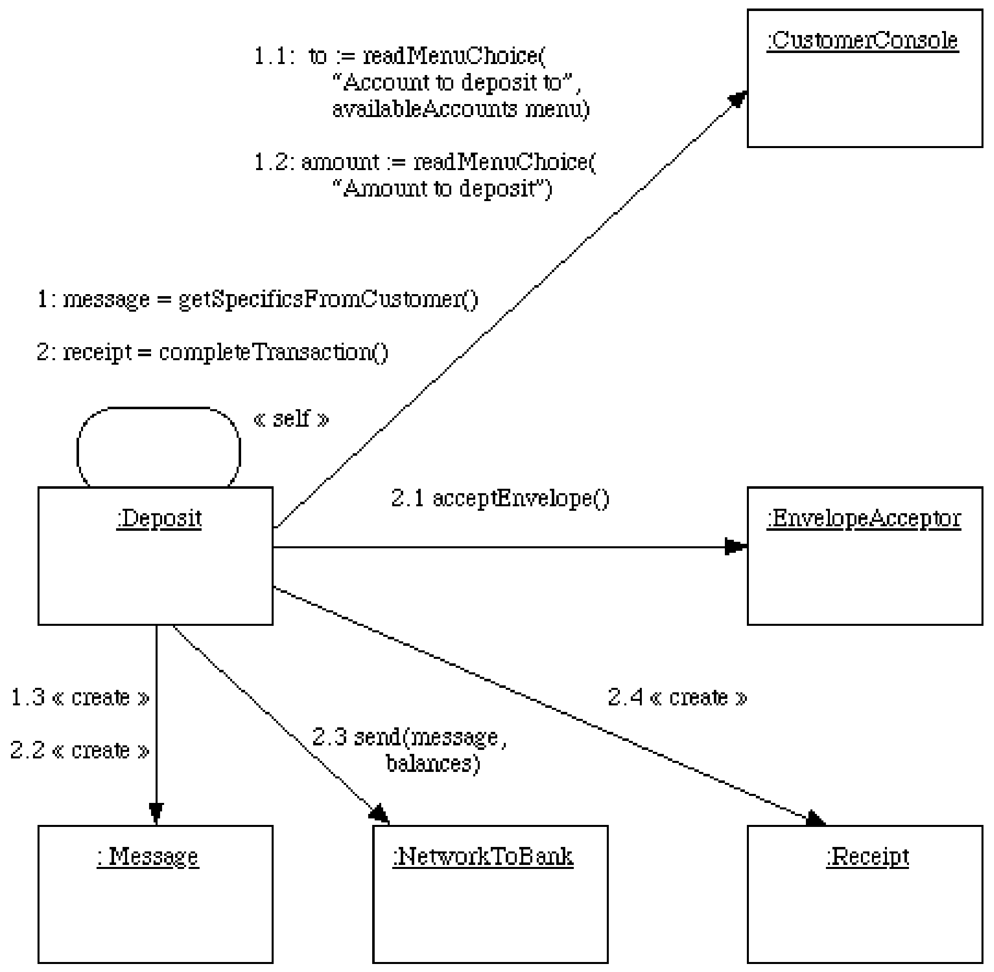

1.1: to := readMenuChoice("Account to deposit to", availableAccounts menu)
1.2: amount := readMenuChoice("Amount to deposit")
1: message = getSpecificsFromCustomer()
2: receipt = completeTransaction()
« self »
:CustomerConsole
:Deposit
2.1 acceptEnvelope()
:EnvelopeAcceptor
1.3 « create »
2.2 « create »
2.3 send(message, balances)
2.4 « create »
: Message
:NetworkToBank
:Receipt

Figure 3.10: Collaboration diagram for deposit use case

State Chart diagram:
A state diagram describes the externally visible behaviour of a system or of an individual object. At any given point in time, the system or object is in a certain state. Being in a state means that it will behave in a specific way in response to any events that occur. Some events will cause the system to change state. In the new state, the system will behave in a different way to events. A state diagram is a directed graph where the nodes are states and the arcs are transitions.
At any given point in time, the system is in one state. It will remain in this state until an event occurs that causes it to change state. A state is represented by a rounded rectangle containing the name of the state. Two other special states are start state and end state. Start state is represented by a black circle, and an end state is represented by a circle with a ring around it.
A transition represents a change of state in response to an event. It is considered to occur instantaneously. The label on each transition is the event that causes the change of state. The following is the state chart that depicts one session in ATM.

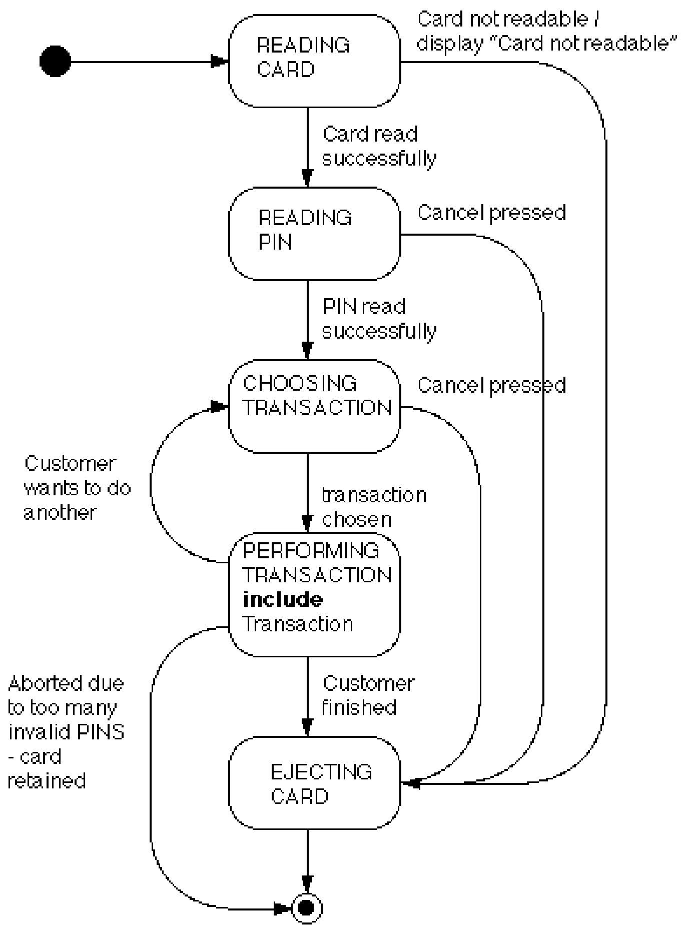

READING CARD
Card not readable / display "Card not readable"
Card read successfully
READING PIN
Cancel pressed
PIN read successfully
CHOOSING TRANSACTION
Cancel pressed
Customer wants to do another
transaction chosen
PERFORMING TRANSACTION include Transaction
Aborted due to too many invalid PINS - card retained
Customer finished
EJECTING CARD

Figure 3.11: Example state chart diagram

4. Objective Questions:

1. Use-case actors are always people, never system devices
a. True
b. False
2. Which of the following UML diagram is not used in creating system analysis model
a. Activity diagram
b. Class diagram
c. Data flow diagram
d. State diagram
3. Which of the following is not an objective for building an analysis model
a. define set of software requirements that can be validated
b. describe customer requirements
c. develop an abbreviated solution for the problem
d. establish basis for software design
4. Which of these is not an element of object oriented analysis model
a. Behavioral elements
b. Class-based elements
c. Data elements
d. Scenario-based elements
5. A generalized description of a collection of similar objects is a
a. class

b. instance
c. subclass
d. super class
6. UML activity diagrams are useful in representing which analysis model elements?
a. Behavioral elements
b. Class-based elements
c. Flow-based elements
d. Scenario-based elements
7. The state diagram
a. depicts the relationship between data objects
b. depicts function that transform the data flow
c. indicates how data are transformed by the system
d. indicates system reactions to external events

5. Subjective Questions:

1. List the characteristics of requirements and explain the various requirement elicitation techniques?
2. Explain the boundary, control and entity objects?
3. Explain sequence and collaboration diagrams?
4. Write a note on state chart diagram?
5. Write a note on activity diagram?

5. University Questions:

1. As a developer, transform the following customer's mission statement into a complete, unambiguous description of the system using usecase and class diagram. Prime Care Rental Company Mission Statement-

Since we automated the tracing of cars at our stores-Using bar codes, counter top terminals and laser readers. We have seen many benefits: the productivity of our rental assistants has increased 20%, car rarely go missing and our customer base has grown strongly.

The management feels that the Internet offers further existing opportunities for increasing efficiency and reducing costs. For example rather than printing catalogs of available cars, we could make catalog available to every Internet surfer for browsing online. For privileged customers, we could provide extra services, such as reservations, at the click of a button. Our target saving in this area is a reduction of 15% in the cost of running each stores.

Within two years, using the full power of e-commerce, we aim to offer all our services via web browser, with delivery and pick-up at the customer's home, thus achieving our ultimate goal of the virtual rental company, with minimum running costs relative to walk-in stores. [May 2010] [20 marks]

References:

1. Roger Pressman, Software Engineering: A Practitioners Approach, (6th Edition), McGraw Hill, 1997.

Objective Questions:

1. Software is a product and can be manufactured using the same technologies used for other engineering artifacts.
A) True
B) False
2. Software deteriorates rather than wears out because…
A) Software suffers from exposure to hostile environments
B) Defects are more likely to arise after software has been used often
C) Multiple change requests introduce errors in component interactions
D) Software spare parts become harder to order
3. Agility is nothing more than the ability of a project team to respond rapidly to change.
A) True
B) False
4. Which of the items listed below is not one of the software engineering layers?
A) Process
B) Manufacturing
C) Methods
D) Tools
5. Software engineering umbrella activities are only applied during the initial phases of software development projects.
A) True
B) False
6. Process models are described as agile because they
A) eliminate the need for cumbersome documentation
B) emphasize maneuverability and adaptability
C) do not waste development time on planning activities
D) make extensive use of prototype creation
7. Which of these terms are level names in the Capability Maturity Model?
A) Performed
B) Repeated
C) Reused
D) Optimized
E) both a and d
8. Software processes can be constructed out of pre-existing software patterns to best meet the needs of a software project.
A) True
B) False
9. Which of these are standards for assessing software processes?
A) SEI
B) SPICE
C) ISO 19002
D) ISO 9001
E) Both b and d

10. The best software process model is one that has been created by the people who will actually be doing the work.

A) True
B) False

(Ans : 1-B,2-C,3-A,4-B,5-B,6-C,7-E,8-E,9-A,10-A)

Short Answer Questions:

1. Explain:

a) Waterfall model

Ans: Refer page.no. 6

2. State the difference between process & project metrics? Describe process metric in detail?

Ans: Refer page.no.18

3. Explain Agile development.

Ans: Refer page .no.17

Long Questions/Answers, Set of Questions for FA/CE/IA/ESE:

1. Compare Waterfall model and Spiral model of Software development. [May 2010] [10 marks]

Ans:

Waterfall

– Testing at end
– Changes cannot be incorporated in between
– Cascade effect
– Document driven
– Well understood products as requirements cannot change
– Linear sequential model or classical life cycle model

Spiral

– Testing is done at every step
– Iterations
– Not completely document driven
– the requirements are not defined in detail

2. Explain the Open Source software life cycle model. [May 2010] [10 marks]

Ans:

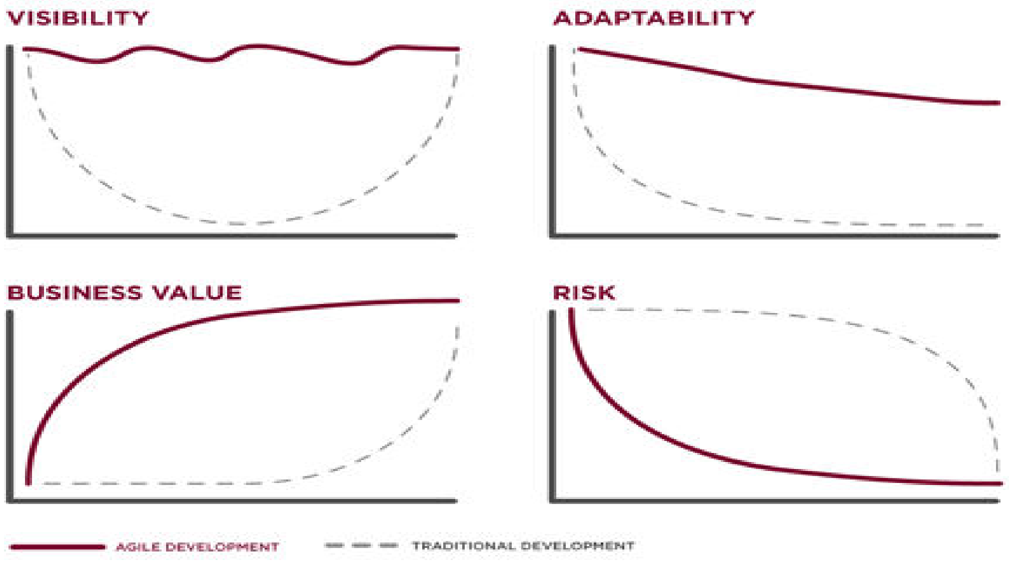
Open source is used by the open source initiative to determine whether or not a software license can be considered open source.

Distribution terms of open source

i. Free redistribution
ii. Source code
iii. Derived works
iv. Integrity of the author's source code
v. No discrimination against fields of endeavor

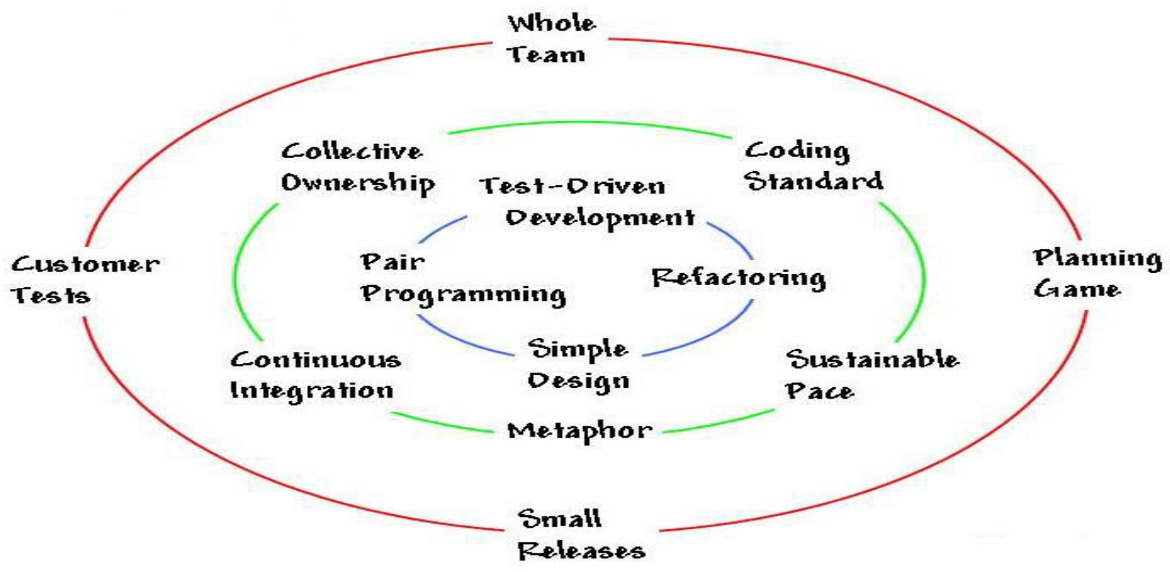
vi. Distribution of license
vii. License must not be specific to a product
viii. License must not restrict other software
ix. License must be technology-central

3. What are the advantages of agile methodology? [May 2010] [5 marks]
Ans:
Agile development delivers increased value, visibility, and adaptability much earlier in the life cycle, significantly reducing project risk.



4. What is an Agile Process? Explain any one Agile Process model with its advantages and disadvantages. [Nov. 2010] [10 marks]
Ans:
Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams
The different Agile Process models are
- Agile Unified Process (AUP)
- Extreme Programming (XP)
- Scrum

Extreme Programming:
Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.



XP Practices

- Core Practices: Whole Team

In Extreme Programming, every contributor to the project is an integral part of the "Whole Team". The team forms around a business representative called "the Customer", who sits with the team and works with them daily.

- Core Practices: Planning Game, Small Releases, Customer Tests

Extreme Programming teams use a simple form of planning and tracking to decide what should be done next and to predict when the project will be done. Focused on business value, the team produces the software in a series of small fully-integrated releases that pass all the tests the Customer has defined.

- Core Practices: Simple Design, Pair Programming, Test-Driven Development, Design Improvement

Extreme Programmers work together in pairs and as a group, with simple design and obsessively tested code, improving the design continually to keep it always just right for the current needs.

- Core Practices: Continuous Integration, Collective Code Ownership, Coding Standard

The Extreme Programming team keeps the system integrated and running all the time. The programmers write all production code in pairs, and all work together all the time. They code in a consistent style so that everyone can understand and improve all the code as needed.

- Core Practices: Metaphor, Sustainable Pace

The Extreme Programming team shares a common and simple picture of what the system looks like. Everyone works at a pace that can be sustained indefinitely.

5.) Write suitable applications of different software models [5M Jun 2015]

Ans: The development models are the various processes or methodologies that are being selected for the development of the project depending on the project's aims and goals. There are many development life cycle models that have been developed in order to achieve different required objectives. The models specify the various stages of the process and the order in which they are carried out.

The selection of model has very high impact on the testing that is carried out. It will define the what, where and when of our planned testing, influence regression testing and largely determines which test techniques to use.

There are various Software development models or methodologies. They are as follows:

Waterfall model

V model

Incremental model

RAD model

Agile model

Iterative model

Spiral model

Choosing right model for developing of the software product or application is very important. Based on the model the development and testing processes are carried out. Different companies based on the software application or product, they select the type of development model whichever suits to their application. But these days in market the 'Agile Methodology' is the most used model. 'Waterfall Model' is the very old model. In 'Waterfall Model' testing starts only after the development is completed. Because of which there are many defects and failures which are reported at the end. So,the cost of fixing these issues are high. Hence, these days people are preferring 'Agile Model'. In 'Agile Model' after every sprint there is a demo-able feature to the customer. Hence customer can see the features whether they are satisfying their need or not.

'V-model' is also used by many of the companies in their product. 'V-model' is nothing but 'Verification' and 'Validation' model. In 'V-model' the developer's life cycle and tester's life cycle are mapped to each other. In this model testing is done side by side of the development.

Likewise 'Incremental model', 'RAD model', 'Iterative model' and 'Spiral model' are also used based on the requirement of the customer and need of the product.

Self-evaluation

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		Tick Your choice
1.	Do you understand the basics of Requirement Elicitation?	○ Yes ○ No
2.	Do you understand the concept of Data Flow Diagram?	○ Yes ○ No
3.	Do you recall Feasibility Analysis?	○ Yes ○ No
4.	Do you understand the Cost- Benefit Analysis?	○ Yes ○ No
5.	Do you remember the Requirement Model?	○ Yes, Completely. ○ Partialy. ○ No, Not at all.