Virtual University of Pakistan

Federal Government University

Software Requirement Engineering

Course Code – CS-708

Course Instructor

Dr. Ghulam Ahmad Farrukh
Ph.D Software Engineering
George Mason University, USA

Reference Books:

1. Software Quality: Analysis and Guidelines for Success by Capers Jones
2. Software Assessments, Benchmarks, and Best Practices by Capers Jones
3. Customer-oriented Software Quality Assurance by Frank P. Ginac
4. Software Engineering by Sommerville
5. Software Engineering: A Practitioner's Approach by Roger S. Pressman
6. Requirements Engineering: Processes and Techniques by Kotonya and Sommerville
7. Inroads to Software Quality by Alka Jarvis and Vern Crandell
8. Software Requirements: Objects, States, and Functions by Alan M. Davis
9. Software Engineering Quality Practices by Ronald K. Kandt
10. High Quality Low Cost Software Inspections by Ronald A. Radice

Video Links: http://www.vumultan.com/CS708Video.aspx

Published By: www.vumultan.com

Software Requirement Engineering

Course Code – CS708

Presented by:

Dr. Ghulam Ahmad Farrukh
Ph.D Software Engineering
George Mason University, USA

Published by: http://www.vumultan.com/

Software Requirements

Introduction

- Requirements form the basis for all software products
- Requirements engineering is the process, which enables us to systematically determine the requirements for a software product

Requirement

- Something required, something wanted or needed
  - Webster's dictionary
- There is a huge difference between wanted and needed and it should be kept in mind all the time

Software Requirements

- A complete description of what the software system will do without describing how it will do it is represented by the software requirements
- Software requirements are complete specification of the desired external behavior of the software system to be built
- They also represent External behavior of the system
- Software requirements may be:
  - ✓ Abstract statements of services and/or constraints
  - ✓ Detailed mathematical functions
- Software requirements may be:
  - ✓ Part of the bid of contract
  - ✓ The contract itself
  - ✓ Part of the technical document, which describes a product

IEEE Definition

- A condition or capability that must be met or possessed by a system...to satisfy a contract, standard, specification, or other formally imposed document
  - IEEE Std 729

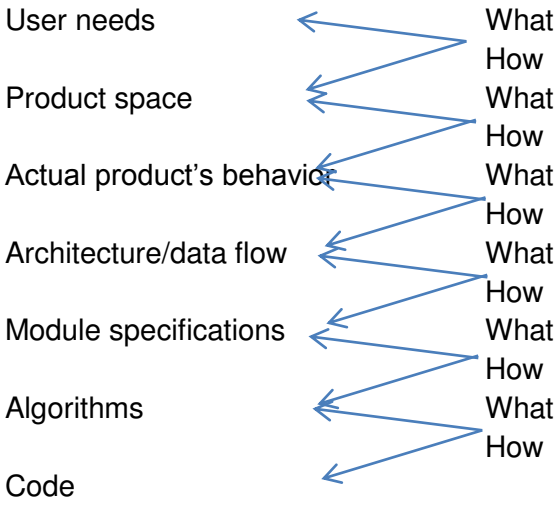
Sources of Requirements

- Stakeholders
  - People affected in some way by the system
- Documents
- Existing system
- Domain/business area

Levels of Software Requirements

- Stakeholders describe requirements at different levels of detail
  - "What versus How"
  - "One person's floor is another person's ceiling"

What versus How



Importance of Software Requirements

- The hardest single part of building a software system is deciding what to build...No other part of the work so cripples the resulting system if done wrong. No other part is difficult to rectify later
  - Fred Brooks

Examples of Requirements

- The system shall maintain records of all payments made to employees on accounts of salaries, bonuses, travel/daily allowances, medical allowances, etc.
- The system shall interface with the central computer to send daily sales and inventory data from every retail store
- The system shall maintain records of all library materials including books, serials, newspapers and magazines, video and audio tapes, reports, collections of transparencies, CD-ROMs, DVDs, etc.
- The system shall allow users to search for an item by title, author, or by International Standard Book Number
- The system's user interface shall be implemented using a web browser
- The system shall support at least twenty transactions per second
- The system facilities which are available to public users shall be demonstrable in ten minutes or less

Kinds of Software Requirements

1. Functional requirements
2. Non-functional requirements
3. Domain requirements
4. Inverse requirements
5. Design and implementation constraints

1: Functional Requirements

- Statements describing what the system does
- Functionality of the system
- Statements of services the system should provide
  - ✓ Reaction to particular inputs
  - ✓ Behavior in particular situations
- Sequencing and parallelism are also captured by functional requirements
- Abnormal behavior is also documented as functional requirements in the form of exception handling
- Functional requirements should be complete and consistent
- Customers and developers usually focus all their attention on functional requirements

Functional Requirements Examples:

- The system shall solve a quadratic equation using the following formula
  - ✓ $x = (-b \pm sqrt(b^2 - 4*a*c))/2*a$
- The user shall be able to search either the entire database of patients or select a subset from it (admitted patients, or patients with asthma, etc.)
- The system shall provide appropriate viewers for the user to read documents in the document store
- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall use to access that order
- The system shall allow customers to return non-perishable items within fifteen days of the purchase. A customer must present the original sale receipt to return an item

Comments on Examples

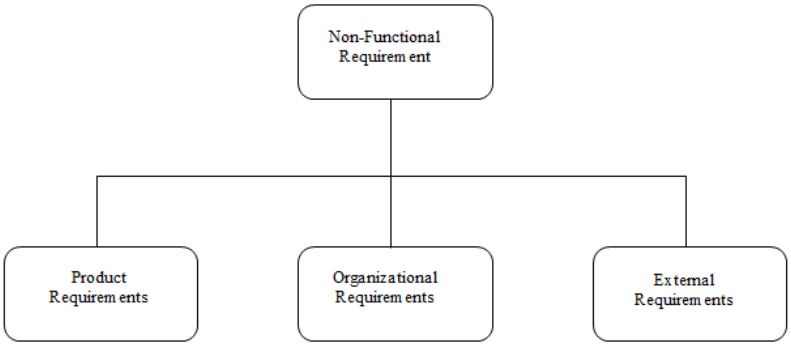
- Notice the level of detail in different requirements described above. Some are very detailed compared to others
- Notice the ambiguity in the requirement, which uses the term 'appropriate viewers'
- This requirement does not mention the formats of documents and types of viewers, which can be used
- Notice the ambiguity in the requirement for solving the quadratic equation. The requirement does not speak about the possibility when the value of 'a' is zero
  - ✓ $x = (-b \pm sqrt(b^2 - 4*a*c)) / 2*a$
- Incomplete and ambiguous requirements are open to multiple interpretations and assumptions
- This can lead to the development of poor quality, or faulty, software products

2: Non-Functional Requirements

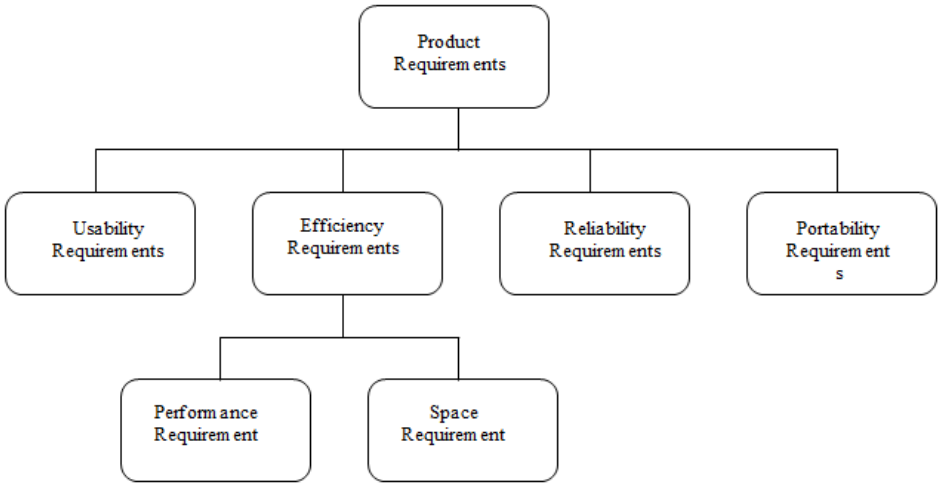
- Most non-functional requirements relate to the system as a whole. They include constraints on timing, performance, reliability, security, maintainability, accuracy, the development process, standards, etc.
- They are often more critical than individual functional requirements
- Capture the emergent behavior of the system, that is they relate to system as a whole
- Must be built into the framework of the software product
- Failure to meet a non-functional system requirement may make the whole system

unusable

- For example, if an aircraft system does not meet reliability requirements, it will not be certified as 'safe'
- If a real-time control system fails to meet its performance requirements, the control functions will not operate correctly
- Non-functional requirements arise through user needs, because of budget constraints, because of organizational policies, because of the need of interoperability with other software and hardware systems, or because of external factors such as safety regulations, privacy legislation, etc.



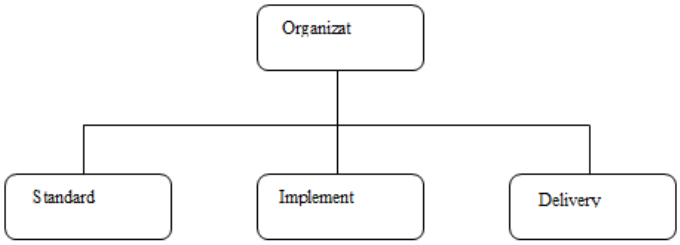
Product Requirements:



Product Requirements Examples

- The system shall allow one hundred thousand hits per minute on the website
- The system shall not have down time of more than one second for continuous execution of one thousand hours

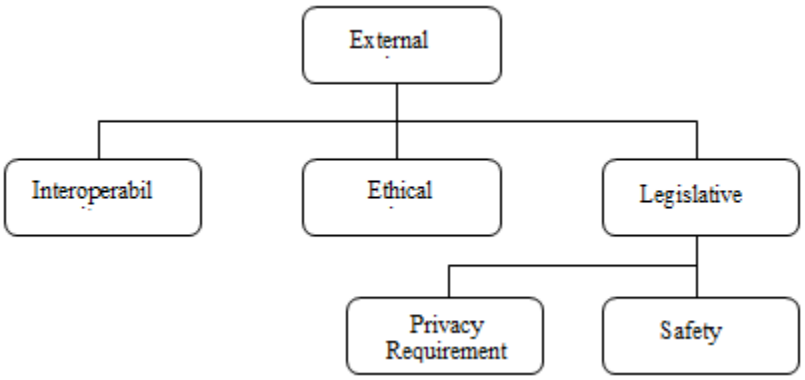
Organizational Requirements:



Organizational Requirements Examples

- The system development process and deliverable documents shall conform to the MIL-STD-2167A
- Any development work sub-contracted by the development organization shall be carried out in accordance with Capability Maturity Model

External Requirements:



External Requirements Examples

- The system shall not disclose any personal information about members of the library system to other members except system administrators
- The system shall comply with the local and national laws regarding the use of software tools

Observations on Non-Functional Requirements

- Non-functional requirements can be written to reflect general goals for the system. Examples include:
  - Ease of use
  - Recovery from failure
  - Rapid user response
- Goals are open to misinterpretation
- Objective verification is difficult
- Distinction between functional and non-functional is not always very clear
- Non-functional requirements should be written in a quantitative manner as much as possible, which is not always easy for customers
- For some goals, there are no quantitative measures, e.g., maintainability

- Goals can be useful to designers and developers, as they give clues to them about priorities of the customers
- Chances of conflicts within non-functional requirements are fairly high, because information is coming from different stakeholders. For example, different stakeholders can give different response times or failure tolerance levels, etc.
- Some negotiations must be done among different stakeholders, to achieve an agreement in these situations
- Non-functional requirements should be highlighted in the requirements document, so that they can be used to build the architecture of the software product

Non-Functional Requirements Discussion

- NFRs are very important to capture the emergent behavior of the system in these there major dimensions
- Product
  - Usability, reliability, portability, efficiency (performance, space)
- Organizational
  - Standards, implementation, delivery
- External
  - Interoperability, ethical, legislative (privacy, safety)

NFRs as Goals

- Non-functional requirements are sometimes written as general goals, which are difficult to verify
- They should be expressed quantitatively using metrics (measures) that can be objectively tested

Example: Goal converted into an NFR

- Goal (unverifiable)
  - The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized

- Non-functional requirement (verifiable)
  - Experienced controllers shall be able to use all the system functions after a total of two hours' training. After this training, the average number of errors made by experienced users shall not exceed two per day

Software Metrics for Non-Functional Requirements (NFRs)

	Property	Measure
1	Speed	• Processed transactions/second • Response time • Screen refresh time
2	Size	• K bytes • Number of function points
3	Ease of use	• Training time • Number of help frames
4	Reliability	• Mean time to failure • Probability of unavailability • Rate of failure occurrence • Availability
5	Robustness	• Time to restart after failure • Percentage of events causing failure • Probability of data corruption on failure
6	Portability	• Percentage of target-dependent statements • Number of target systems

- With the help of these measures the NFRs can be verified quantitatively
- It should also be noted that the cost of quantitatively verifying each NFR may be very high

3: Domain Requirements

- Requirements that come from the application domain and reflect fundamental characteristics of that application domain
- These can be both the functional or non-functional requirements
- These requirements, sometimes, are not explicitly mentioned
- Domain experts find it difficult to convey domain requirements
- Their absence can cause significant dissatisfaction
- Domain requirements can impose strict constraints on solutions. This is particularly true for scientific and engineering domains
- Domain-specific terminology can also cause confusion
- Example: In a commission-based sales businesses, there is no concept of negative commission. However, if care is not taken novice developers can be lured into developing systems, which calculate negative commission
- Banking domain has its own specific constraints, for example, most banks do not allow over-draw on most accounts, however, most banks allow some accounts to be over-drawn

4: Inverse Requirements

- They explain what the system shall not do. Many people find it convenient to describe their needs in this manner
- These requirements indicate the indecisive nature of customers about certain aspects of a new software product
- Example: The system shall not use red color in the user interface, whenever it is asking for inputs from the end-user

5: Design and Implementation Constraints

- They are development guidelines within which the designer must work
- These requirements can seriously limit design and implementation options
- Can also have impact on human resources

Design and Implementation Constraints Examples

- The system shall be developed using the Microsoft .Net platform
- The system shall be developed using open source tools and shall run on Linux operating system

Another view

- There also exists another view of requirements apart from different kinds of requirements we have studied so far.
  - Another view of requirements
  - There are some problems which occur in requirements, that are necessary to be identified and properly attended.
  - Problems in requirements

Another View of Requirements

- In general requirements can be viewed as
  - User/customer requirements OR System contract requirements

User/Customer Requirements

- Functional and non-functional requirements should be stated in natural language with the help of forms or simple diagrams describing the expected services of a system by the User under certain constraints
- These are understandable by users, who have no, or little, technical knowledge
- System design characteristics should be avoided as much as possible
- It is a good practice to separate user requirements from more detailed system requirements in a requirements document
- Including too much information in user requirements, constraints the system designers from coming up with creative solutions
- The rationale associated with requirements is very important. It helps in managing changes to requirements

System Contract Requirements

- Sets out the system services and constraints in detail
- May serve as the basis of contract for implementation of the system
- Should be complete and consistent
- They are used by the designers and developers as the starting point for system design
- They should be understood by technical staff of the customer organization and the development team
- In principle, these requirements should also state 'what' the system does, rather than 'how' it is implemented
- However, with the level of details needed to specify the system completely, it is not possible to exclude all design information
- An initial architecture of the system may be defined to help structure the requirements specification
- In most cases, systems interoperate with other systems
- Use of specific design may be included as an external requirement
- Natural language is often used to describe system requirements
- Some specification languages may be used with natural language, which add structure to specifications and reduce ambiguity
- Unified Modeling Language (UML) is a specification language, which has become the de-facto standard for modeling requirements

Requirements Problems

- The requirements don't reflect the real needs of the customer for the system
- Requirements are inconsistent and/or incomplete
- It is expensive to make changes to requirements after they have been agreed upon
- There are misunderstandings between customers, those developing the system requirements, and software engineers developing or maintaining the system

Problems with Natural Languages

- Requirement specification in natural language pose some problems which include
  - ✓ Lack of clarity
  - ✓ Requirements amalgamation
  - ✓ Requirements confusion
- Natural language understanding relies on the specification readers and writers using the same words for same concept
- A natural language requirements specification is over-flexible. "You can say the same thing in completely different ways"
- It is not possible to modularize natural language requirements. It may be difficult to find all related requirements
  - ✓ To discover the impact of a change, every requirement have to be examined

Impact of Wrong Requirements

- When requirements are wrong, systems are late, unreliable and don't meet customers needs
- his results in enormous loss of time, revenue, market share, and trust of customers

Processes and Process Models

Process

- A process is an organized set of activities, which transforms inputs to outputs
- We can use synonyms of process such as: procedure, method, course of action, etc.
- Processes are essential for dealing with complexity in real world
- Processes document the steps in solving a certain problem
- They allow knowledge to be reused
- They Allow people to apply the process in their peculiar but similar problems
- Examples of Processes
  - ✓ An instruction manual for operating a microwave oven
  - ✓ An instruction manual for assembling a computer or its parts
  - ✓ A procedure manual for operating a motor vehicle radio and CD player
  - ✓ A quality manual for software development.
  - ✓ Such a manual describes the processes, which should be used to assure the quality of the software

Software Processes

- Software engineering, as a discipline, has many processes
- These processes help in performing different software engineering activities in an organized manner
  1. Requires creativity
  2. Provides interactions between a wide range of different people
  3. Helps in engineering judgment
  4. Requires background knowledge
  5. Experiences
- Examples of Software Processes
  - ✓ Software engineering development process (SDLC)
  - ✓ Requirements engineering process
  - ✓ Design process
  - ✓ Quality assurance process
  - ✓ Change management process

Software Requirements Engineering Process

- Before discussing different aspects of requirements engineering process, let us discuss the concept of process models

Process Models

- A process model is a simplified description of a process presented from a particular perspective
- There may be several different models of the same process
- No single model gives a complete understanding of the process being modeled

Variations in Process Models

- A process model is produced on the anticipated need for that model. We may need
  - A model to help explain how process information has been organized
  - A model to help understand and improve a process
  - A model to satisfy some quality management standard

Types of Process Model

- Coarse-grain activity models
- Fine-grain activity models
- Role-action models
- Entity-relation models

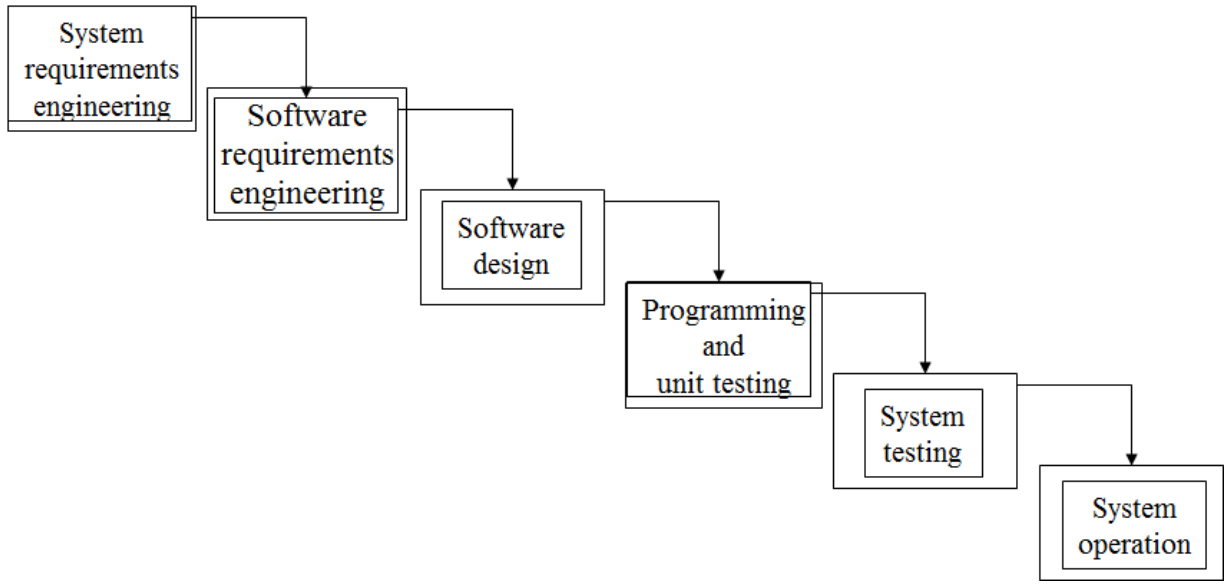
Coarse-grain Activity Model

- This type of model provides an overall picture of the process
- Describes the context of different activities in the process
- It doesn't document how to enact a process

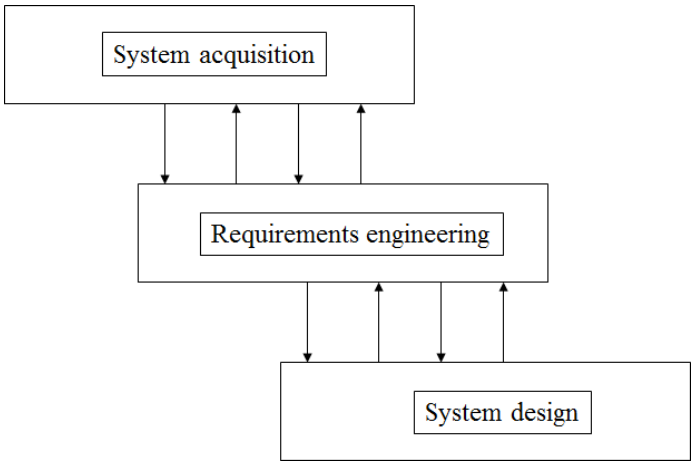
Context of Requirements Engineering

- Software requirements follow the "system requirements" and "system design"
- The primary goal is understanding
- Software requirements are followed by software design in a software development life cycle

Context of RE Process in Waterfall Model

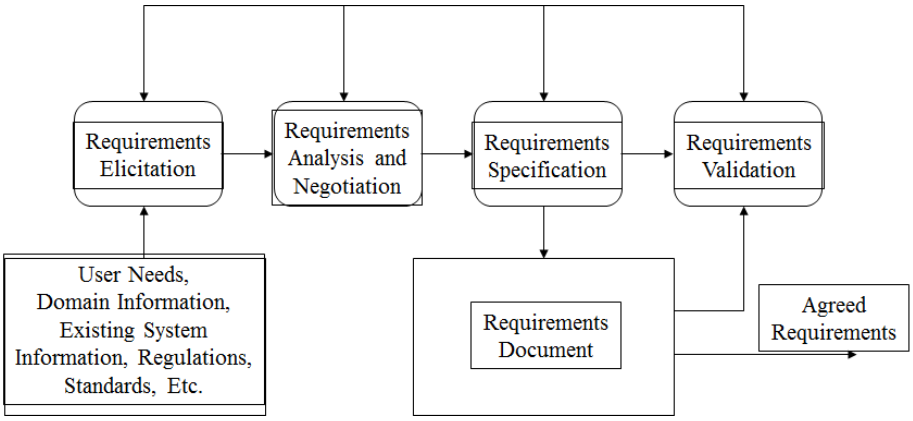


Another Perspective on Context of RE Process

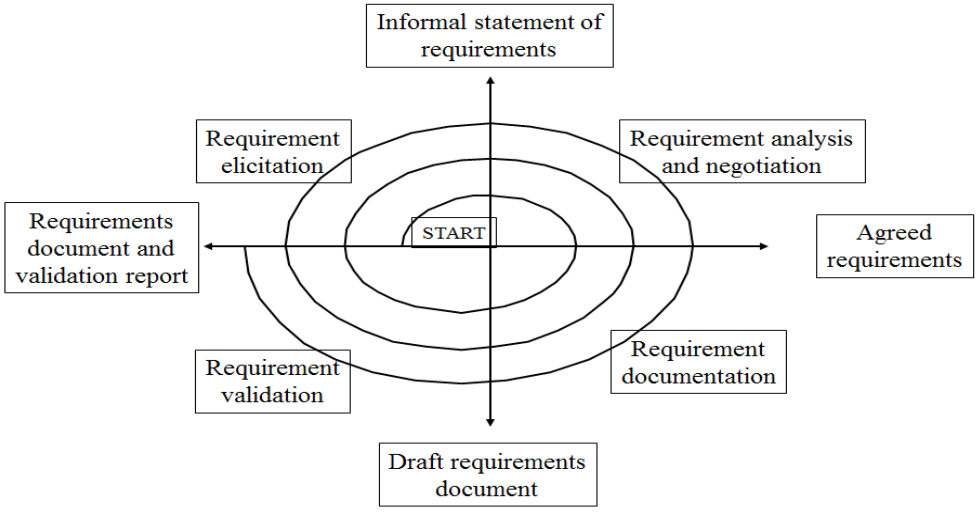


Coarse-grain Activity Model of the Requirements Engineering Process

- Requirements engineering process is an example of coarse-grain activity model



Spiral Model of RE Process



Fine-grain Activity Models

- These are more detailed models of a specific process, which are used for understanding and improving existing processes
- We'll discuss some fine-grain processes within the general requirements engineering processes in later lectures

Role-action Models

- These are models, which show the roles of different people involved in the process and the actions which they take
- They are useful for process understanding and automation

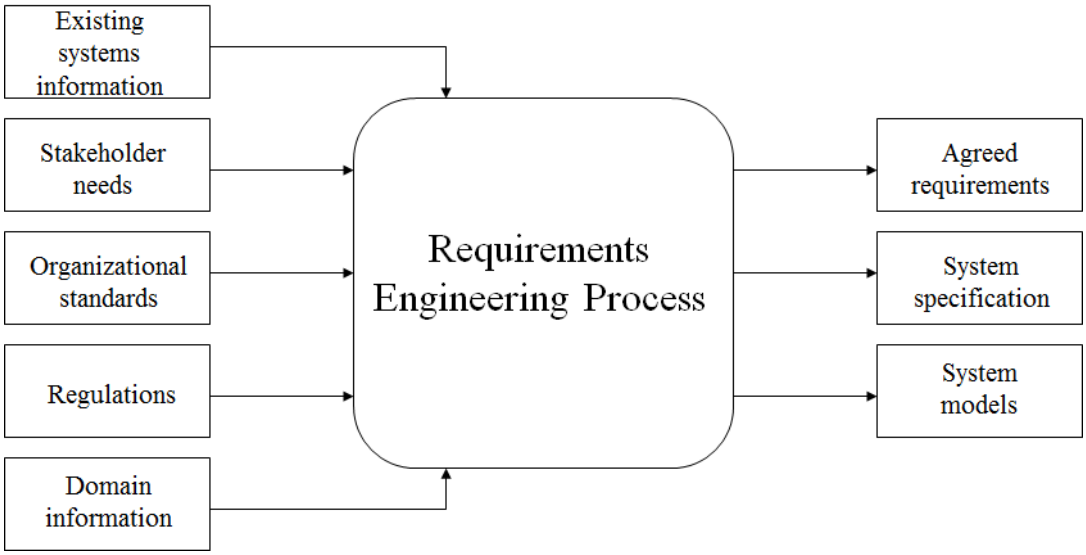
Entity-relation Models

- The models show the process inputs, outputs, and intermediate results and the relationships between them
- They are useful in quality management systems

Requirements Engineering Process

- The process(es) involved in developing system requirements is collectively known as Requirements Engineering Process

RE Process - Inputs and Outputs



RE Process – Inputs

It includes existing system information

- Information: It include the information about the functionality of systems to be replaced and the Information about other systems, which interact with the system being specified
- Stakeholder needs: Description of what system stakeholders need from the system to support their work

- Organizational standards: Standards used in an organization regarding system development practice, quality management, etc.
- Regulations: External regulations such as health and safety regulations, which apply to the system
- Domain information: General information about the application domain of the system
- Agreed requirements: A description of the system requirements, which is understandable by stakeholders and which has been agreed by them
- System specification: This is a more detailed specification of the system, which may be produced in some cases
- System models: A set of models such as a data-flow model, an object model, a process model, etc., which describes the system from different perspectives

RE Process Variability

- RE processes vary radically from one organization to another, and even within an organization in different projects
- Unstructured process rely heavily on the experience of the people, while systematic processes are based on application of some analysis methodology , but they still require human judgment
- RE processes vary radically from one organization to another, and even within an organization in different projects
- Unstructured process rely heavily on the experience of the people, while systematic processes are based on application of some analysis methodology , but they still require human judgment

Variability Factors

- There are four factors which count towards the variability of the Requirements Engineering Process
    1. Technical maturity: The technologies and methods used for requirements engineering vary from one organization to other
    2. Disciplinary involvement: The types of engineering and managerial disciplines involved in requirements vary from one organization to another
    3. Organizational culture: The culture of an organization has important effect on all business and technical processes
    4. Application domain: Different types of application system need different types of requirements engineering process.

RE Process

- Requirement Engineering Process has a formal starting and ending point in the overall software development life cycle.
- Begins
    - ✓ There is recognition that a problem exists and requires a solution
    - ✓ A new software idea arises
- Ends
    - ✓ With a complete description of the external behavior of the software to be built

- It is a continuous process in which the related activities are repeated until requirements are of acceptable quality
- It is one of the most critical processes of system development
- Based on the need of individual software projects and organizational needs, requirements engineering processes are tailored
- An important point to remember is that
- "There is no ideal requirements engineering process!"

Two Main Tasks of RE

- There are two main tasks which need to be performed in the requirements engineering process.
    1. Problem analysis: Analysis of a software problem
    2. Product description: Complete specification of the desired external behavior of the software system to be built. Also known as functional description, functional requirements, or specifications

Problem Analysis

- Problem analysis is the first and foremost task of requirements engineering process. It includes:
    - ✓ Brainstorming, interviewing, eliciting requirements
    - ✓ Identifying all possible constraints
    - ✓ Expansion of information
- Trading off constraints and organizing information
- Complete understanding should be achieved

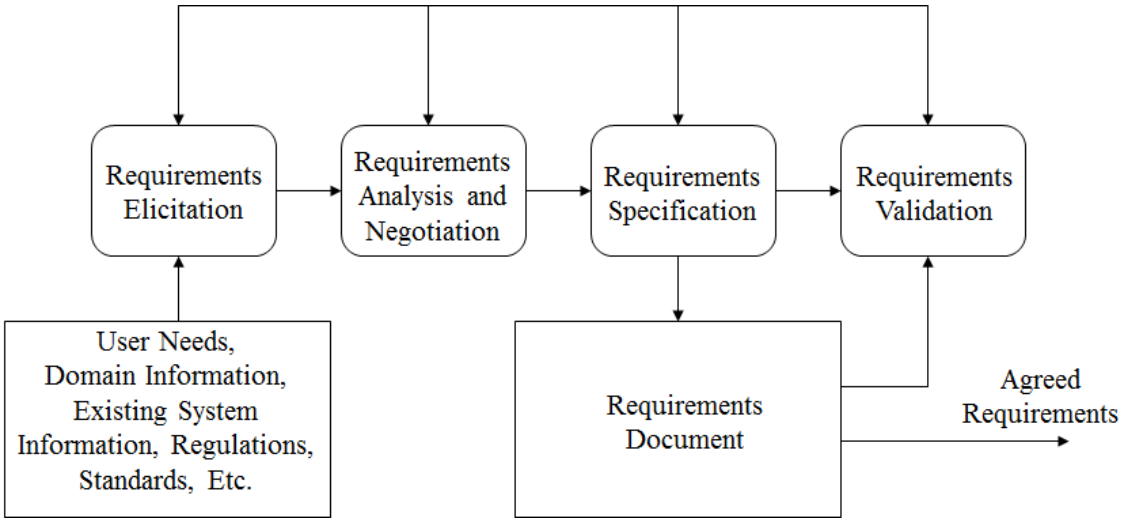
Product Description

- Product description is another task of requirements engineering process. In this task we:
    - ✓ Make decisions to define the external behavior of the software product
    - ✓ Organize ideas, resolve conflicting views, and eliminate inconsistencies and ambiguities

What Really Happens

- It should be kept in mind that: "Both problem analysis and product description run in parallel and iteratively throughout the requirements engineering process".

Requirements Engineering Activities



Requirements Elicitation

- Requirements elicitation activity is performed by
- Determining the system requirements through consultation with stakeholders, from system documents, domain knowledge, and market studies
- Requirements acquisition or requirements discovery

Requirements Analysis and Negotiation

- Requirements analysis and negotiation activity is performed by
- Understanding the relationships among various customer requirements and shaping those relationships to achieve a successful result
- Negotiations among different stakeholders and requirements engineers
- Incomplete and inconsistent information needs to be tackled here
- Some analysis and negotiation needs to be done on account of budgetary constraints

Requirements Specification

- Requirements specification includes
- Building a tangible model of requirements using natural language and diagrams
- Building a representation of requirements that can be assessed for correctness, completeness, and consistency

Requirements Document

- Detailed descriptions of the required software system in form of requirements is captured in the requirements document
- Software designers, developers and testers are the primary users of the document

Requirements Validation

- It involves reviewing the requirements model for consistency and completeness

- This process is intended to detect problems in the requirements document, before they are used as a basis for the system development

Requirements Management

- Although, it is not shown as a separate activity in RE Process, it is performed through out the requirements engineering activities.
- Requirements management asks to identify, control and track requirements and the changes that will be made to them

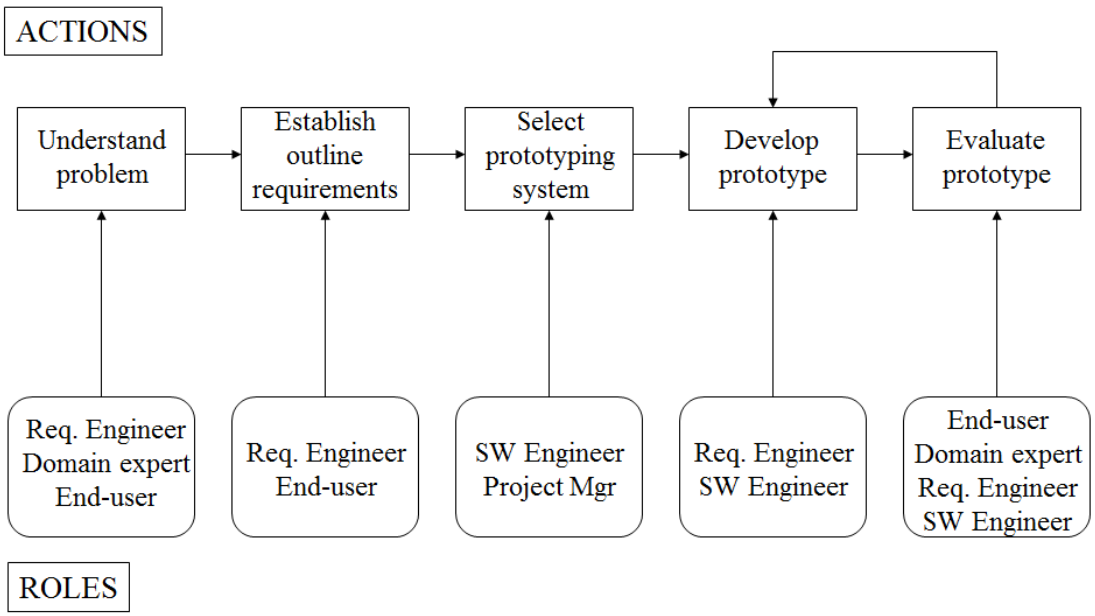
Who are Actors?

- Actors in a process are the people involved in the execution of that process
- Actors are normally identified by their roles rather than individually, e.g., project manager, purchasing director, and system engineer

Actors in the RE Process

- Requirements engineering involves people who are primarily interested in the problem to be solved (end-users, etc) as well as people interested in the solution (system designers, etc.)
- Another group of people, such as health & safety regulators, and maintenance engineers may be effected by the existence of the system
- Role-action diagrams are process models which show the actors associated with different process activities
- They document the information needs of different people involved in the process
- They use model of prototype software system as part of requirements elicitation process

Role-Action Diagram for Software Prototyping



Role Descriptions

Role	Description
Domain Expert	Responsible for proving information about the application domain and the specific problem in that domain, which is to be solved
System End-user	Responsible for using the system after delivery
Requirements Engineer	Responsible for eliciting and specifying the system requirements
Software Engineer	Responsible for developing the prototype software system
Project Manager	Responsible for planning and estimating the prototyping project

Human and Social Factors

- Requirements engineering processes are dominated by human, social and organizational factors because they always involve a range of stakeholders from different backgrounds and with different individual and organizational goals
- System stakeholders may come from a range of technical and non-technical background and from different disciplines

Stakeholder Types

- Software engineers
- System end-users
- Managers of system end-users
- External regulators
- Domain experts

Factors Influencing Requirements

- Personality and status of stakeholders
- The personal goals of individuals within an organization
- The degree of political influence of stakeholders within an organization

Process Support

- One way to minimize errors in the requirements engineering is to use process models and to use CASE tools
- The most mature CASE tools support well-understood activities such as programming and testing and the use of structured methods
- Support for requirements engineering is still limited because of the informality and the variability of the process

CASE Tools for RE

- Modeling and validation tools support the development of system models which can be used to specify the system and the checking of these models for completeness and consistency
- Management tools help manage a database of requirements and support the management of changes to these requirements

Process Improvement

- Process improvement is concerned with modifying processes in order to meet some improvement objectives
- Improvement objectives
  - ✓ Quality improvement
  - ✓ Schedule reduction
  - ✓ Resource reduction

Planning Process Improvement

- Some important questions arise:
- What are the problems with current processes?
- What are the improvement goals?
- How can process improvement be introduced to achieve these goals?
- How should process improvements be controlled and managed?

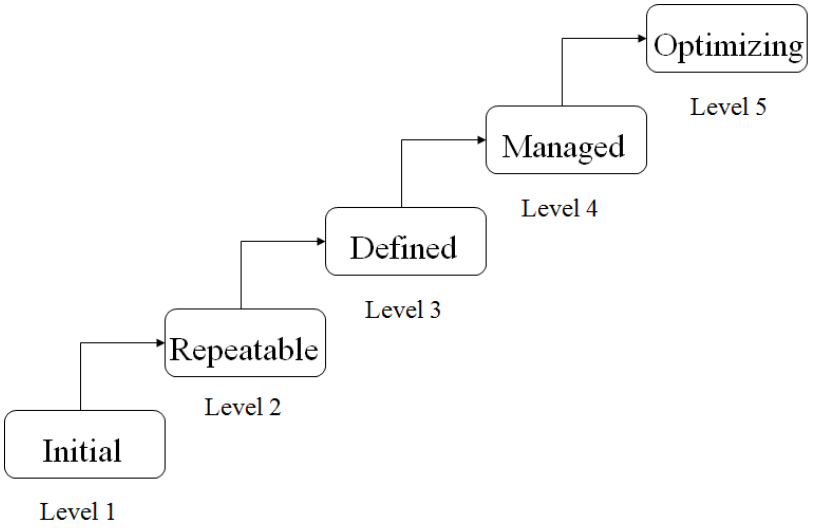
RE Process Problems

- Lack of stakeholder involvement
- Business needs not considered
- Lack of requirements management
- Lack of defined responsibilities
- Stakeholder communication problems
- Over-long schedules and poor quality requirements documents

Process Maturity

- Process maturity can be thought of as the extent that an organization has defined its processes, actively controls these processes and provides systematic human and computer-based support for them
- The SEI's Capability Maturity Model is a framework for assessing software process maturity in development organizations

Capability Maturity Model



CMM Level 1: Initial

- Organizations have an undisciplined process and it is left to individuals that how to manage the process and which development techniques to use

CMM Level 2: Repeatable

- Organizations have basic cost and schedule management procedures in place. They are likely to be able to make consistent budget and schedule predictions for projects in the same application area

CMM Level 3: Defined

- The software process for both management and engineering activities is documented, standardized and integrated into a standard software process for the organization

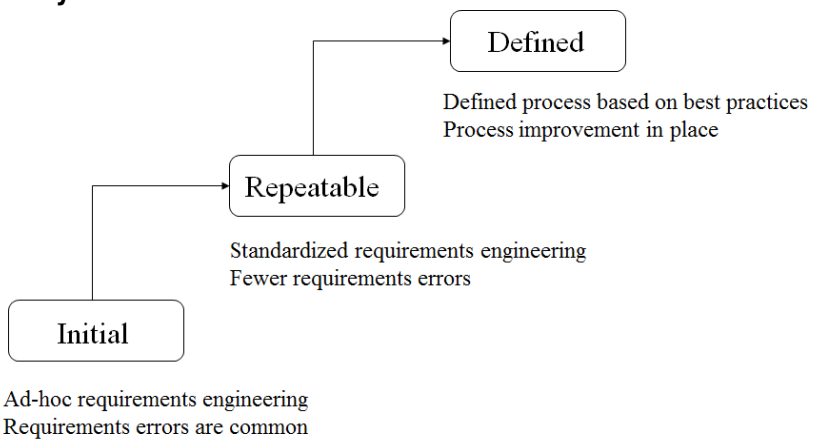
CMM Level 4: Managed

- Detailed measurements of both process and product quality are collected and used to control the process

CMM Level 5: Optimizing

- The organization has a continuous process improvement strategy, based on objective measurements, in place

RE Process Maturity Model



Initial RE Process Maturity Level

- There is no defined RE process.
- It suffer from requirements problems such as requirements volatility, unsatisfied stakeholders and high rework costs.
- It is dependent on individual skills and experience
- Defined standards for requirements documents, policies and procedures for requirements management
- Defined RE process based on good practices and techniques. Active process improvement process is in place

Best Practices for RE Process Improvement

- RE processes can be improved by the systematic introduction of best requirements engineering practices
- Each improvement cycle identifies best practice guidelines and works to introduce them in an organization
- Best practices will be discussed throughout the semester

Requirements Engineering Costs

- About fifteen percent (15%) of system development costs
- However, if the requirements engineering process is not executed properly, this cost can increase substantially

Social and Cultural Issues in Requirements Engineering

- Some aspects of the requirements engineering process deal with social and cultural issues
- What is the best way to deal with these issues?
- Some think that these issues fall outside the scope of requirements engineering process, and fall under management, interpersonal skills, or ethics

Social Issues in RE

- Requirements engineering is a social process, as it involves interaction among clients, engineers, and other systems
- Requirements engineering is not an entirely formal process, because it involves discovering client needs and reconciling them with technical possibilities

Stakeholders in RE Process

- At least three major groups participate in requirements engineering process;
  - ✓ The client organization
  - ✓ The requirements team
  - ✓ The development team
- There may be other interested parties, e.g., regulatory authorities

Six Areas of Social Issues

- Within the client organization
- Within the requirements team
- Between the client and the requirements team
- Between the development and requirements teams
- Within the development team
- Between the development team and the client

Issues within the Client Organization

- In a large organization, there are usually competing divisions or groups, so the notion of 'the client' is not obvious
- Intended users of the system may be different people from the ones who interact with

the requirements team

- The users of the system should be brought into the requirement engineering process, as they hold the key of the eventual success of the software engineering project
- The requirement process reveals the problems within the client organization, which must be addressed by facilitating communication among different stakeholders
- The problems within the client organization must not be buried, as they effect the implementation of the project
- The new automated system may have profound impact on how the business is conducted or how information is classified within the organization
- Success of the project requires that every group within the organization understand different aspects of the new system
- Problems of tacit knowledge
  - ✓ Say-do problem

Issues within the Requirements Team

- How work is organized?
- What methods and notations are used?
- What team members think about organization and how jelled requirement team is?

Issues between Client Organization and Requirements Team

- Financial arrangements
- Ethical obligations
- Legal safeguards
- Personal relationships
- Denial of information
- Management of changes

Issues between Development and Requirement Teams

- Development team needs to work very closely with the requirements team to resolve inconsistencies and to get details
- In some cases, requirements team may be disbanded or assigned other tasks

Issues of Development Team

- Team members may be demoralized
- There may be high turn over rate
- The deadlines may slip
- Developers dislike documentation
- Development teams may have to communicate with clients directly, to gain better understanding of the project's possibilities and limitations, both for initial development and maintenance

Cultural Issues in RE

- Advances in the internet and communication technologies has enabled customers and developers to collaborate with each other in geographically and temporally dispersed environments, There may be
  - ✓ Time zones differences
  - ✓ Language and terminology differences

- ✓ Religious and racial differences
- ✓ Ethical issues
- ✓ Political differences
- ✓ Differences in business environment

Example: A Billion

- Scientific community and US consider the following number to be a billion
  1,00,00,00,000
- For the rest of the world, a billion is
  10,00,00,00,00,000

Differences in Time Zones

- Working hours of clients and developers may differ by eight hours or more
- Arranging phone calls and video conferences become a hassle as one party has to come to office very early or stay very late
- Analysts start assuming requirements

Language and Terminology Differences

- Clients and developers may speak different languages or different dialects
- Requirements errors are introduced by not understanding other partner's language and terminology properly
- People and government in the US, and worldwide scientific community consider the following number to be a billion
  1,00,00,00,000
- For the rest of the world, a billion is
  10,00,00,00,00,000
- Globally, people communicate with fellow citizens using sports lingo to convey certain situations and concepts, even in the business environment
- This can cause misunderstandings
- Use of the word 'hockey' in Pakistan and US means two different sports: 'field hockey' and 'ice hockey' respectively

Religious and Racial Differences

- Insensitive comments on religious and racial backgrounds of people involved in software engineering projects can become a major hindrance in the successful execution of the requirements engineering process

Ethical Issues

- Access to confidential client information
- Possibility of elimination of jobs
- Differences of opinions with the client on the project

Political Differences

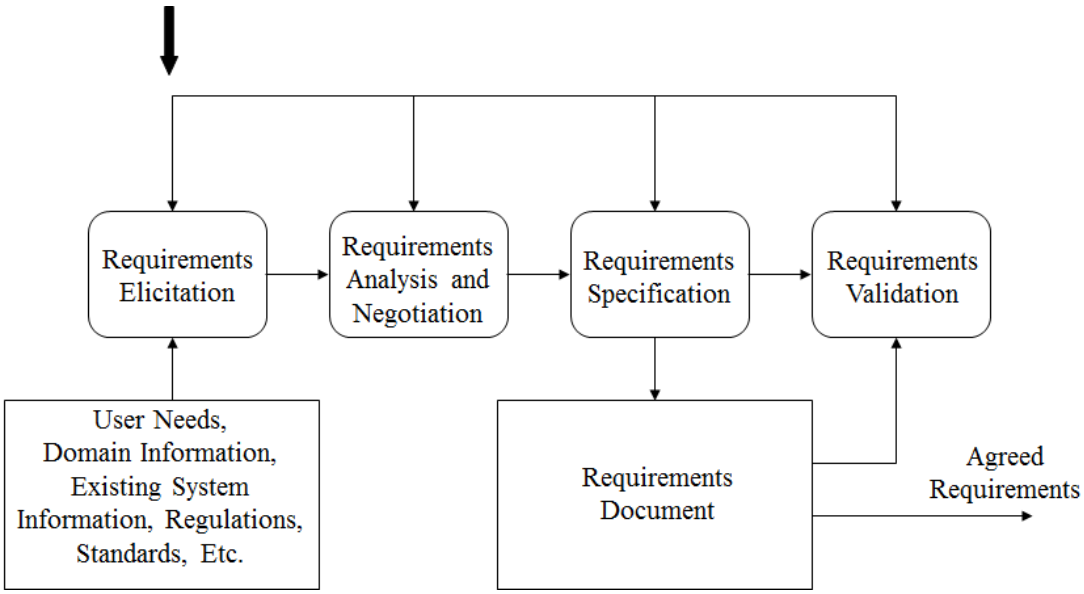
- Differences in political ideologies and personal convictions can also lead to

unprofessional environment in the execution of the requirements engineering process
- Some people do not want to work on military software program

Differences in Business Environments
- Every society has its own culture within the business community, which must be understood for successful execution of the requirements engineering process

Addressing Social and Cultural Issues
- Understand social and cultural issues and differences
- Avoid judgmental comments and offensive remarks on un-related views and beliefs of others
- Create an environment of respect and professionalism
- Focus on discovering the needs of the customers
- Use state-of-the-art technology to facilitate activities in the requirements engineering process

Requirements Engineering Process



RE Process 1: Requirements Elicitation
- Elicit means to gather, acquire, extract, and obtain, etc.
- Requirements elicitation means gathering requirements or discovering requirements
- Activities involved in discovering the requirements for the system

Basics of Knowledge Acquisition
- These are the sources of knowledge acquisition
  - ✓ Reading
  - ✓ Listening
  - ✓ Asking
  - ✓ Observing

Requirements Elicitation Techniques

- Individual
- Group
- Modeling
- Cognitive

Problems in Requirements Elicitation

- Problems of scope
  - ✓ The boundary of the system is ill-defined
  - ✓ Unnecessary design information may be given
- Problems of understanding
  - ✓ Users have incomplete understanding of their needs
  - ✓ Users have poor understanding of computer capabilities and limitations
  - ✓ Analysts have poor knowledge of problem domain
  - ✓ User and analyst speak different languages
  - ✓ Ease of omitting “obvious” information
  - ✓ Conflicting views of different users
  - ✓ Requirements are often vague and untestable, e.g., “user-friendly” and “robust”
- Problems of volatility
  - ✓ Requirements evolve over time and hence there are some requirements which are bound to change during the system development process due to one reason or the other.

Contexts in Requirements Elicitation Process

- It is important to consider the context in which requirements are being elicited. Requirements elicitation process may be followed in the following contexts
  - ✓ Organization
  - ✓ Environment
  - ✓ Project
  - ✓ Constraints imposed by people
- Organization
  - ✓ Submitters of input
  - ✓ Users of output
  - ✓ Ways in which the new system change the business process
- Environment
  - ✓ Hardware and software
  - ✓ Maturity of the target system domain
  - ✓ Certainty of the target system’s interfaces to the larger system
  - ✓ The target system’s role in the larger system
- Project
  - ✓ The attributes of the different stakeholder communities, such as the end users, sponsors, developers, and requirements analysts. Examples of such attributes are:
    - Management style
    - Management hierarchy
    - Domain experience
    - Computer experience

- The constraints imposed by the people
  - ✓ They are involved in the elicitation process, e.g., managerial constraints concerning cost, time, and desired quality in the target system

Requirements Elicitation Guidelines

- Assess the business and technical feasibility for the proposed system
- Identify the people who will help specify requirements and understand their organizational bias
- Define the technical environment
- Identify "domain constraints" that limit the functionality or performance of the system
- Define one or more requirements elicitation methods (interviews, focus groups, team meetings)
- Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded
- Identify ambiguous requirements as candidates for prototyping
- Create usage scenarios to help customers/users better identify requirements

Ethnomethodology

- Looks for behaviors that may be different in a specific culture but which have the same underlying purpose or meaning
- Conversational analysis
- Measurement of body system functions
- Non-verbal behavior studies
- Detailed video analysis

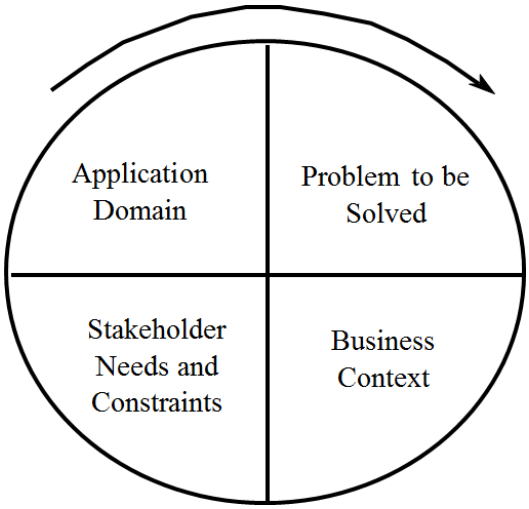
Requirements and Psychology

- Errors in statements can happen in two places
  - ✓ Perception of facts – reality
  - ✓ Linguistic representation of one of these perceptions – personal reality
- To remove these errors, requirements should be reviewed (during and after elicitation)

Use Case Modeling

- Define actors and black-box use cases
- The functional requirements of the system are defined in terms of use cases and actors
- The use case descriptions are a behavioral view

Components of Requirements Elicitation



Dimensions to Requirements Elicitation

- Application domain understanding
- Problem understanding
- Business understanding
- Understanding the needs and constraints of system stakeholders
- Application domain understanding
  - ✓ Knowledge of the general area where the system is applied
- Problem understanding
  - ✓ The details of the specific customer problem where the system will be applied must be understood
- Business understanding
  - ✓ Understand how systems interact and contribute to overall business goals
- Understanding the needs and constraints of system stakeholders
  - ✓ Understand, in detail, the specific needs of people who require system support in their work

Elicitation and Analysis Processes

- Requirements elicitation and requirements analysis are closely linked processes

Requirements Elicitation Stages

- Objective setting
- Background knowledge acquisition
- Knowledge organization
- Stakeholder requirements collection

Objective Setting

- Overall organizational objectives should be established at this stage
- These include general goals of business, an outline description of the problem to be

solved and why the system may be necessary, and the constraints on the system such as budget, schedule, and interoperability constraints

Background Knowledge Acquisition

- Requirements engineers gather and understand background information
- This includes information about the organization where the system is to be installed, information about the application domain of the system, and information about any existing systems which are in use and which may be replaced

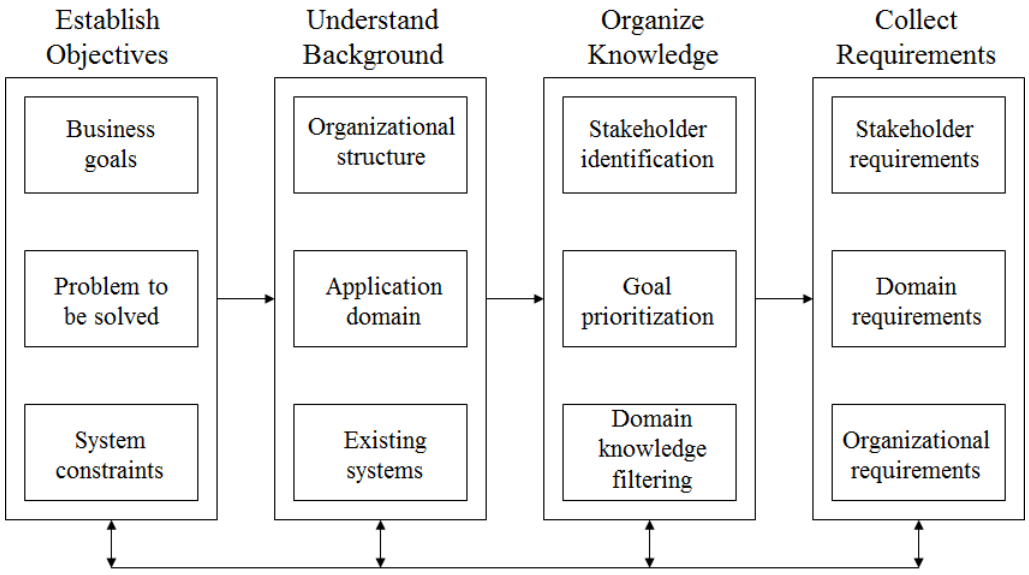
Knowledge Organization

- The large amount of knowledge which has been collected in previous stage must be organized and collated
- Identifying system stakeholders and their roles in the organization, prioritizing the goals of the organization and discarding domain knowledge which does not contribute directly to the system requirements

Stakeholder Requirements Collection

- It involves consulting system stakeholders to discover their requirements, and deriving requirements which come from the application domain and the organization which is acquiring the system

A General Requirements Elicitation Process



Comments on this Process

- It is an idealized process, while the reality of requirements elicitation tends to be much messier
- The activities are usually mixed up with each other

- If objective setting activities are not carried out, significant analysis problems occur, as no objective and business goals are available to prioritize requirements
- It is an idealized process, while the reality of requirements elicitation tends to be much messier
- The activities are usually mixed up with each other
- If objective setting activities are not carried out, significant analysis problems occur, as no objective and business goals are available to prioritize requirements
- The output from the requirements elicitation process should be a draft document which describes the system requirements, which is then analyzed to discover problems and conflicts in the requirements definition
- This process is followed by the requirements analysis process, which will be discussed in another lecture

Basics of Knowledge Acquisition

- Reading
- Listening
- Asking
- Observing
- Results in large volume of information, which must be organized to make it understandable

Knowledge Structuring Techniques

- Partitioning
  - ✓ Organization of knowledge into aggregation relationships, where requirements knowledge is described in terms of its parts
  - ✓ Booking system example: a booking record may be may be defined as a flight reference, source & destination of flight, the name & address of the passenger, fare, and date of travel
- Abstraction
  - ✓ Organization of knowledge according to general/specific relationships. Requirement knowledge is described by relating specific instances to abstract structures
  - ✓ Passenger abstraction may represent all classes of passengers (children, adults, full-fare paying, concessionary passengers, etc.)
- Projection
  - ✓ Organization of knowledge from several different perspectives or viewpoints
  - ✓ Booking system example: travel agents, airline management, check-in desk operators, passengers, a bookings database, etc.

Specific Elicitation Techniques

- Interviews
- Scenarios
- Observations and social analysis
- Requirements reuse

Interviews

- The requirements engineer or analyst discusses the system with different stakeholders and builds up an understanding of their requirements
- Interviews are less effective for understanding the application domain and the organizational issues due to terminology and political factors

Types of Interviews

- Closed interviews
  - ✓ The requirements engineer looks for answers to a pre-defined set of questions
- Open interviews
  - ✓ There is no predefined agenda and the requirements engineer discusses, in an open-ended way, what stakeholders want from the system

Interviewing Essentials

- Interviewers must be open-minded and should not approach the interview with pre-conceived notions about what is required
- Stakeholders must be given a starting point for discussion. This can be a question, a requirements proposal or an existing system
- Interviewers must be aware of organizational politics - many real requirements may not be discussed because of their political implications

Interview Steps

- Prepare
- Conduct
  - ✓ Opening
  - ✓ Body
  - ✓ Closing
- Follow through

<u>Prepare for the Interview</u>

- Before developing questions
  - ✓ Define the purpose and objectives
  - ✓ Determine whether the interview should be conducted by one person or a team (define roles for team members)
  - ✓ Contact interviewee to arrange time, place, and logistics of the interview and outline the purpose and format
  - ✓ Obtain background information
- After contacting the interviewee
  - ✓ Develop the interview guide
  - ✓ List name and title of interviewee and date of the interview
  - ✓ List questions in the order you will ask them
  - ✓ Move from general to specific
  - ✓ Include open questions to elicit essay type response (e.g., Describe…, Tell me…, How…)
  - ✓ Include closed questions to obtain specific information (e.g., Who? How much? Where?)

Conduct the Interview

- Opening
  - ✓ Establish rapport and build trust and credibility
    - Make eye contact
    - Shake hands
    - Introduce yourself (and your team); provide information about role(s) in the interview process
  - ✓ Clarify purpose, time frame, and key objectives
  - ✓ Transition to the core of the interview by leading into the first question
- Body
  - ✓ Follow your interview guide as you ask questions; use probes to follow up on a response
  - ✓ Be flexible and open-minded
  - ✓ Listen actively
  - ✓ Monitor your voice and body language
  - ✓ Identify interviewee's main concerns
  - ✓ Maintain rapport
  - ✓ Take accurate notes
  - ✓ Use silence and pauses
  - ✓ Ask for and obtain relevant documentation
  - ✓ Ask "catch-all" question at the end
- Closing
  - ✓ Summarize findings and link to purpose
  - ✓ Answer any questions the interviewee has
  - ✓ Determine and agree on next steps
  - ✓ Set next meeting, if necessary
  - ✓ Thank the interviewee for his/her input and for taking the time to meet with you

Follow Through

- Immediately after the interview, fill in your notes; be sure to jot down impressions and important ideas
- Review any documentation received from the interviewee
- Write an interview report, if necessary
- Follow up on leads obtained during the interview
  - ✓ Contact other potential interviewees
  - ✓ Research other data sources
- Follow up in agreed-upon next steps
- Send a thank you note to the interviewee, if appropriate

Listening

- The art of listening is most important. You can best impress your client by listening and giving due attention to what the client or customer is saying
- This requires effort on part of the interviewer
- Listening Steps
  - ✓ Hear
    - Listen to learn as much as you can so that you will know how to respond

- Give the speaker your undivided attention; don't just wait for your turn to speak
- Concentrate on the message, not the person
- Don't interrupt
- Tune out distractions such as interfering noises, wandering thoughts, and emotional reactions to the speaker's message
- Suspend judgment about the message until you have heard all the facts
- Take notes on the speaker's key points, if appropriate
- Learn to manage your own emotional filters, personal blinders, and biases, which can keep you from hearing what is really being said

✓ Interpret
- Observe the speaker's nonverbal cues (gestures, facial expressions, and tone of voice) and factor them into your interpretation
- Listen for the attitudes and motives behind the words
- Listen for the speaker's needs and wants
- Put the message in a broader context
- Integrate what you've just heard into what you already know about the speaker or subject

✓ Respond
- Nonverbal Response to the Message
  - Make eye contact
  - Nod affirmatively
  - Use facial expressions and gestures to indicate that you are listening
- Verbal Response to the Message
  - Ask questions and probe to get more specific information and ensure understanding
  - Rephrase the message using different words to check the meaning
  - Make empathetic remarks that acknowledge you understand the speaker's feelings, without offering opinions or judging him or her

✓ Evaluate
- Identify the main point of the message and its supporting evidence
- Clarify facts, perceptions, and opinions
- Distinguish between fact and opinion
- Group facts in like categories and logical order (importance, chronology)
- Base your opinion about the message on the facts
- Use the total message – the needs, the context, and the content – to follow through on what you hear

Brainstorming

- Facilitated application specification technique (FAST)
- Group activity
- All members are equal
- Off-site meeting location is preferred

Scenarios

- Scenarios are stories which explain how a system might be used. They should include
  - ✓ A description of the system state before entering the scenario
  - ✓ The normal flow of events in the scenario
  - ✓ Exceptions to the normal flow of events
  - ✓ Information about concurrent activities
  - ✓ A description of the system state at the end of the scenario
- Scenarios are examples of interaction sessions which describe how a user interacts with a system
- Discovering scenarios exposes possible system interactions and reveals system facilities which may be required

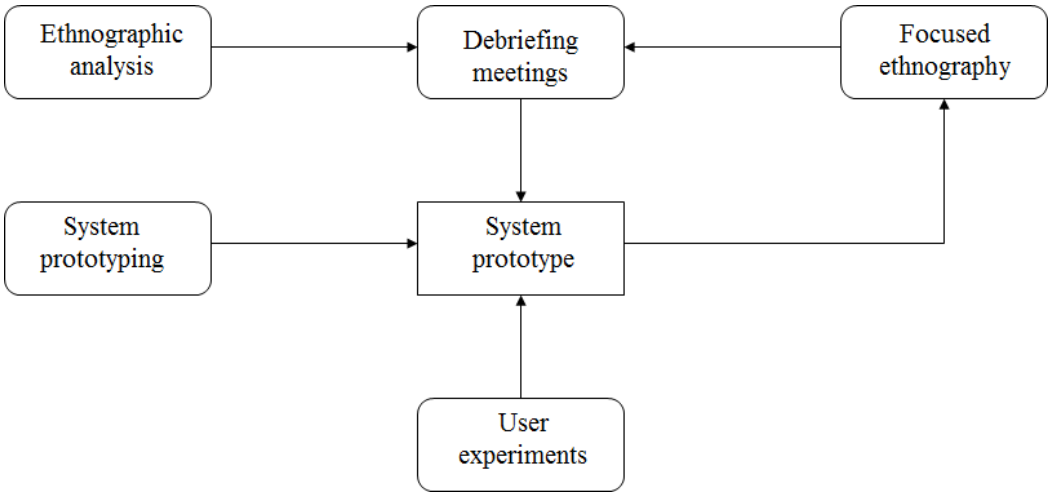
Scenarios and Use-Cases

- The term use-case (i.e., a specific case of system usage) is sometimes used to refer to a scenario
  - ✓ A use-case is a scenario
  - ✓ A scenario is a collection of use-cases. Therefore, each exceptional interaction is represented as a separate use-case
  - ✓ A use-case is a collection of scenarios

Observation and Social Analysis

- People often find it hard to describe what they do because it is so natural to them. Sometimes, the best way to understand it is to observe them at work
- Ethnography is a technique from the social sciences which has proved to be valuable in understanding actual work processes
- Actual work processes often differ from formal, prescribed processes
- An ethnographer spends an extended time observing people at work and building up a picture of how work is done

Ethnography in Requirements Elicitation



Ethnography Guidelines

- Assume that people are good at doing their job and look for non-standard ways of working
- Spend time getting to know the people and establish a trust relationship
- Keep detailed notes of all work practices. Analyze them and draw conclusions from them
- Combine observation with open-ended interviewing
- Organize regular de-briefing session where the ethnographer talks with people outside the process
- Combine ethnography with other elicitation techniques

Requirements Reuse

- Reuse involves taking the requirements which have been developed for one system and using them in a different system
- Requirements reuse saves time and effort as reused requirements have already been analyzed and validated in other systems
- Currently, requirements reuse is an informal process but more systematic reuse could lead to larger cost savings

Reuse Possibilities

- Where the requirement is concerned with providing application domain information
- Where the requirement is concerned with the style of information presentation. Reuse leads to a consistency of style across applications
- Where the requirement reflects company policies such as security policies

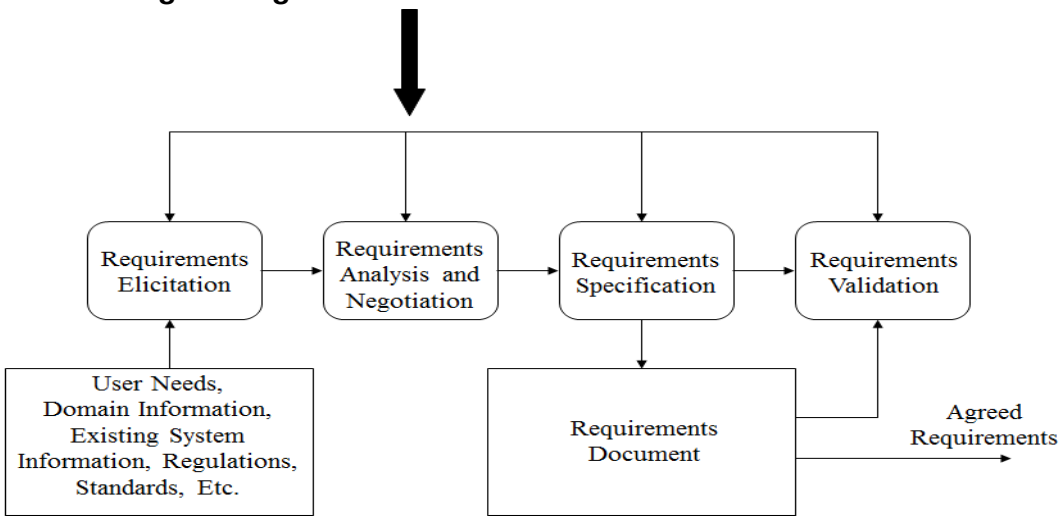
Prototyping

- A prototype is an initial version of a system which may be used for experimentation
- Prototypes are valuable for requirements elicitation because users can experiment with the system and point out its strengths and weaknesses. They have something concrete to criticize

Recap of Requirements Elicitation

- Requirements elicitation deals with discovering requirements for a software product
- It is an iterative process and consists of many activities including establishing objectives, understanding background, organizing knowledge, and collecting requirements
- Introduced the concept of elicitation and requirements elicitation process
- Basics of knowledge acquisition (reading, listening, asking, & observing)
- Knowledge acquisition techniques (individual, group, modeling, cognitive)
- Elicitation problems (scope, understandability, volatility)
- Context (organization, environment, project, constraints imposed by people)
- Guidelines for knowledge acquisition
- Discussed in detail some requirements elicitation techniques, especially interviews

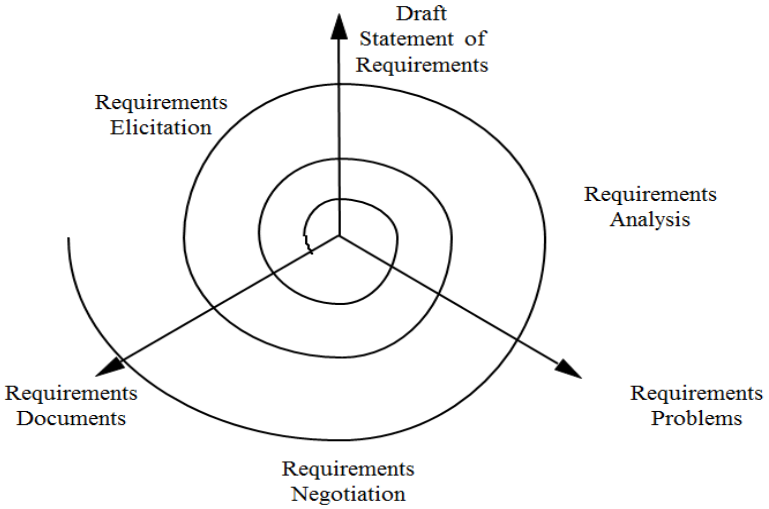
Requirements Engineering Process



RE Process 2: Requirements Analysis and Negotiation

- We'll discuss requirements analysis and negotiation separately, in order to understand them clearly and to appreciate that different skills are needed to perform them
- They are inter-leaved activities and join to form a major activity of the requirements engineering process
- The aim of requirements analysis is to discover problems with the system requirements, especially incompleteness and inconsistencies
- Some analysis is inter-leaved with requirements elicitation as problems are sometimes obvious as soon as a requirement is expressed
- Detailed analysis usually takes place after the initial draft of the requirements document is produced
- Analysis is concerned with incomplete set of requirements, which has not been discussed by stakeholders

Iterative Aspects of Elicitation, Analysis, and Negotiation



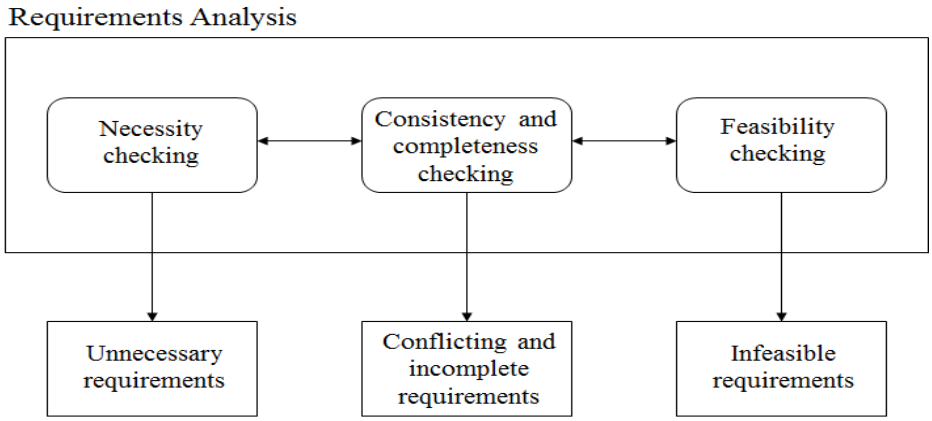
Comments on Requirements Analysis

- Analysts read the requirements, highlight problems, and discuss them in requirements review meetings
- This is a time-consuming and expensive activity
- Analysts have to think about implications of the draft statements of requirements
- People do not think in the same way and different analysts tackle the process in different ways
- It is not possible to make this activity a structured and systematic process
- It depends on the judgment and experience of process participants

Requirements Analysis Stages

- Necessity checking
  - ✓ The need for the requirement is analyzed. In some cases, requirements may be proposed which don't contribute to the business goals of the organization or to the specific problem to be addressed by the system
- Consistency and completeness checking
  - ✓ The requirements are cross-checked for consistency and completeness. Consistency means that no requirements should be contradictory; Completeness means that no services or constraints which are needed have been missed out
- Feasibility checking
  - ✓ The requirements are checked to ensure that they are feasible in the context of the budget and schedule available for the system development

Requirements Analysis Process



Analysis Techniques

- Analysis checklists
  - ✓ A checklist is a list of questions which analysts may use to assess each requirement
- Interaction matrices
  - ✓ Interaction matrices are used to discover interactions between requirements and to highlight conflicts and overlaps

Analysis Checklists

- Each requirement may be assessed against the checklist
- When potential problems are discovered, these should be noted carefully
- They can be implemented as a spreadsheet, where the rows are labeled with the requirements identifiers and columns are the checklist items
- The are useful as they provide a reminder of what to look for and reduce the chances that you will forget some requirements checks
- They must evolve with the experience of the requirements analysis process
- The questions should be general, rather than restrictive, which can be irrelevant for most systems
- Checklists should not include more than ten items, because people forget items on long checklists reading through a document. Example of analysis checklist

Checklist Items

- Premature design
- Combined requirements
- Unnecessary requirements
- Use of non-standard hardware
- Conformance with business goals
- Requirements ambiguity
- Requirements realism
- Requirements testability

Checklist Items Description

- Premature design
  - ✓ Does the requirement include premature design or implementation information?
- Combined requirements
  - ✓ Does the description of a requirement describe a single requirement or could it be broken down into several different requirements?
- Unnecessary requirements
  - ✓ Is the requirement 'gold plating'? That is, is the requirement a cosmetic addition to the system which is not really necessary
- Use of non-standard hardware
  - ✓ Does the requirement mean that non-standard hardware or software must be used? To make this decision, you need to know the computer platform requirements
- Conformance with business goals
  - ✓ Is the requirement consistent with the business goals defined in the introduction to the requirements document?
- Requirements ambiguity
  - ✓ Is the requirement ambiguous i.e., could it be read in different ways by different people? What are the possible interpretations of the requirement?
- Requirements realism
  - ✓ Is the requirement realistic given the technology which will be used to implement the system?
- Requirements testability
  - ✓ Is the requirement testable, that is, is it stated in such a way that test engineers can derive a test which can show if the system meets that requirement?

Requirements Interactions

- A very important objective of requirements analysis is to discover the interactions between requirements and to highlight requirements conflicts and overlaps
- A requirements interaction matrix shows how requirements interact with each other, which can be constructed using a spreadsheet
- Each requirement is compared with other requirements, and the matrix is filled as follows:
  - ✓ For requirements which conflict, fill in a 1
  - ✓ For requirements which overlap, fill in a 1000
  - ✓ For requirements which are independent, fill in a 0
- Consider an example
- An Interaction Matrix

Requirement	R1	R2	R3	R4	R5	R6
R1	0	0	1000	0	1	1
R2	0	0	0	0	0	0
R3	1000	0	0	1000	0	1000
R4	0	0	1000	0	1	1
R5	1	0	0	1	0	0
R6	1	0	1000	1	0	0

Comments on Interaction Matrices

- If you can't decide whether requirements conflict, you should assume that a conflict exists. If an error is made it is usually fairly cheap to fix; it can be much more expensive to resolve undetected conflicts
- In the example, we are considering, we can see that R1 overlaps with R3 and conflicts with R5 and R6
- R2 is an independent requirement
- R3 overlaps with R1, R4, and R6
- The advantage of using numeric values for conflicts and overlaps is that you can sum each row and column to find the number of conflicts and the number of overlaps
- Requirements which have high values for one or both of these figures should be carefully examined
- A large number of conflicts or overlaps means that any changes to that requirement will probably have a major impact of the rest of the requirements
- Interaction matrices work only when there is relatively small number of requirements, as each requirement is compared with every other requirement
- The upper limit should be about 200 requirements
- These overlaps and conflicts have to be discussed and resolved during requirements negotiation, which we'll discuss next

Requirements Negotiation

- Disagreements about requirements are inevitable when a system has many stakeholders. Conflicts are not 'failures' but reflect different stakeholder needs and

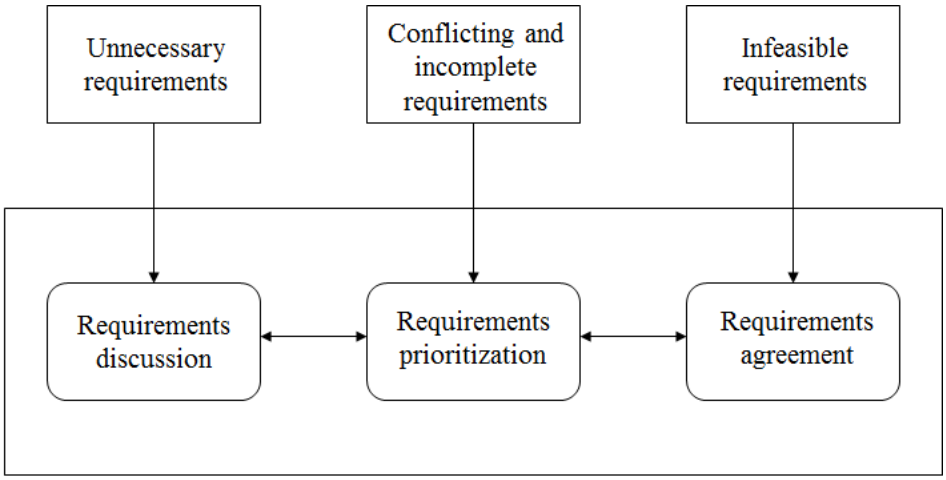
priorities

- Requirements negotiation is the process of discussing requirements conflicts and reaching a compromise that all stakeholders can agree to
- In planning a requirements engineering process, it is important to leave enough time for negotiation. Finding an acceptable compromise can be time-consuming
- The final requirements will always be a compromise which is governed by the needs of the organization in general, the specific requirements of different stakeholders, design and implementation constraints, and the budget and schedule for the system development

Requirements Negotiation Stages

- Requirements discussion
  - ✓ Requirements which have been highlighted as problematic are discussed and the stakeholders involved present their views about the requirements
- Requirements prioritization
  - ✓ Disputed requirements are prioritized to identify critical requirements and to help the decision making process
- Requirements agreement
  - ✓ Solutions to the requirements problems are identified and a compromised set of requirements are reached. Generally, this will involve making changes to some of the requirements

Requirements Negotiation Process



Requirements Negotiation

Comments on Requirements Negotiation

- In principle, requirements negotiation should be an objective process
- The judgments should be the requirements for the system should be based on technical and organizational needs
- Reality is, however, often different

- Negotiations are rarely conducted using only logical and technical arguments
- They are influenced by organizational and political considerations, and the personalities of the people involved
- A strong personality may force their priorities on other stakeholders
- Requirements may be accepted or rejected because they strengthen the political influence in the organization of some stakeholders
- End-users may be resistant to change and may block requirements, etc.
- The majority of time in requirements negotiation is usually spent resolving requirements conflicts. A requirement conflicts with other requirements if they ask for different things
- Example of access of data in a distributed system
- Even after years of experience, many companies do not allow enough time for resolution of conflicts in requirements
- Conflicts should not be viewed as 'failures', they are natural and inevitable – rather healthy

Resolution of Requirements Conflicts

- Meetings are the most effective way to negotiate requirements and resolve requirements conflicts
- All requirements which are in conflict should be discussed individually
- Negotiation meetings should be conducted in three stages

Stages of Negotiation Meetings

- Information stage
  - ✓ An information stage where the nature of the problems associated with a requirement is explained
- Discussion stage
  - ✓ A discussion stage where the stakeholders involved discuss how these problems might be resolved
    - All stakeholders with an interest in the requirement should be given the opportunity to comment. Priorities may be assigned to requirements at this stage
- Resolution stage
  - ✓ A resolution stage where actions concerning the requirement are agreed
    - These actions might be to delete the requirement, to suggest specific modifications to the requirement or to elicit further information about the requirement

Requirements Error/Defect

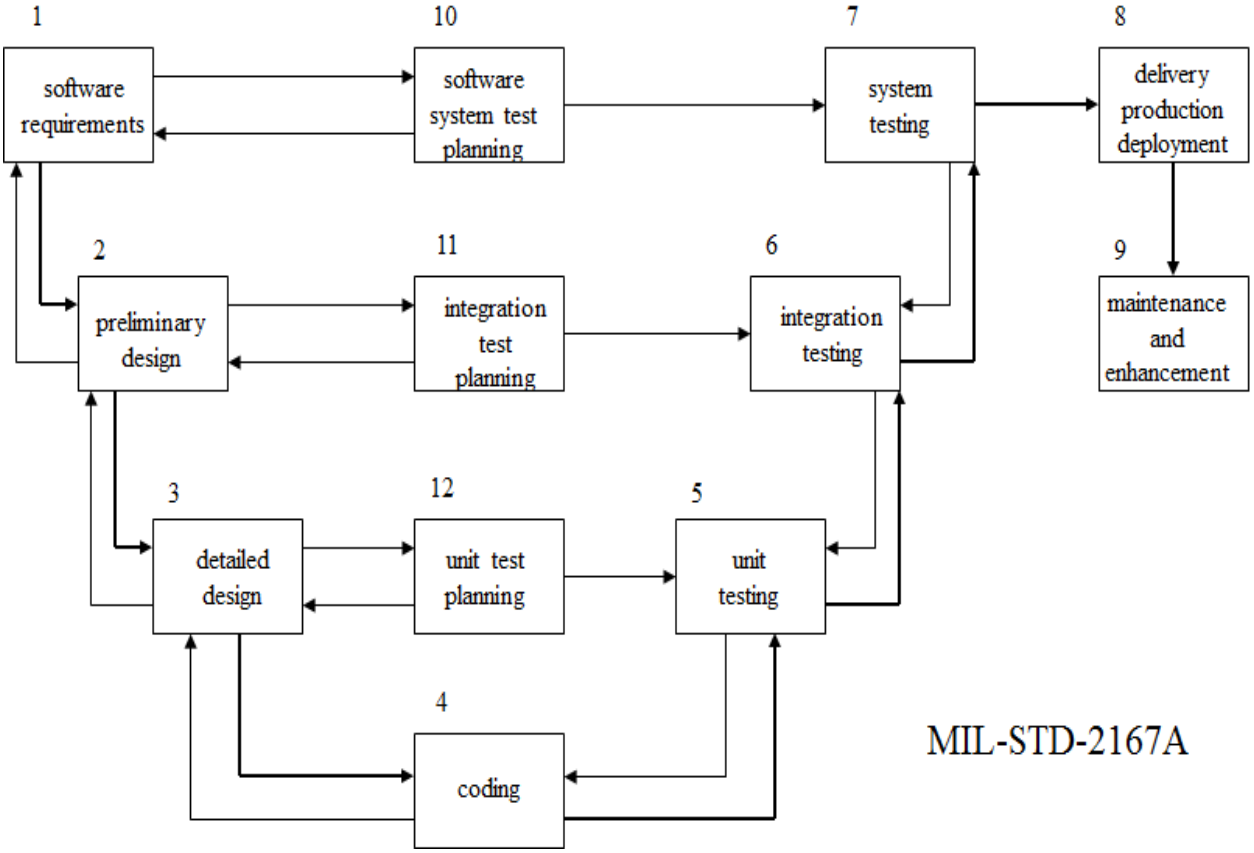
- A deficiency in the requirements quality that can hamper software development

Requirements Errors

- Errors and omissions find their way in different requirements documents
- If not removed, requirements errors usually flow downstream into design, code, and user manuals

- It is difficult to detect requirements errors once they flow downstream
- Requirements errors are most expensive to eliminate

Software Development Process



Types of Requirements Errors

- Errors of omission
  - ✓ Errors of omission are most common among requirements errors
  - ✓ Domain experts easily forget to convey domain knowledge to requirements engineers, because they consider that to be obvious and implicit
- Errors of clarity and ambiguity
  - ✓ Second most common errors are those of clarity and ambiguity
  - ✓ Primarily, because natural languages (like English) are used to state requirements, while such languages are themselves ambiguous
  - ✓ For example: object
- Errors of commission
  - ✓ Errors of commission can also find their way into the requirements documents
- Errors of speed and capacity
  - ✓ Performance, that is errors of speed and capacity, are also found in requirements
  - ✓ Primarily, these occur due to conflicting understanding or competing needs of different stakeholders

Negative Impact of Requirements Errors

- The resulting software may not satisfy user's real needs
- Multiple interpretations of requirements may cause disagreements between customers and developers, wasting time and money, and perhaps resulting in lawsuits
- Negative impact on humans
  - ✓ Unsatisfied customers and developers
  - ✓ Lack of interest in automation of processes
  - ✓ Blame game

Addressing Requirements Errors

- Prevention
- Removal

Prevention vs. Removal

- For requirements errors, prevention is usually more effective than removal
- Joint application development (JAD), quality function deployment (QFD), and prototyping are more effective in defect prevention
- Requirements inspections and prototyping play an important role in defect removal

Defect Prevention

- Don't let defects/errors become part of the requirements document or requirements model in the first place
- How is it possible?
- Understanding application domain and business area is the first step in defect prevention
- Training in different requirements engineering activities (elicitation, analysis and negotiation, specification, and validation) is also very important for defect prevention
- Allocating enough time to conduct requirements engineering activities also is very important in this regard
- Willing and active participation of stakeholders in different activities of requirements engineering. That is why JAD is very useful in defect prevention as far as requirements errors are concerned
- An overall commitment to quality and emphasis on using documented processes is also a very important
- An overall commitment to process improvement

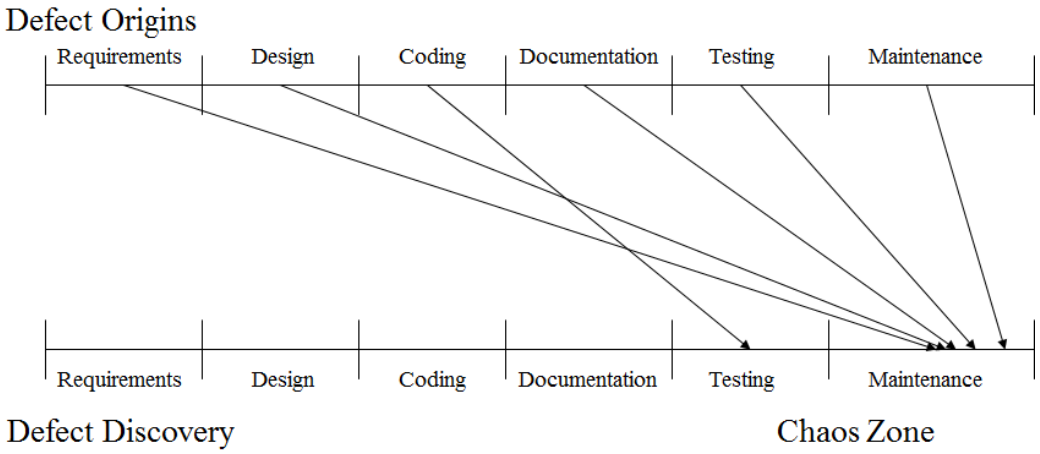
Inspections

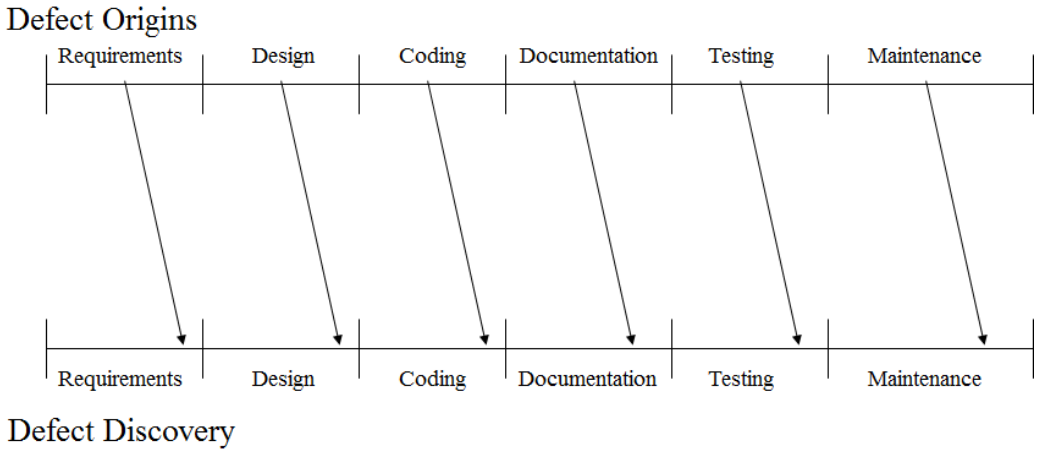
- Inspections, by all accounts, do a better job of error removal than any competing technology, and they do it at a lower cost
  - ✓ Robert Glass
- Inspections are conducted by a group of people working on the project, with the objective to remove defects or errors
- Every member of the inspection team has to read and evaluate requirements documents before coming to the meeting and a formal meeting is conducted to discuss

requirements errors

- Requirements errors detected during this inspections save lot of money and time as requirements errors do not flow into the design and development phases of software development process
- A complete description of inspections must address five dimensions:
  - ✓ Technical
  - ✓ Managerial
  - ✓ Organizational
  - ✓ Assessment
  - ✓ Tool support

Defect Detection Without Inspections





Observations

- Requirements engineers are trained to write requirements documents, but have no training on reading/reviewing requirements documents
- Reviewers typically rely on ad hoc reading techniques, with no well-defined procedure, learning largely by doing

Techniques for Reading Requirements Documents

- Ad hoc review
  - ✓ A review with no formal, systematic procedure, based only individual experience
- Checklist review
  - ✓ A list of items is provided to reviewers, which makes this inspection process more focused
- Defect-based reading
  - ✓ Provides a set of systematic procedures that reviewers can follow, which are tailored to the formal software cost reduction notation
- Perspective-based reading
  - ✓ Researchers at Experimental Software Engineering Group at the University of Maryland, College Park, have created Perspective-Based Reading (PBR) to provide a set of software reading techniques for finding defects in English-language requirements documents

Different Perspectives

- PBR operates under the premise that different information in the requirements is more or less important for the different uses of the document
- Each user of the requirements document finds different aspects of the requirements important for accomplishing a particular task
- PBR provides a set of individual reviews, each from a particular requirements user's point of view, that collectively cover the document's relevant aspects
- This process is similar to constructing system use cases, which requires identifying who will use the system and in what way

Steps in PBR

- Selecting a set of perspectives for reviewing the requirements document
- Creating or tailoring procedures for each perspective usable for building a model of the relevant requirements information
- Augmenting each procedure with questions for finding defects while creating the model
- Applying procedures to review the document

Two Questions

- What information in these documents should they check?
- How do they identify defects in that information?

Benefits of Different Perspectives

- Systematic
  - ✓ Explicitly identifying the different uses for the requirements gives reviewers a definite procedure for verifying whether those uses are achievable
- Focused
  - ✓ PBR helps reviewers concentrate more effectively on certain types of defects, rather than having to look for all types

- Goal-oriented and customizable
  - ✓ Reviewers can tailor perspectives based on the current goals of the organization
- Transferable via training
  - ✓ PBR works from a definite procedure, and not the reviewer's own experience with recognizing defects, new reviewers can receive training in the procedures' steps

Identifying Defects

- A series of questions are used to identify different types of requirements defects
- Requirements that do not provide enough information to answer the questions usually do not provide enough information to support the user. Thus, reviewers can identify and fix defects beforehand

Requirements Defects that PBR Helps Detect

- Missing information
  - ✓ Any significant requirement related to functionality, performance, design constraints, attributes, or external interface not included
  - ✓ Undefined software responses to all realizable classes of input data in all realizable classes of situations
  - ✓ Sections of the requirements document
  - ✓ Figure labels and references, tables, and diagrams
  - ✓ Definitions of terms and units of measures
- Ambiguous information
  - ✓ Multiple interpretations caused by using multiple terms for the same characteristic or multiple meanings of a term in a particular context
- Inconsistent information
  - ✓ Two or more requirements that conflict with one another
- Incorrect fact
  - ✓ A requirement-asserted fact that cannot be true under the conditions specified for the system
- Extraneous information
  - ✓ Unnecessary or unused information (at best, it is irrelevant; at worst, it may confuse requirements users)
- Miscellaneous defects
  - ✓ Other errors, such as including a requirement in the wrong section

Benefits of PBR's Detailed Questions

- Allow controlled improvement
  - ✓ Reviewers can reword, add, or delete specific questions
- Allow training
  - ✓ Reviewers can train to better understand the parts of a representation or work product that correspond to particular questions

PBR General Questions

- Does the requirement make sense from what you know about the application or from what is specified in the general description?
- Do you have all the information necessary to identify the inputs to the requirement? Based on the general requirements and your domain knowledge, are these inputs correct for this requirement?
- Have any of the necessary inputs been omitted?
- Are any inputs specified that are not needed for this requirement?
- Is this requirement in the appropriate section of the document?

Results of PBR Experiments

- PBR provides a framework that represents an improved approach for conducting requirements reviews
- This approach will only be effective when an organization tailors the framework to its own needs and uses feedback from its reviewers to continually improve and refine the techniques
- PBR seems best suited for reviewers with a certain range of experience (not too little; not too much)
- Development teams that use PBR to inspect requirements documents tend to detect more defects than they do using other less- structured approaches
- Relatively novice reviewers can use PBR techniques to apply their expertise in other development tasks to defect detection
- Using PBR improves team meeting by helping team members build up expertise in different aspects of a requirements document
- It creates high-level representations of the software system, usable as a basis of work products in later stages of the development
- Each development organization can customize PBR's set of perspectives, level of detail, and types of questions
- PBR facilitates controlled improvements, providing a definite procedure, alterable according to projects metrics and reviewers' feedback

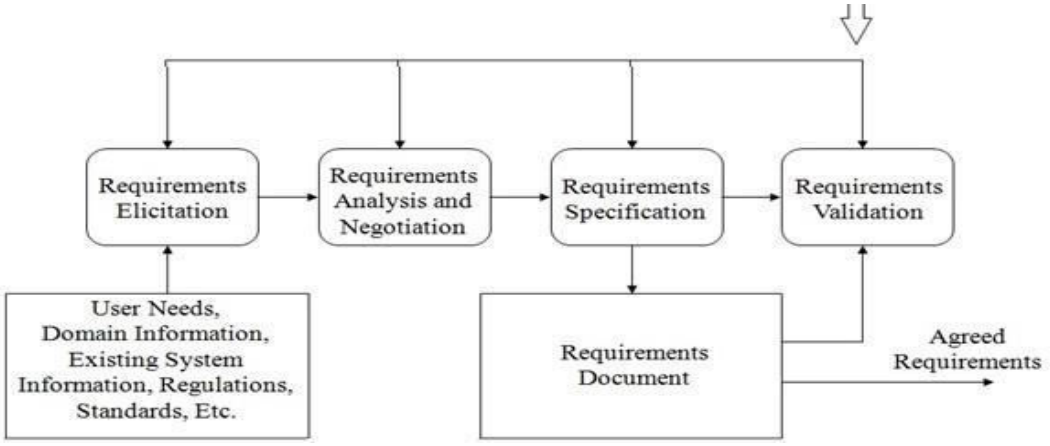
PBR General Questions

- Does the requirement makes sense from what you know about the application or from what is specified in the general description?
- Do you have all the information necessary to identify the inputs to the requirement? Based on the general requirements and your domain knowledge, are these inputs correct for this requirement?
- Have any of the necessary inputs been omitted?
- Are any inputs specified that are not needed for this requirement?
- Is this requirement in the appropriate section of the document?

Results of PBR Experiments

- PBR provides a framework that represents an improved approach for conducting requirements reviews
- This approach will only be effective when an organization tailors the framework to its own needs and uses feedback from its reviewers to continually improve and refine the techniques
- PBR seems best suited for reviewers with a certain range of experience (not too little; not too much)
- Development teams that use PBR to inspect requirements documents tend to detect more defects than they do using other less- structured approaches
- PBR seems best suited for reviewers with a certain range of experience (not too little; not too much)
- Development teams that use PBR to inspect requirements documents tend to detect more defects than they do using other less- structured approaches
- It creates high-level representations of the software system, usable as a basis of work products in later stages of the development
- Each development organization can customize PBR's set of perspectives, level of detail, and types of questions
- PBR facilitates controlled improvements, providing a definite procedure, alterable according to projects metrics and reviewers' feedback

Requirements Engineering Process



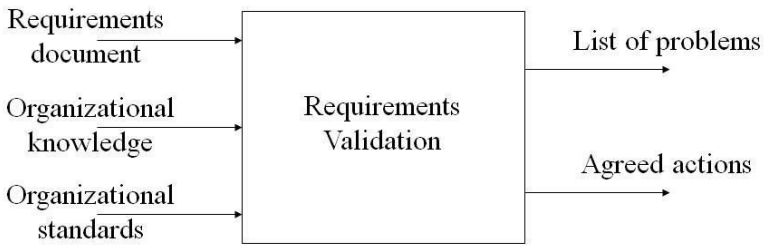
Validation Objectives

- Certifies that the requirements document is an acceptable description of the system to be implemented
- Checks a requirements document for
  - ✓ Completeness and consistency
  - ✓ Conformance to standards
  - ✓ Requirements conflicts
  - ✓ Technical errors
  - ✓ Ambiguous requirements

Analysis and Validation

- Analysis works with raw requirements as elicited from the system stakeholders
  - ✓ "Have we got the right requirements" is the key question to be answered at this stage
- Validation works with a final draft of the requirements document i.e., with negotiated and agreed requirements
  - ✓ "Have we got the requirements right" is the key question to be answered at this stage

Validation Inputs and Outputs

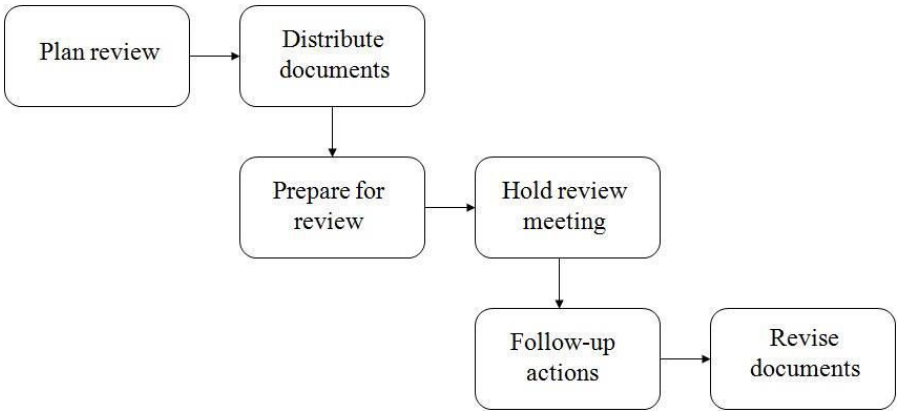


- Requirements Document
  - ✓ Should be a complete version of the document, not an unfinished draft. Formatted and organized according to organizational standards
- Organizational Knowledge
  - ✓ Knowledge, often implicit, of the organization which may be used to judge the realism of the requirements
- Organizational Standards
  - ✓ Local standards e.g. for the organization of the requirements document
- List of Problems
  - ✓ List of discovered problems in the requirements document
- Agreed Actions
  - ✓ List of agreed actions in response to requirements problems. Some problems may have several corrective actions; some problems may have no associated actions

Requirements Reviews

- A group of people read and analyze the requirements, look for problems, meet and discuss the problems and agree on actions to address these problems

Requirements Review Process



Review Activities

- Plan review
  - ✓ The review team is selected and a time and place for the review meeting is chosen
- Distribute documents
  - ✓ The requirements document is distributed to the review team members
- Plan review
  - ✓ The review team is selected and a time and place for the review meeting is chosen
- Distribute documents
  - ✓ The requirements document is distributed to the review team members
- Plan review
  - ✓ The review team is selected and a time and place for the review meeting is chosen
- Distribute documents
  - ✓ The requirements document is distributed to the review team members

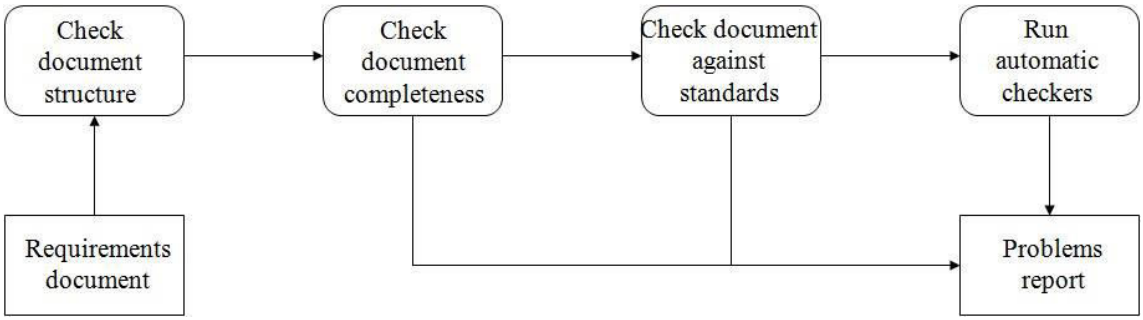
Problem Actions

- Requirements clarification
  - ✓ The requirement may be badly expressed or may have accidentally omitted information which has been collected during requirements elicitation
- Missing information
  - ✓ Some information is missing from the requirements document. It is the responsibility of the requirements engineers who are revising the document to discover this information from system stakeholders
- Requirements conflict
  - ✓ There is a significant conflict between requirements. The stakeholders involved must negotiate to resolve the conflict
- Unrealistic requirement
  - ✓ The requirement does not appear to be implement-able with the technology available or given other constraints on the system. Stakeholders must be consulted to decide how to make the requirement more realistic

Pre-review Checking

- Reviews are expensive because they involve a number of people spending time reading and checking the requirements document
- Reviews are expensive because they involve a number of people spending time reading and checking the requirements document

Pre-review Checking Stages



Review Team Membership

- Reviews should involve a number of stakeholders drawn from different backgrounds
  - ✓ People from different backgrounds bring different skills and knowledge to the review
  - ✓ Stakeholders feel involved in the RE process and develop an understanding of the needs of other stakeholders
- Review team should always involve at least a domain expert and an end-user

Review Checklists

- Understandability
  - ✓ Can readers of the document understand what the requirements mean?
- Redundancy
  - ✓ Is information unnecessarily repeated in the requirements document?
- Completeness
  - ✓ Does the checker know of any missing requirements or is there any information missing from individual requirement descriptions?
- Ambiguity
  - ✓ Are the requirements expressed using terms which are clearly defined? Could readers from different backgrounds make different interpretations of the requirements?
- Consistency
  - ✓ Do the descriptions of different requirements include contradictions? Are there contradictions between individual requirements and overall system requirements?
- Organization
  - ✓ Is the document structured in a sensible way? Are the descriptions of requirements organized so that related requirements are grouped?

- Conformance to standards
  - ✓ Does the requirements document and individual requirements conform to defined standards? Are departures from the standards, justified?
- Traceability
  - ✓ Are requirements unambiguously identified, include links to related requirements and to the reasons why these requirements have been included?

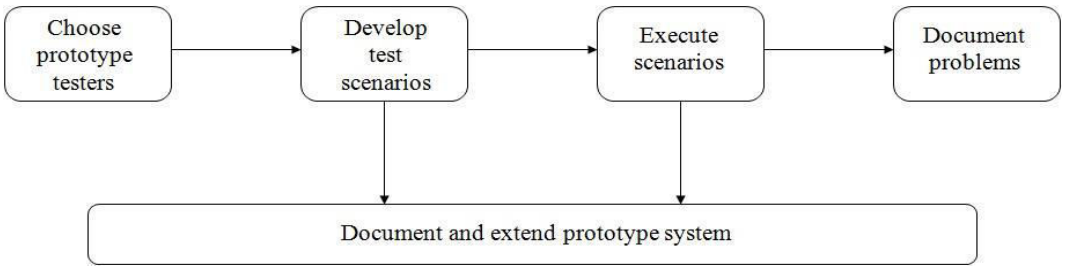
Checklist Questions & Quality Attributes

- Is each requirement uniquely identified?
  - ✓ Traceability, conformance to standards
- Are specialized terms defined in the glossary
  - ✓ Understandability
- Does a requirement stand on its own or do you have to examine other requirements to understand what it means?
  - ✓ Understandability, completeness
- Do individual requirements use the terms consistently
  - ✓ Ambiguity
- Is the same service requested in different requirements? Are there any contradictions in these requests?
  - ✓ Consistency, redundancy
- If a requirement makes reference to some other facilities, are these described elsewhere in the document?
  - ✓ Completeness
- Are related requirements grouped together? If not, do they refer to each other?
  - ✓ Organization, traceability

Prototyping

- Prototypes for requirements validation demonstrate the requirements and help stakeholders discover problems
- Validation prototypes should be complete, reasonably efficient and robust. It should be possible to use them in the same way as the required system
- User documentation and training should be provided

Prototyping for Validation



Prototyping Activities

- Choose prototype testers
  - ✓ The best testers are users who are fairly experienced and who are open-minded about the use of new systems. End-users who do different jobs should be involved so that different areas of system functionality will be covered
- Develop test scenarios
  - ✓ Careful planning is required to draw up a set of test scenarios which provide broad coverage of the requirements. End-users shouldn't just play around with the system as this may never exercise critical system features
- Execute scenarios
  - ✓ The users of the system work, usually on their own, to try the system by executing the planned scenarios
- Document problems
  - ✓ Its usually best to define some kind of electronic or paper problem report form which users fill in when they encounter a problem

User Manual Development

- Writing a user manual from the requirements forces a detailed requirements analysis and thus can reveal problems with the document
- Information in the user manual
  - ✓ Description of the functionality and how it is implemented
  - ✓ Which parts of the system have not been implemented
  - ✓ How to get out of trouble
  - ✓ How to install and get started with the system

System Models

- For some projects, system models may be developed based on the agreed set of requirements
- These models may be data-flow models of the system's functionality, object models, event models, entity-relation models

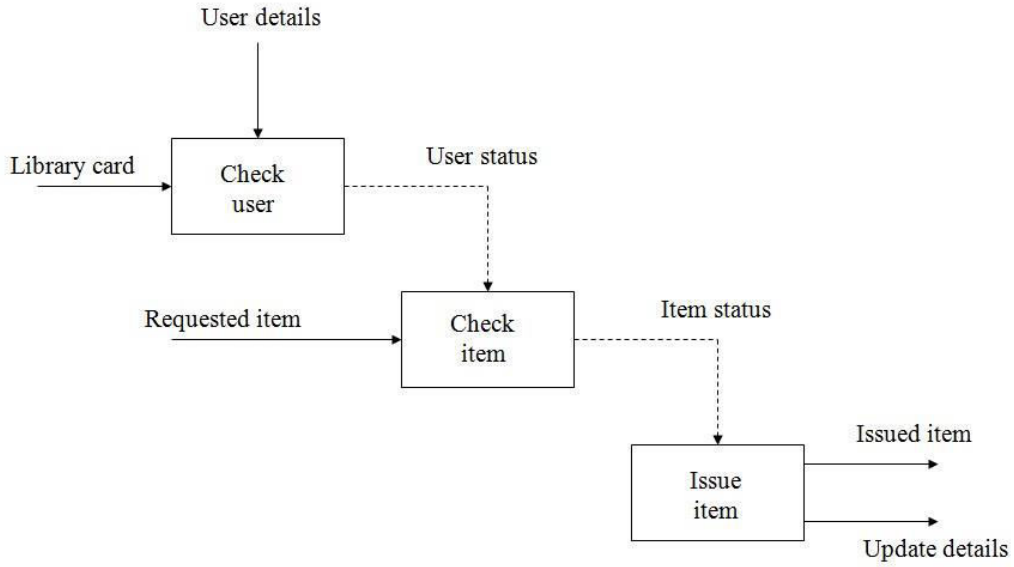
Model Validation

- Validation of system models is an essential part of the validation process
- Some checking is possible with automated tools
- Paraphrasing the model is an effective checking technique

Objectives of Model Validation

- To demonstrate that each model is self-consistent
- If there are several models of the system, to demonstrate that these are internally and externally consistent
- To demonstrate that the models accurately reflect the real requirements of system stakeholders. This is very difficult

Data-flow Diagram for Issue



Paraphrased Description

Check user	
Inputs and sources	User's library card from end-user
Transformation function	Checks that the user is a valid library user
Transformation outputs	The user's status
Control information	User details from the database
Check item	
Inputs and sources	The user's status from Check user
Transformation function	Checks if an item is available for issue
Transformation outputs	The item's status
Control information	The availability of the item
Issue item	
Inputs and sources	*None*
Transformation function	Issues an item to the library user. Items are stamped with a return date.
Transformation outputs	The item issued to the end user Database update details
Control information	Item status - items only issued if available

Requirements Testing

- Each requirement should be testable i.e., it should be possible to define tests to check whether or not that requirement has been met
- Inventing requirements tests is an effective validation technique as missing or ambiguous information in the requirements description may make it difficult to formulate tests
- Each functional requirement should have an associated test

Test Case Definition

- What usage scenario might be used to check the requirement?
- Does the requirement, on its own, include enough information to allow a test to be defined?
- Is it possible to test the requirement using a single test or are multiple test cases required?
- Could the requirement be re-stated to make the test cases more obvious?

Hard-to-Test Requirements

- System requirements
  - ✓ Requirements which apply to the system as a whole. In general, these are the most difficult requirements to validate irrespective of the method used as they may be influenced by any of the functional requirements. Tests, which are not executed, cannot test for non-functional system-wide characteristics such as usability
- Exclusive requirements
  - ✓ These are requirements which exclude specific behavior. For example, a requirement may state that system failures must never corrupt the system database. It is not possible to test such a requirement exhaustively
- Some non-functional requirements
  - ✓ Some non-functional requirements, such as reliability requirements, can only be tested with a large test set. Designing this test set does not help with requirements validation

Requirements Management

- The process of managing changes to the requirements for a system
- In this lecture, we'll talk about the reasons for changes in requirements and how to manage them

Requirements Management and Traceability

- Requirements cannot be managed effectively without requirements traceability
  - ✓ A requirement is traceable if you can discover who suggested the requirement, why the requirement exists, what requirements are related to it and how that requirement relates to other information such as systems designs, implementations and user documentation

Change - A Constant

- There is nothing permanent except change
  - ✓ Heraclitus (500 B.C.)
- No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle
- Software is like a sponge due to its susceptibility to change

Changing Requirements

- All stakeholders want to change requirements, due to different reasons
- Studies have shown that very significant percentage of delivered defects can be traced back to changing user requirements
- A major issue in requirements engineering is the rate at which requirements change once the requirements phase has "officially" ended
- This rate is on average 3% per month in the subsequent design phase, and should go down after that
- This rate should come down to 1% per month during coding
- Ideally, this should come down to no changes in testing, however, this is very rare

Sources of Change

- New business or market conditions dictate changes in product requirements or business rules
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by computer-based system
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure
- Budgetary or scheduling constraints cause a redefinition of the system or product

Why All This Modification?

- As time passes, all constituencies know more
  - ✓ About what they need
  - ✓ Which approach would be best
  - ✓ How to get it done and still make money
- Statement of the fact: most changes are justified!

Managing Changing Requirements?

- Following quality assurance mechanisms can limit the damage done by changing requirements
  - ✓ Formal change management procedures
  - ✓ State-of-the-art configuration control tools
  - ✓ Requirements reviews

Main Concerns in Requirements Management

- Managing changes to agreed requirements
- Managing the relationships between requirements
- Managing the dependencies between the requirements document and other documents produced in the systems engineering process

CASE Tools for Requirements Management

- Requirements management involves the collection, storage and maintenance of large amounts of information
- There are now a number of CASE tools available which are specifically designed to support requirements management
- Configuration management tools may be adapted for requirements engineering

Stable and Volatile Requirements

- Requirements changes occur while the requirements are being elicited, analyzed and validated and after the system has gone into service
- Some requirements are more stable, while others may be more subject to change than others
- Stable requirements are concerned with the essence of a system and its application domain. They change more slowly than volatile requirements
- Volatile requirements are specific to the instantiation of the system in a particular environment and for a particular customer

Requirements Change Factors

- Requirements errors, conflicts and inconsistencies
  - ✓ As requirements are analyzed and implemented, errors and inconsistencies emerge and must be corrected. These may be discovered during requirements analysis and validation or later in the development process
- Evolving customer/end-user knowledge of the system
  - ✓ As requirements are developed, customers and end-users develop a better understanding of what they really require from a system
- Technical, schedule or cost problems
  - ✓ Problems may be encountered in implementing a requirement. It may be too expensive or take too long to implement certain requirements

Requirements Change Factors

- Changing customer priorities
  - ✓ Customer priorities change during system development as a result of a changing business environment, the emergence of new competitors, staff changes, etc.
- Environmental changes
  - ✓ The environment in which the system is to be installed may change so that the system requirements have to change to maintain compatibility

- Organizational changes
  - ✓ The organization which intends to use the system may change its structure and processes resulting in new system requirements

Types of Volatile Requirements

- Mutable requirements
  - ✓ These are requirements which change because of changes to the environment in which the system is operating
- Emergent requirements
  - ✓ These are requirements which cannot be completely defined when the system is specified but which emerge as the system is designed and implemented
- Consequential requirements
  - ✓ These are requirements which are based on assumptions about how the system will be used. When the system is put into use, some of these assumptions will be wrong
- Compatibility requirements
  - ✓ These are requirements which depend on other equipment or processes

Requirements Identification

- It is essential for requirements management that every requirement should have a unique identification
- The most common approach is requirements numbering based on chapter/section in the requirements document
- Problems with this are:
  - ✓ Numbers cannot be unambiguously assigned until the document is complete
  - ✓ Assigning chapter/section numbers is an implicit classification of the requirement. This can mislead readers of the document into thinking that the most important relationships are with the requirements in the same section

Requirements Identification Techniques

- Dynamic renumbering
  - ✓ Some word processing systems allow for automatic renumbering of paragraphs and the inclusion of cross-references. As you re-organize your document and add new requirements, the system keeps track of the cross-reference and automatically renumbers your requirement depending on its chapter, section and position within the section
- Database record identification
  - ✓ When a requirement is identified it is entered in a requirements database and a database record identifier is assigned. This database identifier is used in all subsequent references to the requirement
- Symbolic identification
  - ✓ Requirements can be identified by giving them a symbolic name which is associated with

the requirement itself. For example, EFF-1, EFF-2, EFF-3 may be used for requirements which relate to system efficiency

Storing Requirements

- Requirements have to be stored in such a way that they can be accessed easily and related to other system requirements
- Requirements Storage Techniques
  - ✓ In one or more word processor files
  - ✓ In a specially designed requirements database

Word Processor Documents: Advantages

- Requirements are all stored in the same place
- Requirements may be accessed by anyone with the right word processor
- It is easy to produce the final requirements document

Word Processor Documents: Disadvantages

- Requirements dependencies must be externally maintained
- Search facilities are limited
- Not possible to link requirements with proposed requirements changes
- Not possible to have version control on individual requirements
- No automated navigation from one requirement to another

Requirements Database: Advantages

- Good query and navigation facilities
- Support for change and version management

Requirements Database: Disadvantages

- Readers may not have the software/skills to access the requirements database
- The link between the database and the requirements document must be maintained

Requirements Database Choice Factors

- The statement of requirements
  - ✓ If there is a need to store more than just simple text, a database with multimedia capabilities may have to be used
- The number of requirements
  - ✓ Larger systems usually need a database which is designed to manage a very large volume of data running on a specialized database server
- Teamwork, team distribution and computer support
  - ✓ If the requirements are developed by a distributed team of people, perhaps from different organizations, you need a database which provides for remote, multi-site access

- CASE tool use
  - ✓ The database should be the same as or compatible with CASE tool databases. However, this can be a problem with some CASE tools which use their own proprietary database
- Existing database usage
  - ✓ If a database for software engineering support is already in use, this should be used for requirements management

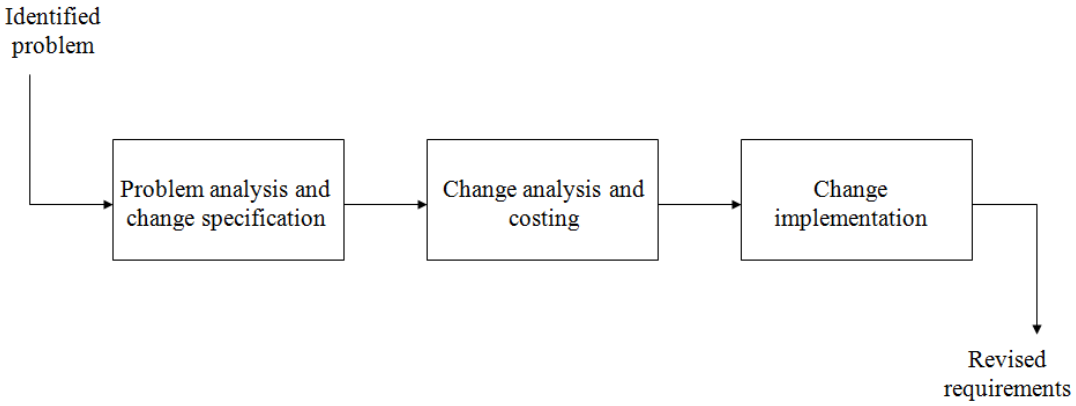
Change Management

- Change management is concerned with the procedures, processes and standards which are used to manage changes to system requirements
- Without formal change management, it is impossible to ensure that proposed changes support business goals

Change Management Policies

- The change request process and the information required to process each change request
- The process used to analyze the impact and costs of change and the associated traceability information
- The membership of the body which formally considers change requests
- The software support (if any) for the change control process

Change Management Stages



Problem Analysis and Change Specification

- Some requirements problem is identified
- This could come from an analysis of the requirements, new customer needs, or operational problems with the system. The requirements are analyzed using problem information and requirements changes are proposed

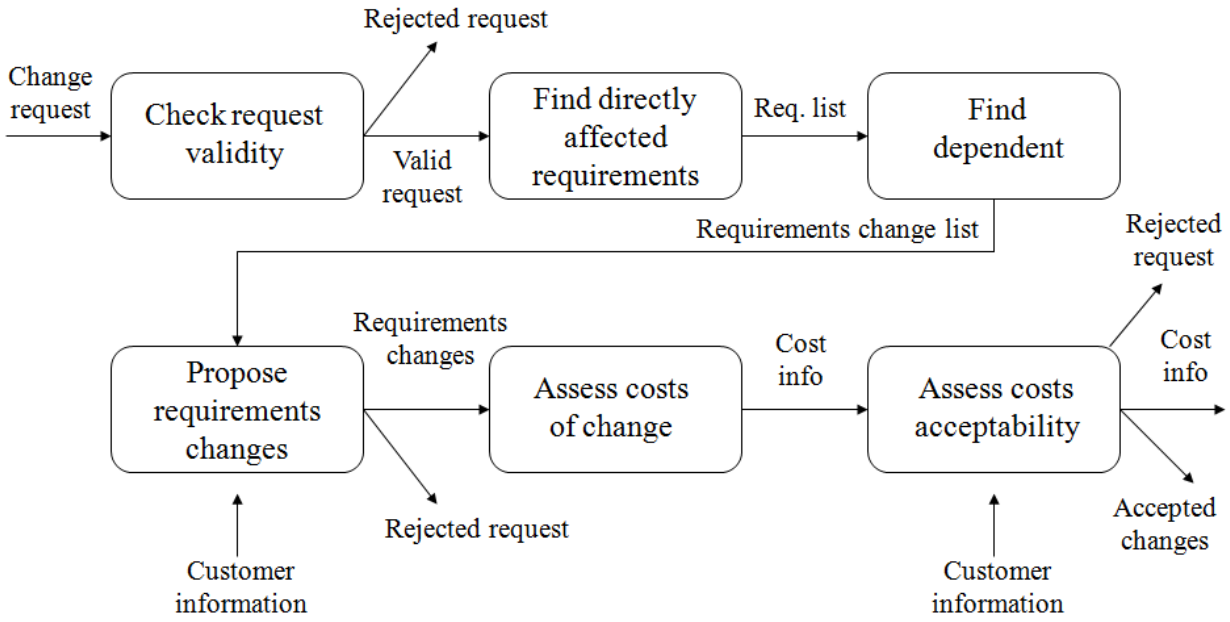
Change Analysis and Costing

- This checks how many requirements (and, if necessary, system components) are affected by the change and roughly how much it would cost, in both time and money, to make the change

Change Implementation

- A set of amendments to the requirements document or a new document version is produced. This should, of course, be validated using whatever normal quality checking procedures are used

Change Analysis and Costing Process



Change Analysis Activities

- The change request is checked for validity. Customers can misunderstand requirements and suggest unnecessary changes
- The requirements which are directly affected by the change are discovered
- Traceability information is used to find dependent requirements affected by the change
- The actual changes which must be made to the requirements are proposed
- The costs of making the changes are estimated.
- Negotiations with customers are held to check if the costs of the proposed changes are acceptable

Change Request Rejection Reasons

- If the change request is invalid. This normally arises if a customer has misunderstood something about the requirements and proposed a change which isn't necessary
- If the change request results in consequential changes which are unacceptable to the user.
- If the cost of implementing the change is too high or takes too long

Change Processing

- Proposed changes are usually recorded on a change request form which is then passed to all of the people involved in the analysis of the change. It may include
  - ✓ Fields to document the change analysis
  - ✓ Data fields
  - ✓ Responsibility fields
  - ✓ Status field
  - ✓ Comments field

Tool Support for Change Management

- May be provided through requirements management tools or through configuration management tools

Tools Features

- Electronic change request forms which are filled in by different participants in the process
- A database to store and manage these forms
- A change model which may be instantiated so that people responsible for one stage of the process know who is responsible for the next process activity
- Electronic transfer of forms between people with different responsibilities and electronic mail notification when activities have been completed
- In some cases, direct links to a requirements database

Requirements Traceability

- Refers to ability to describe and follow the life of a requirement, in both a forwards and backwards direction
- That is from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases

Tracing Requirements

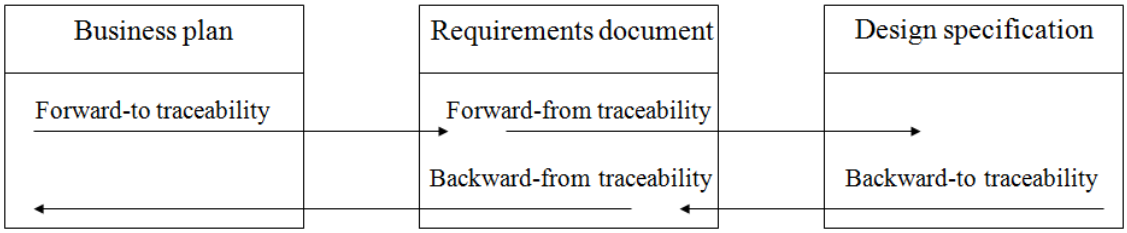
- It is important to trace requirements both ways
  - ✓ Origin of a requirement
  - ✓ How is it implemented
- This is a continuous process

Classifications of Requirements Traceability

- Backward-from traceability
  - ✓ Links requirements to their sources in other documents or people
- Forward-from traceability
  - ✓ Links requirements to design and implementation components
- Backward-to traceability
  - ✓ Links design and implementation components back to requirements
- Forward-to traceability

- ✓ Links other documents (which may have preceded the requirements document) to relevant requirements

Backwards and Forwards Traceability



Categories of Traceability

- Requirements-sources traceability
  - ✓ Links the requirement and the people or documents which specified the requirement
- Requirements-rationale traceability
  - ✓ Links the requirement with a description of why that requirement has been specified. This can be a distillation of information from several sources
- Requirements-requirements traceability
  - ✓ Links requirements with other requirements which are, in some way, dependent on them. This should be a two-way link (dependent on them and is-dependent on)
- Requirements-architecture traceability
  - ✓ Links requirements with the sub-systems where these requirements are implemented. This is particularly important where sub-systems are being developed by different sub-contractors
- Requirements-design traceability
  - ✓ Links requirements with specific hardware or software components in the system, which are used to implement the requirement
- Requirements-interface traceability
  - ✓ Links requirements with the interfaces of external systems, which are used in the provision of the requirements

Traceability Tables

- Requirements traceability information can be kept in traceability tables, each table relating requirements to one or more aspects of the system or its environment
- A Generic Traceability Table

	A01	A02	A03		Aii
R01		✓	✓		
R02	✓	✓			
R03		✓			✓
Rnn	✓	✓			

Need for Traceability Policy

- Huge amount of information, which is expensive to collect, analyze, and update
- Need to continuously update traceability information
- A traceability policy is needed

Traceability Policy

- Traceability information
- Traceability techniques
- When to collect information
- Roles
- Documentation of policy exceptions
- Process of managing information

Traceability Information

- No. of requirements
- Estimated lifetime
- Level of organization's maturity
- Project team and composition
- Type of system
- Specific customer requirements

Basic Types of Requirements Traceability

- Pre-RS traceability
  - ✓ Concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirements production)
- Post-RS traceability
  - ✓ Concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirements deployment)

Pre-RS Traceability

- Depends on the ability to trace requirements from and back to, their originating statements, through the process of requirements production and refinement, in which statements from diverse sources are eventually integrated into a single requirement in the RS
- Changes in the process need to be re-worked into the RS

Post-RS Traceability

- Depends on the ability to trace requirements from, and back to, a baseline (the RS), through a succession of artifacts in which they are distributed
- Changes to the baseline need to be re-propagated through this chain

Pre-RS Traceability and Rationale

- Mostly only Post-RS traceability is considered sufficient
- Pre-RS traceability captures the rationale for each requirement, which is a very important aspect in managing requirements properly

Prototyping

- It is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem

Prototype

- An initial version of the system under development, which is available early in the development process
- A prototype can be a subset of a system, and vice versa, but they are not the same
- In hardware systems, prototypes are often developed to test and experiment with system designs
- In software systems, prototypes are more often used to help elicit and validate the system requirements. There are other uses also
- It should be easy for a prototype to be developed quickly, so that it can be used during the development process
- Prototypes are valuable for requirements elicitation because users can experiment with the system and point out its strengths and weaknesses. They have something concrete to criticize

Types of Prototyping

- Throw-away prototyping
- Evolutionary prototyping

Throw-away Prototyping

- Intended to help elicit and develop the system requirements
- The requirements which should be prototyped are those which cause most difficulties to customers and which are the hardest to understand. Little documentation is needed
- Determine the feasibility of a requirement
- Validate that a particular function is really necessary
- Uncover missing requirements
- Determine the viability of a user interface
- Writing a preliminary requirements document
- Implementing the prototype based on those requirements
- Achieving user experience with prototype
- Writing the real SRS
- Developing the real product

Evolutionary Prototyping

- Intended to deliver a workable system quickly to the customer
- The requirements which should be supported by the initial versions of this prototype are those which are well-understood and which can deliver useful end-user functionality
- Documentation of the prototype is needed to build upon
- This process repeats indefinitely until the prototype system satisfies all needs and has thus evolved into the real system
- Evolutionary prototype may not be built in a ‘dirty’ fashion. The evolutionary prototype evolves into the final product, and thus it must exhibit all the quality attributes of the final product

Comparison of Prototyping

	Throwaway	Evolutionary
Development approach	Quick and dirty. No rigor	No sloppiness. Rigorous
What to build	Build only difficult parts	Build understood parts first. Build on solid foundation
Design drivers	Optimize development time	Optimize modifiability
Ultimate goal	Throw it away	Evolve it

Prototyping Benefit

- The prototype allows users to experiment and discover what they really need to support their work
- Establishes feasibility and usefulness before high development costs are incurred
- Essential for developing the 'look and feel' of a user interface. Helps customers in 'visualizing' their requirements
- Forces a detailed study of the requirements which reveals inconsistencies and omissions

Prototyping Costs

- Training costs
  - ✓ Prototype development may require the use of special purpose tools
- Development costs
  - ✓ Depend on the type of prototype being developed

Prototyping Problems

- Extended development schedules
  - ✓ Developing a prototype may extend the schedule although the prototyping time may be recovered because rework is avoided
- Incompleteness
  - ✓ It may not be possible to prototype emergent system requirements

Additional Benefits of Prototyping

- Developing a system prototype is worth the investment in time and money
- Real needs of the customers will be reflected in the requirements set
- Rework will be reduced
- Defect prevention

Developing Prototypes

- Conventional system development techniques usually take too long, and prototypes are needed early in the elicitation process to be useful
- Rapid development approaches are used for prototype development

Approaches to Prototyping

- Paper prototyping
  - ✓ A paper mock-up of the system is developed and used for system experiments
  - ✓ This is very cheap and very effective approach to prototype development
  - ✓ No executable software is needed
  - ✓ Paper versions of the screens, which might be presented to the user are drawn and various usage scenarios are planned
  - ✓ For interactive systems, this is very effective way to find users' reactions and the required information
- 'Wizard of Oz' prototyping
  - ✓ A person simulates the responses of the system in response to some user inputs
  - ✓ Relatively cheap as only user interface software needs to be developed
  - ✓ The users interact through this user interface software and all requests are channeled to the a person, who simulates the system's responses
  - ✓ This is particularly useful for new systems, which are extensions of existing software systems, and the users are familiar with the existing user interface
  - ✓ The person simulating the system is called 'Wizard of Oz'
- Executable prototyping
  - ✓ A fourth generation language or other rapid development environment is used to develop an executable prototype
  - ✓ This is an expensive option and involves writing software to simulate the functionality of the proposed system
  - ✓ 4GLs based around database systems are useful for developing prototypes, which involve information management
  - ✓ Visual programming languages such as Visual Basic or ObjectWorks
  - ✓ These languages are supported by powerful development environments, which include access to reusable objects and user interface development utilities. Support for database-oriented applications is not that strong
  - ✓ Internet-based prototyping solutions based on WWW browsers and languages. Here, we a ready-made user interface and Java applets can be used to add functionality to the user interface

Comments on Prototyping

- Prototyping interactive applications is easier than prototyping real-time applications
- Prototyping is used better to understand and discover functional requirements, as compared to non-functional requirements

Writing Requirements

- Requirements specification should establish an understanding between customers and suppliers about what a system is supposed to do, and provide a basis for validation and verification
- Typically, requirements documents are written in natural languages (like, English, Japanese, French, etc.)
- Natural languages, by their nature, are ambiguous
- Structured languages can be used with the natural languages to specify requirements

Problems with Natural Languages

- Natural language understanding relies on the specification readers and writers using the same words for same concept
- A natural language requirements specification is over-flexible. You can say the same thing in completely different ways
- It is not possible to modularize natural language requirements. It may be difficult to find all related requirements
    - ✓ To discover the impact of a change, every requirement have to be examined

Problems with Requirements

- The requirements are written using complex conditional clauses (if A then B then C…), which are confusing
- Terminology is used in a sloppy and inconsistent way
- The writers of the requirement assume that the reader has a specific knowledge of the domain or the system and they leave essential information out of the requirements document

Impact of These Problems

- Difficult to check the requirements for errors and omissions
- Different interpretations of the requirements may lead to contractual disagreements between customer and the system developer

Structured Language Specifications

- Structured natural language
- Design description languages
- Graphical notations
- Mathematical notations

Comments on Special-Purpose Languages

- These languages cannot completely define requirements
- They are not understandable by all stakeholders
- Therefore, there is always a need for well-written, natural language statements of requirements

Essentials for Writing Requirements

- Requirements are read more often than they are written. Investing effort in writing requirements, which are easy to read and understand is almost always cost-effective
- Readers of requirements come from diverse backgrounds. If you are requirements writer, you should not assume that readers have the same background and knowledge as you
- Recollect our discussion on cultural issues in requirements engineering
- Writing clearly and concisely is not easy. If you don't allow sufficient time for requirements descriptions to be drafted, reviewed and improved, you will inevitably end up with poorly written requirements
- Different organizations write requirements at different levels of abstraction from deliberately vague product specifications to detailed and precise descriptions of all aspects of a system
- Level of detail needed is dependent on
    - ✓ Type of requirements (stakeholder or process requirements)
    - ✓ Customer expectations
    - ✓ Organizational procedures
    - ✓ External standards or regulations
- Writing good requirements requires a lot of analytic thought
- Specifying rationale of requirement is one way to encourage such thought

Guidelines for Writing Requirements

- Define standard templates for describing requirements
    - ✓ Define a set of standard format for different types of requirements and ensure that all requirement definitions adhere to that format
    - ✓ Standardization means that omissions are less likely and makes requirements easier to read and check
- Use language simply, consistently, and concisely
    - ✓ Use language consistently. In particular, distinguish between mandatory and desirable requirements. It is usual practice to define mandatory requirements using 'shall' and desirable requirements using 'should'. Use 'will' to state facts or declare purpose
    - ✓ Use short sentences and paragraphs, using lists and table
    - ✓ Use text highlighting to pick out key parts of the requirements
- Use diagrams appropriately
    - ✓ Use diagrams to present broad overviews and show relationships between entities
    - ✓ Avoid complex diagrams
- Supplement natural language with other descriptions of requirements
    - ✓ If readers are familiar with other types of descriptions of requirements (like equations, etc.) then use those
    - ✓ Particularly applicable to scientific and engineering domains
- Don't try to write everything in natural language
    - ✓ Specify requirements quantitatively
        - o Specify requirements quantitatively wherever possible

- This is applicable to properties of system, such as reliability or performance
- Recollect our discussion on metrics for non-functional requirements

Additional Guidelines for Writing Requirements

- State only one requirement per requirement statement
- State requirements as active sentences
- Always use a noun or a definite pronoun when referring to a thing
- Do not use more than one conjunction when writing requirements statements
- Avoid using weak words and phrases. Such words and phrases re generally imprecise and allow the expansion or contraction of requirements beyond their intent
- Examples of Words to be Avoided
  - ✓ About, adequate, and/or, appropriate, as applicable, as appropriate, desirable, efficient, etc., if practical, suitable, timely, typical, when necessary
- State the needed requirements without specifying how to fulfill them
- Write complete statements
- Write statements that clearly convey intent

Requirements Document

- The requirements document is a formal document used to communicate the requirements to customers, engineers and managers
- It is also known as software requirements specifications or SRS
- The services and functions which the system should provide
- The constraints under which the system must operate
- Overall properties of the system i.e., constraints on the system's emergent properties
- Definitions of other systems which the system must integrate with
- Information about the application domain of the system, e.g., how to carry out particular types of computation
- Constraints on the process used to develop the system
- It should include both the user requirements for a system and a detailed specification of the system requirements
- In some cases, the user and system requirements may be integrated into one description, while in other cases user requirements are described before (as introduction to) system requirements
- Typically, requirements documents are written in natural languages (like, English, Japanese, French, etc.)
- Natural languages, by their nature, are ambiguous
- Structured languages can be used with the natural languages to specify requirements
- For software systems, the requirements document may include a description of the hardware on which the system is to run
- The document should always include an introductory chapter which provides an overview of the system and the business needs
- A glossary should also be included to document technical terms

- And because multiple stakeholders will be reading documents and they need to understand meanings of different terms
- Also because stakeholders have different educational backgrounds
- Structure of requirements document is also very important and is developed on the basis of following information
    - ✓ Type of the system
    - ✓ Level of detail included in requirements
    - ✓ Organizational practice
    - ✓ Budget and schedule for RE process

Users of Requirements Documents

- System customers
    - ✓ Specify the requirements and read them to check that they meet their needs. They specify changes to the requirements
- Project managers
    - ✓ Use the requirements document to plan a bid for the system and to plan the system development process
- System engineers
    - ✓ Use the requirements to understand what system is to be developed
- System test engineers
    - ✓ Use the requirements to develop validation tests for the system
- System maintenance engineers
    - ✓ Use the requirements to help understand the system and the relationships between its parts

Six Requirements for RS

- It should specify only external behavior
- It should specify constraints on the implementation
- It should be easy to change
- It should serve as a reference tool for system maintainers
- It should record forethought about the lifecycle of the system
- It should characterize acceptable responses to undesired events
    - ✓ Heninger (1980)

How to Organize an SRS?

- Clients/developers may have there own way of organizing an SRS
- US Department of Defense
- NASA
- IEEE/ANSI 830-1993 Standard
    1. Introduction
        - ✓ 1.1 Purpose of the requirements document

- ✓ 1.2 Scope of the product
- ✓ 1.3 Definitions, acronyms, and abbreviations
- ✓ 1.4 References
- ✓ 1.5 Overview of the remainder of the document

2. General description
   - ✓ 2.1 Product perspective
   - ✓ 2.2 Product functions
   - ✓ 2.3 User characteristics
   - ✓ 2.4 General constraints
   - ✓ 2.5 Assumptions and dependencies
3. Specific requirements
   - ✓ Covering functional, non-functional, and interface requirements. These should document external interfaces, functionality, performance requirements, logical database requirements, design constraints, system attributes, and quality characteristics
4. Appendices
5. Index

Comments on IEEE Standard

- It is good starting point for organizing requirements documents
- First two sections are introductory chapters about background and describe the system in general terms
- The third section is the main part of the documents
- The standard recognizes that this section varies considerably depending on the type of the system

Comments on Organization of SRS

- It should be possible to specify different systems
- It should allow for omitting certain sub-sections and also adding new sections
- These variations should be documented also
- Each SRS has some parts, which are stable and some, which are variant
- Stable parts include introductory chapters and glossary, which should appear in all requirements documents
- Variant parts are those chapters, which can be changed depending on the system

An SRS based on IEEE Standard

- Preface
- Introduction
- Glossary
- General user requirements
- System architecture (reusable architectural components)

- Hardware specification
- Detailed software specification
- Reliability and performance requirements
- Appendices
  - ✓ Hardware interface specifications
  - ✓ Reusable components
  - ✓ Data-flow model
  - ✓ Object-model

Specifying Requirements

- Requirements are specified in a document called software requirements specifications (SRS)
- SRS is used for communication among customers, analysts, and designers
- Supports system-testing activities
- Controls the evolution of the system

Validation Objectives

- Certifies that the requirements document is an acceptable description of the system to be implemented
- Checks a requirements document for
  - ✓ Completeness and consistency
  - ✓ Conformance to standards
  - ✓ Requirements conflicts
  - ✓ Technical errors
  - ✓ Ambiguous requirements

What Should Be Included in SRS?

- Functional requirements
  - ✓ They define what the system does. These describe all the inputs and outputs to and from the system as well as information concerning how the inputs and outputs interrelate.
- Non-functional requirements
  - ✓ They define the attributes of a system as it performs its job. They include system's required levels of efficiency, reliability, security, maintainability, portability, visibility, capacity, and standards compliance
  - ✓ Some of these are quality attributes of a software product

What Should Not Be Included in SRS?

- Project requirements (for example, staffing, schedules, costs, milestones, activities, phases, reporting procedures)
- Designs
- Product assurance plans (for example, configuration management plans, verification and validation plans, test plans, quality assurance plans)

SRS Quality Attributes

- Correct
  - ✓ An SRS is correct if and only if every requirement stated therein represents something required of the system to be built
- Unambiguous
  - ✓ An SRS is unambiguous if and only if every requirement stated therein has only one interpretation
  - ✓ At a minimum all terms with multiple meanings must appear in a glossary
  - ✓ All natural languages invite ambiguity
  - ✓ Example of Ambiguity
    - o "Aircraft that are nonfriendly and have an unknown mission or the potential to enter restricted airspace within 5 minutes shall raise an alert"
    - o Combination of "and" and "or" make this an ambiguous requirement
- Complete
  - ✓ An SRS is complete if it possesses the following four qualities
    - o Everything that the software is supposed to do is included in the SRS
    - o Definitions of the responses of the software to all realizable classes of input data in all realizable classes of situations is included
  - ✓ All pages are numbered; all figures and tables are numbered, named, and referenced; all terms and units of measure are provided; and all referenced material and sections are present
  - ✓ No sections are marked
  - ✓ "To Be Determined" (TBD)
- Verifiable
  - ✓ An SRS is verifiable if and only if every requirement stated therein is verifiable. A requirement is verifiable if and only if there exists some finite cost effective process with which a person or machine can check that the actual as-built software product meets the requirement
- Consistent
  - ✓ An SRS is consistent if and only if:
    - o No requirement stated therein is in conflict with other preceding documents, such as specification or a statement of work
    - o No subset of individual requirements stated therein conflict
  - ✓ Conflicts can be any of the following
    - o Conflicting behavior
    - o Conflicting terms
    - o Conflicting characteristics
    - o Temporal inconsistency
- Understandable by customer
  - ✓ Primary readers of SRS in many cases are customers or users, who tend to be experts in an application area but are not necessarily trained in computer science

- Modifiable
  - ✓ An SRS is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently
  - ✓ Existence of index, table of contents, cross-referencing, and appropriate page-numbering
  - ✓ This attribute deals with format and style of SRS
- Traced
  - ✓ An SRS is traced if the origin of its requirements is clear. That means that the SRS includes references to earlier supportive documents
- Traceable
  - ✓ An SRS is traceable if it is written in a manner that facilitates the referencing of each individual requirement stated therein
  - ✓ Techniques for Traceability
    - Number every paragraph hierarchically and never include more than one requirement in any paragraph
    - Assign every requirement a unique number as we have discussed this earlier
    - Use a convention for indicating a requirement, e.g., use shall statement
  - ✓ Traced and Traceability
    - Backward-from-requirements traceability implies that we know why every requirement in the SRS exists
    - Forward-from-requirements traceability implies that we understand which components of the software satisfy each requirement
    - Backward-to-requirements traceability implies that every software component explicitly references those requirements that it helps to satisfy
    - Forward-to-requirements traceability implies that all documents that preceded the SRS can reference the SRS
- Design independent
  - ✓ An SRS is design independent if it does not imply a specific software architecture or algorithm
  - ✓ Sometimes, it is not possible to have no design information due to constraints imposed by the customers or external factors
- Annotated
  - ✓ The purpose of annotating requirements contained in an SRS is to provide guidance to the development organization
  - ✓ Relative necessity
  - ✓ Relative stability
- Concise
  - ✓ The SRS that is shorter is better, given that it meets all characteristics
- Organized
  - ✓ An SRS is organized if requirements contained therein are easy to locate. This implies that requirements are arranged so that requirements that are related are co-related
  - ✓ We had discussed organization of SRS in the last lecture

Phrases to Look for in an SRS

- Always, Every, All, None, Never
- Certainly, Therefore, Clearly, Obviously, Evidently
- Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly
- Etc., And So Forth, And So On, Such As
- Good, Fast, Cheap, Efficient, Small, Stable
- Handled, Processed, Rejected, Skipped, Eliminated
- If…Then…(but missing Else)

The Balancing Act

- Achieving all the preceding attributes in an SRS is impossible
- Once you become involved in writing an SRS, you will gain insight and experience necessary to do the balancing act
- There is no such thing as a perfect SRS

Use Case Modeling

Introduction

- No system exists in isolation. Every interesting system interacts with human or automated actors that use that system for some purpose, and those actors expect that system to behave in predictable ways
- A use case specifies the behavior of a system or part of a system
- It is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
- They are used in requirements elicitation process
- Use cases are applied to capture the intended behavior of the system under development, without having to specify how the behavior is implemented
- They provide a way for developers, end users, and domain experts to come to a common understanding
- Use cases serve to help validate the architecture and to verify the system as it evolves during development
- Use cases are realized by collaborations whose elements work together to carry out each use case
- Well-structured use cases denote essential system or subsystem behaviors only
- They are neither overly general nor too specific

How Things Are Used in Real World?

- A house is built around the needs of occupants
  - ✓ How they will use their house?
- A car is built around the needs of drivers and passengers
  - ✓ How will they use that car?
- This is en example of use case-based analysis

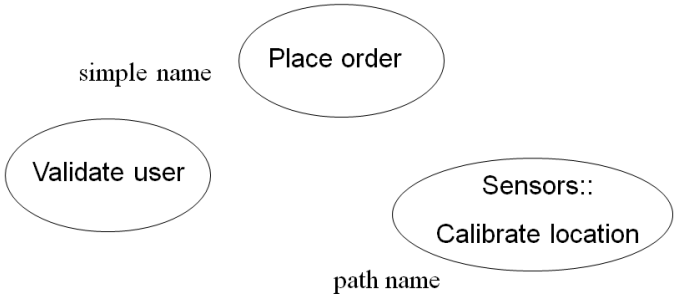
Use Cases

- Use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
- There are a number of important parts to this definition
- A use case describes a set of sequences, in which each sequence represents the interaction of the things outside the system (its actors) with the system itself
- A use case represents a functional requirement of the system as a whole
- An actor represents a coherent set of roles that users of use cases play when interacting with these use cases
- Actors can be human or they can be automated systems
- One central use case of a bank is to process loans
- In modeling a bank, processing a loan involves, among other things, the interaction between a customer and a loan officer
- A use case may have variants
- In all interesting systems, you will find use cases that are specialized versions of other use cases, use cases that are included as part of other use cases, and use cases that extend the behavior of other core use cases
- You can factor the common, reusable behavior of a set of use cases by organizing them according to these three kinds of relationships
- A use case carries out some tangible amount of work
- From the perspective of a given actor, a use case does something that's of value to an actor
- In modeling a bank, a processing of loan results in the delivery of an approved loan
- Use cases can be applied to the whole system, part of a system (including subsystems), and even individual classes and interface
- Use cases not only represent the desired behavior, but can also be used as the basis of test cases for these elements as they evolve during development
- Use cases applied to whole system are excellent sources of integration and system tests
- Use cases applied to subsystems are excellent sources of regression tests
- The UML provides a graphical representation of a use case (an eclipse) and an actor (a stick figure)

Use Case Names

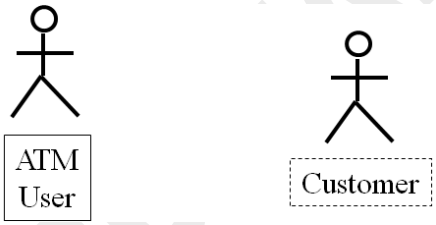
- Every use case must have a name that distinguishes it from other use cases
- A name is a textual string, consisting of any number of letters, numbers, and most punctuation marks
- In practice, use case names are short active verb phrases naming some behavior found in the vocabulary of the system being modeled
- A name alone is known as simple name
- A path name is the use case name prefixed by the name of the package in which that use case lives
- It must be unique within its enclosing package

Use Cases

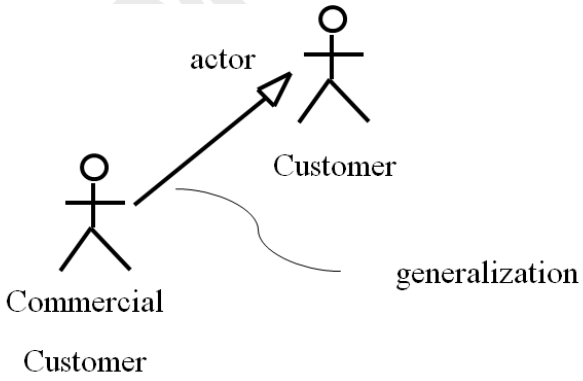


Actors

- An actor represents a coherent set of roles that users of use cases play when interacting with the use cases
- Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system
- Actors can be of general kind
- They can also be specialized using generalization relationship



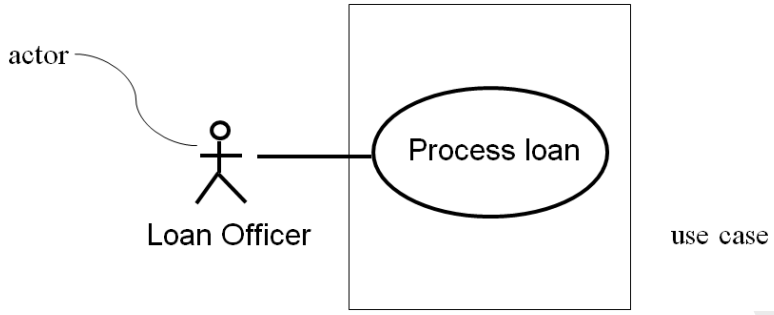
Specialized Actors



Use Case Diagrams

- A use case diagram is the diagram that shows a set of use cases and actors and their relationships
- It has a name and graphical contents that are a projection into a model
- Contents of Use Case Diagrams
  - ✓ Use cases
  - ✓ Actors
  - ✓ Dependency, generalization, and association relationships
- Actors may be connected to use cases only by association

- An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages
- A Use Case Diagram



- Use case diagrams are used to model the use case view of the system being modeled
- This involves modeling the context of a system, subsystem, or class, or modeling requirements of the behavior of these elements

Identifying Use Cases

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update, or delete that information?
- Does the system need to notify an actor about changes in its internal state?
- Are there any external events that the system must know about? What actor informs the system about those events?
- Startup, shutdown, diagnostics, installation, training, changing a business process

Documenting Use Cases

- Includes basic functionality, alternatives, error conditions, preconditions, post-conditions
- Preconditions - the state the system must be in at the start of the use case
- Post-conditions - the state the system must be in at the end of the use case
- Flow of events - a series of declarative statements listing the steps of a use case from the actor's point of view
- Alternatives - allows a different sequence of events than what was used for the basic path

Documenting Use Cases

- Name
- Summary
  - ✓ Short description of use case
- Dependency (on other use cases)
- Actors
- Preconditions
  - ✓ Conditions that are true at start of use case

- Flow of Events
  - ✓ Narrative description of basic path
- Alternatives
  - ✓ Narrative description of alternative paths
- Post-condition
  - ✓ Condition that is true at end of use case

Use Cases and Flow of Events

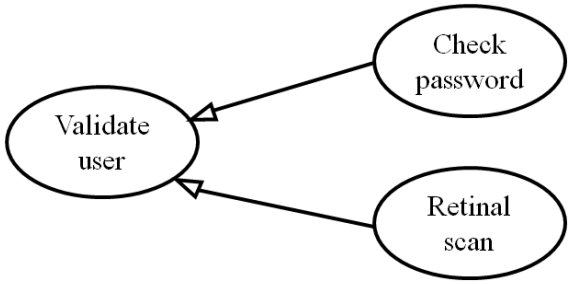
- A use case describes what the system (or subsystem, class, or interface) does but it does not specify how it does it
- It is important that you keep clear the separation of concerns between this outside and inside view
- The behavior of a use case can be specified by describing a flow of events in text clearly enough for an outsider to understand it easily
- How and when the use case starts and ends
- When the use case interacts with the actors and what objects are changed
- The basic flow and alternative flows of behavior
- A use case's flow of events can be specified in a number of ways:
  - ✓ Informal structured text (example to follow)
  - ✓ Formal structured text (with pre- and post-conditions)
  - ✓ Pseudocode

Organizing Use Cases

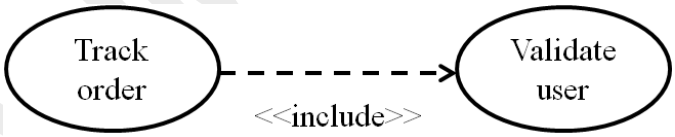
- Use cases can be organized by grouping them in packages
- You can also organize use cases by specifying generalization, include, and extend relationships among them
- You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it)
- Generalization among use cases is just like generalization among classes
- It means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears
- You may have a use case Validate User, which is responsible for the verifying the identity of the user
- This use case can be used in many different application domains

Specialized Use Cases

- You may have two specialized children of this use case (Check password and Retinal scan)

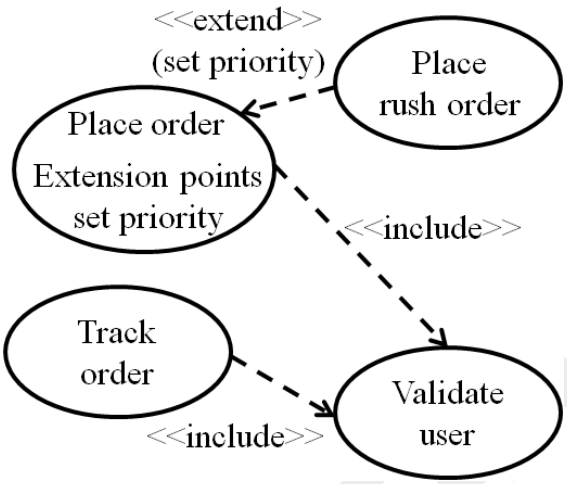


- An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base
- The included use case never stands alone, but is only instantiated as part of some larger base that includes it
- It is like the base use case
- Include relationship is used to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own
- This is an example of dependency
- Track order use case includes Validate user use case
- Including a Use Case
  - ✓ Obtain and verify the order number. include (Validate user). For each part in the order, query its status, then report back to the user



- An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by extending use case
- The base use case may stand alone, but under certain conditions, its behavior may be extended by another use case
- A base use case may be extended only at certain points, called its extension points
- Extend relationship can be used to model
- Optional behavior
- Separate sub-flow that is executed only under given conditions
- Several flows that may be inserted at a certain point, governed by explicit interaction with an actor
- This is also an example of dependency
- Consider the Place order use case, which extends the Place rush order
- Extending a Use Case

- ✓ include (Validate user)
- ✓ Collect the user's order items
- ✓ (set priority)
- ✓ Submit the order for processing

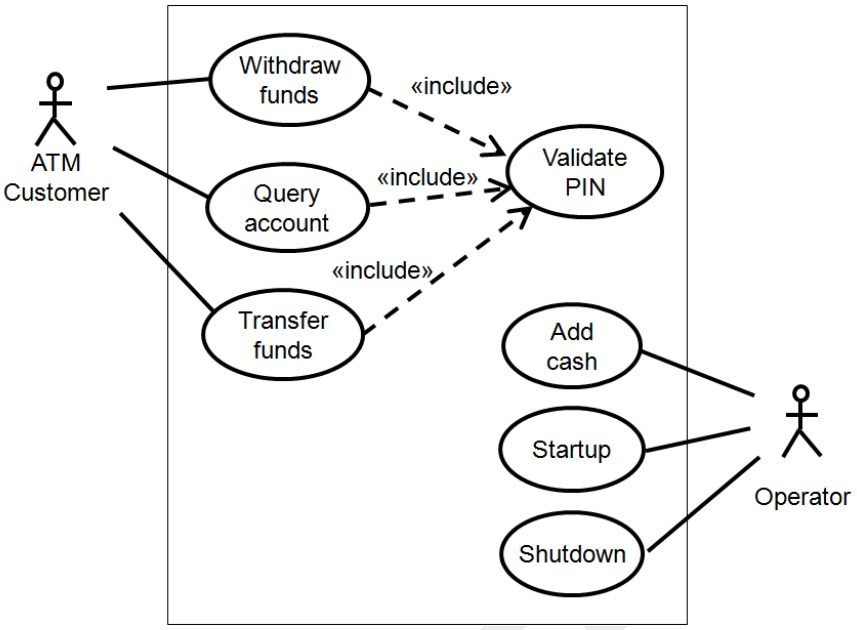


- In this example, set point is an extension point
- A use case may have more than one extension points
- Apply use cases to model the behavior of an element (system, subsystem, class)
- Outside view by domain experts
- Developers can approach the element
- Basis for testing

Guidelines for Use Cases

- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions
- Organize actors by identifying general and more specialized roles
- For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event
- Consider the exceptional ways in which each actor interacts with the element
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior

Use Case Diagram for ATM



Abstract Use Case Example

- Name: Validate PIN
- Summary : System validates customer PIN
- Dependency: none
- Actors: ATM Customer
- Preconditions: ATM is idle, displaying a Welcome message.
- Flow of Events: Basic Path
    1. Customer inserts the ATM card into the Card Reader
    2. If the system recognizes the card, it reads the card number
    3. System prompt customer for PIN number
    4. Customer enters PIN
    5. System checks the expiration date and whether the card is lost or stolen
    6. If card is valid, the system then checks whether the user-entered PIN matches the card PIN maintained by the system
    7. If PIN numbers match, the system checks what accounts are accessible with the ATM card
    8. System displays customer accounts and prompts customer for transaction type: Withdrawal, Query, or Transfer
- Alternatives:
    - ✓ If the system does not recognize the card, the card is ejected
    - ✓ If the system determines that the card date has expired, the card is confiscated
    - ✓ If the system determines that the card has been reported lost or stolen, the card is confiscated

- Alternatives (cont.):
  - ✓ If the customer-entered PIN does not match the PIN number for this card, the system re-prompts for PIN
  - ✓ If the customer enter the incorrect PIN three times, the system confiscates the card
  - ✓ If the customer enters Cancel, the system cancels the transaction and ejects the card
- Postcondition: Customer PIN has been validated

Concrete Use Case Example

- Name: Withdraw Funds
- Summary : Customer withdraws a specific amount of funds from a valid bank account
- Dependency: Include Validate PIN abstract use case
- Actors: ATM Customer
- Preconditions: ATM is idle, displaying a Welcome message.
- Flow of Events: Basic Path
  1. Include Validate PIN abstract use case
  2. Customer selects Withdrawal, enters the amount, and selects the account number
  3. System checks whether customer has enough funds in the account and whether the daily limit will not be exceeded
  4. If all checks are successful, system authorizes dispensing of cash
  5. System dispenses the cash amount
  6. System prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance
  7. System ejects card
  8. System displays Welcome balance
- Alternatives:
  - ✓ If the system determines that the account number is invalid, it displays an error message and ejects the card
  - ✓ If the system determines that there are insufficient funds in the customer's account, it displays an apology and ejects the card
  - ✓ If the system determines that the maximum allowable daily withdrawal amount has been exceeded, it displays an apology and ejects the card
  - ✓ If the ATM is out of funds, the system displays an apology, ejects the card, and shuts down the ATM
- Postcondition: Customer funds have been withdrawn

A Well-Structured Use Case

- Names a single, identifiable, and reasonably atomic behavior of the system or part of the system
- Factors common behavior by pulling such behavior from other use cases
- Factors variants by pushing such behavior into other use cases that extend it
- Describes the flow of events clearly enough for an outsider to easily understand it
- Is described by a minimal set of scenarios that specify the normal and variant semantics of the use case

Banking System Case Study

Problem Description

- A bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card is the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.
- If the PIN is validated satisfactorily, the customer is prompted for a withdrawal, query, or transfer transaction. Before withdrawal transaction can be approved, the system determines that sufficient funds exist in the requested account, that the maximum daily limit will not be exceeded, and that there are sufficient funds available at the local cash dispenser. If the transaction is approved, the requested amount of cash is dispensed, a receipt is printed containing information about the transaction, and the card is ejected. Before a transfer transaction can be approved, the system determines that the customer has at least two accounts and that there are sufficient funds in the account to be debited. For approved query and transfer requests, a receipt is printed and card ejected. A customer may cancel a transaction at any time; the transaction is terminated and the card is ejected. Customer records, account records, and debit card records are all maintained at the server.
- An ATM operator may start up and close down the ATM to replenish the ATM cash dispenser and for routine maintenance. It is assumed that functionality to open and close accounts and to create, update, and delete customer and debit card records is provided by an existing system and is not part of this problem.
- 'Designing Concurrent, Distributed, and Real-Time Applications with UML' by H. Gomaa, Addison-Wesley, 2000

Use Case Model

- The use cases are described in the use case model
- There are two actors of this system
  - ✓ ATM Customer
  - ✓ Operator

ATM Customer

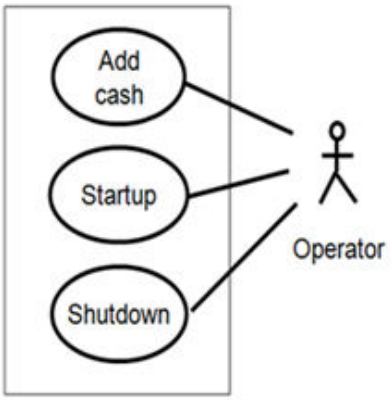
- Withdraws funds from the checking or savings account
- Query the balance of an account
- Transfer funds from one account to another
- The ATM customer interacts with the system via the ATM card reader, keyboard/display, cash dispenser, and receipt printer

ATM Operator

- Shutdowns the ATM
- Replenishes the ATM cash dispenser
- Starts the ATM

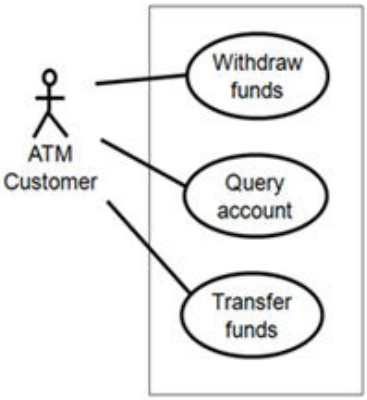
Use Cases for ATM Operator

- Add cash
- Startup
- Shutdown



User Case for ATM Customer

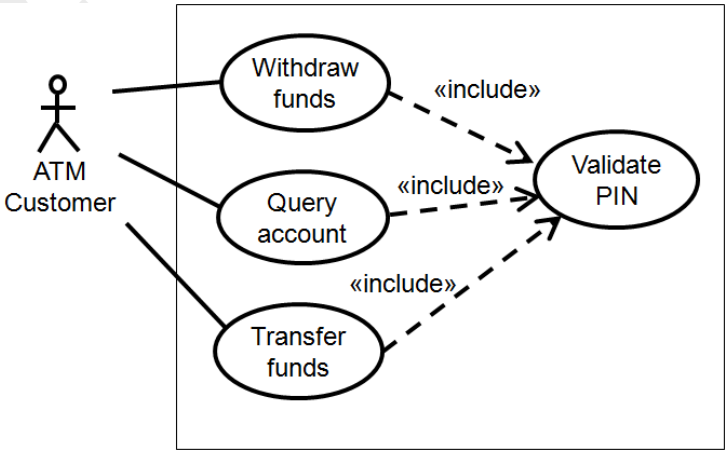
- Withdraw funds
- Query account
- Transfer funds



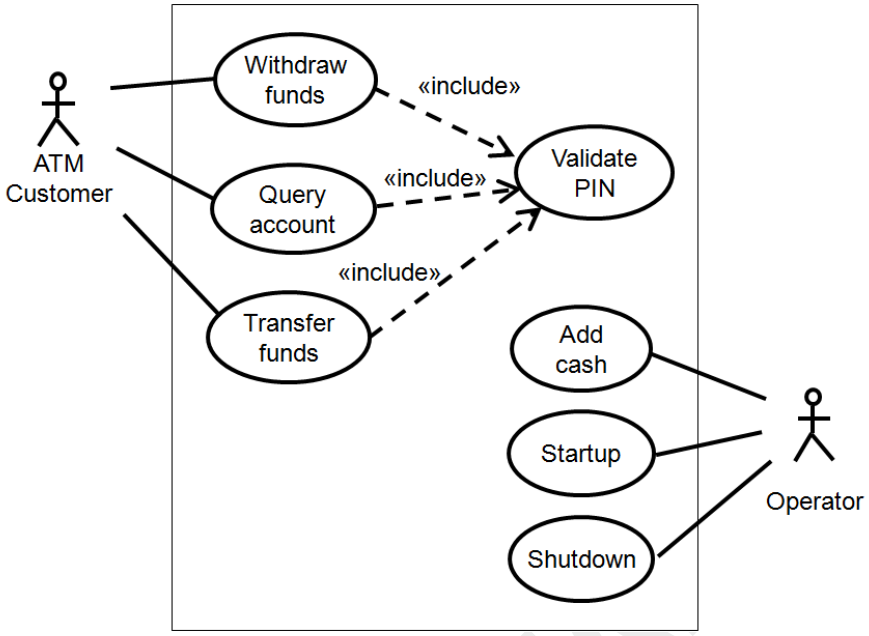
Organizing Use Cases for ATM Customer

- Comparing these three use cases, it can be seen that the first part of each use case – namely, the PIN validation – is common to all three use cases
- This common part of the three use cases is factored out as an abstract inclusion use case called Validate PIN
- The Withdraw funds, Query account, Transfer funds use cases can each be rewritten more concisely as concrete use cases that include the Validate PIN abstract use case

Uses Case Diagram for ATM Customer



Use Case Diagram for ATM



Use Cases of the ATM System

- Abstract use cases
  - ✓ Validate PIN
- Concrete use cases
  - ✓ Withdraw Funds
  - ✓ Query Account
  - ✓ Transfer Funds

Query Account Use Case

- Name: Query Account
- Summary: Customer receives the balance of a valid bank account
- Actor: ATM Customer
- Dependency: Include Validate PIN abstract use case
- Precondition: ATM is idle, displaying a Welcome message
- Flow of Events: Basic Path
  1. Include Validate PIN abstract use case
  2. Customer selects Query, enters account number
  3. System reads account balance
  4. System prints a receipt showing transaction number, transaction type, and account balance
  5. System ejects card
  6. System displays Welcome message
- Alternative:
  - ✓ If the system determines that the account number is invalid, it displays an error message and ejects the card
- Postcondition:
  - ✓ Customer account has been queried

Transfer Funds Use Case

- Name: Transfer Funds
- Summary: Customer transfers funds from one valid bank account to another
- Actor: ATM Customer
- Dependency: Include Validate PIN abstract use case
- Precondition: ATM is idle, displaying a Welcome message
- Flow of Events: Basic Path
  - ✓ 1. Include Validate PIN abstract use case
  - ✓ 2. Customer selects Transfer and enters amounts, from account, and to account
  - ✓ 3. If the system determines the customer has enough funds in the from account, it performs the transfer
  - ✓ 4. System prints a receipt showing transaction number, transaction type, amount transferred, and account balance
  - ✓ 5. System ejects card
  - ✓ 6. System displays Welcome message
- Alternatives:
  - ✓ If the system determines that the from account number is invalid, it displays an error message and ejects the card
  - ✓ If the system determines that the to account number is invalid, it displays an error message and ejects the card
  - ✓ If the system determines that there are insufficient funds in the customer's from account, it displays an apology and ejects the card
- Postcondition:
  - ✓ Customer funds have been transferred

Software Modeling

Need for Modeling

- Modeling is a central part of all activities that lead up to the development good software
- We build models to communicate the desired structure and behavior of our system
- We build models to visualize and control the system's architecture
- We build models to better understand the system we are building, often exposing opportunities for simplification and reuse
- We build models to minimize risk
- We build models so that we can better understand the system we are developing
- We build models of complex systems because we cannot comprehend such a system in its entirety
- A model is a simplification of reality

Four Aims of Modeling

- Models help us to visualize a system as it is or as we want it to be
- Models permit us to specify the structure or behavior of a system
- Models give us a template that guides us in constructing a system
- Models document the decisions we have made

Do We Model Everything?

- Modeling is not just for big systems
- Small pieces of software can also benefit from modeling
- Larger and more complex systems benefit more from modeling

Principles of Modeling

- The choice of what models to create has profound influence on how a problem is attacked and how a solution is shaped
- Every model may be expressed at different levels of precision
- The best models are connected to reality
- No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models

Problem Analysis

- Activity that encompasses learning about the problem to be solved, understanding the needs of potential users, trying to find out who the user really is, and understanding all constraints on the solution
- Defining the product space – the range of all possible software solutions that meets all known constraints

Points to Note

- Understanding the needs of potential users
- Trying to find out who the user really is
- Understanding all constraints on the solution
- All three are very difficult

Need for a Software Solution

- There is recognition that a problem exists and requires a solution
- A new software idea arises
- Either there is no automated system or there is a need for some improvements in the existing automated system

Subjects to Study for Modeling Manual System

- People and/or machines currently playing some role
- Items produced, processed, or consumed by these people and machines
- Functions performed by these people and machines
- Basic modes of operation that determine what functions are performed and when

Subjects to Study for Modeling to Improve a System

- People/machines who have a need for some service to be performed or some item to be produced
- Items that need to be produced to satisfy the need
- Items that are necessary in order to produce the required new service or item
- Functions that need to be performed in order to generate the required new service or item
- Basic modes of operations that determine what functions are performed and when

Steps in Problem Analysis

- Gain agreement on the problem definition
- Understand the root causes – the problem behind the problem
- Identify the stakeholders and the users
- Define the solution system boundary
- Identify the constraints to be imposed on the solution

Principles of Modeling

- Partitioning
  - ✓ Captures aggregation/part of relations among elements in the problem domain
- Abstraction
  - ✓ Captures generalization/specialization relations among elements in the problem domain
- Projection
  - ✓ Captures different views of elements in the problem domain

Avoid the Urge to Design

- One can easily get into the trap of completely designing the proposed system
- The focus of modeling during requirements engineering is understanding and that of preliminary design is optimization

Software Modeling

- A number of modeling techniques have been developed over the years

Features of Modeling Techniques

- Facilitate communication
- Provide a means of defining the system boundary
- Provide a means of defining partitions, abstractions, and projections
- Encourage the analyst to think and document in terms of the problem as opposed to the solution
- Allow for opposing alternatives but alert the analyst to their presence
- Make it easy to modify the knowledge structure

Modeling Techniques

- Object-oriented modeling
  - ✓ Static and dynamic modeling
- Functional modeling
- Dynamic modeling

Object-Oriented Modeling

- The main building block of all software systems is the object or class
- An object is a thing, generally drawn from the vocabulary of the problem or the solution space
- A class is a description of a set of common objects
- OO paradigm helps in software modeling of real-life objects in a direct and explicit fashion
- It also provides a mechanism so that the object can inherit properties from their ancestors, just like real-life objects
- Every object has identity, state, and behavior
- The object-oriented approach has proven itself to be useful, because it has proven to be valuable in building systems in all sorts of problem domains and encompassing all degrees of size and complexity
- Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+
- A number of consequences flow from the choice of viewing the world in an OO fashion
  - ✓ What is the structure of good OO architecture?
  - ✓ What artifacts the project should create?
  - ✓ Who should create them?
  - ✓ How should they be measured?

Object-Oriented Modeling Methods

- Shlaer/Mellor - 1988
- Coad/Yourdon – 1991
- Booch - 1991
- OMT by Rumbaugh et. al. – 1991
- Wirfs-Brock – 1991
- And many more

Unification Effort

- 1994 key researchers Grady Booch, James Rambaugh, and Ivar Jacobson joined hands to unify their respective methodologies
- In 1997 Object Management Group (OMG) finally approved the Unified Modeling Language (UML)

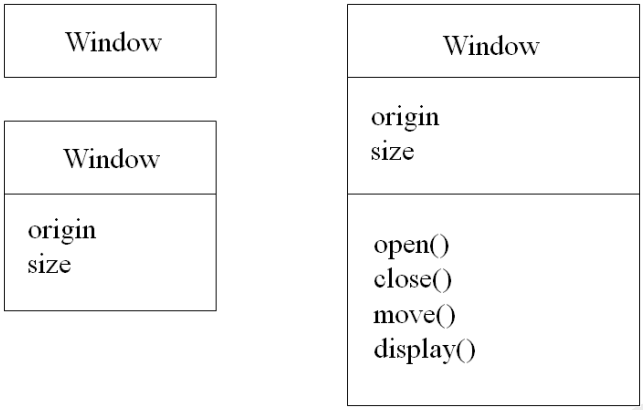
Unified Modeling Language

- Visualizing, specifying, constructing, and documenting object-oriented systems is exactly the purpose of the unified modeling language or UML
- The rules of UML focus on the conceptual and physical representation of a system
- Process independent
- Notation has well-defined semantics
- It has become the de-facto standard for modeling
- Many vendors provide tools that support different modeling views
- UML provides a very rich set of concept areas
    - ✓ Static structure
    - ✓ Dynamic behavior
    - ✓ Implementation constructs
    - ✓ Model organization
    - ✓ Extensibility mechanisms

Static Structure

- Any precise model must first define the universe of discourse, that is, the key concepts from the application, their internal properties, and their relationships to each other
- This set of constructs is the static view
- The application concepts are modeled as classes, each of which describes a set of discrete objects that hold information and communicate to implement behavior
- The information they hold is modeled as attributes; the behavior they perform is modeled as operations

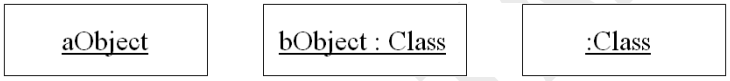
UML Notation for Classes



Objects and Classes

- An object is an instantiation of a class
- It has an identity, state, and behavior

UML Notation for Objects

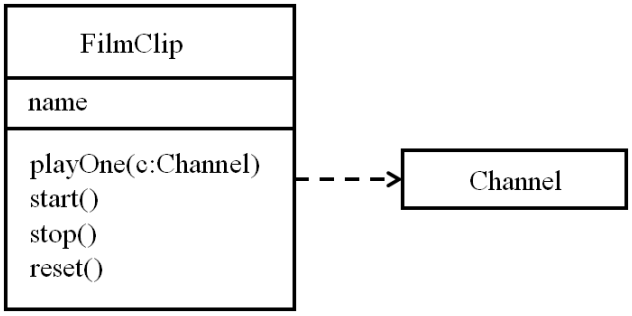


Relationships between Objects

- A relationship is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships
- Dependencies, generalizations, and associations are all static things defined at the level of classes
- In the UML, these relationships are usually visualized in class diagrams
- These relationships convey the most important semantics in an object-oriented model

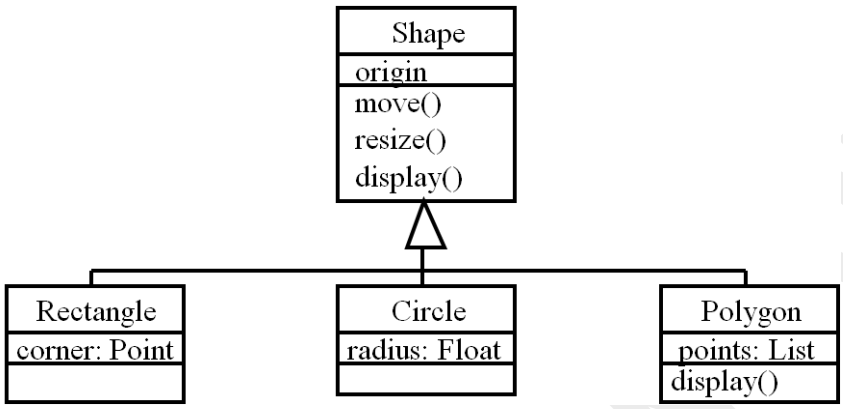
Dependency Relationship

- A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse
- Graphically, a dependency is rendered as a dashed line, directed to the thing being depended on
- Use dependencies when you want to show one thing using another thing



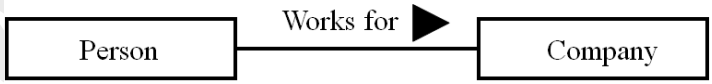
Generalization Relationship

- A generalization is a relationship between a general thing (called a super class or parent) and a more specific kind of that thing (called the subclass or child)
- Generalization is sometimes called an 'is-a-kind-of' relationship

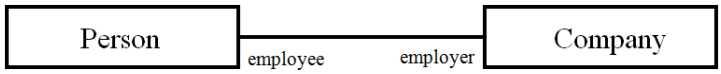


Association Relationship

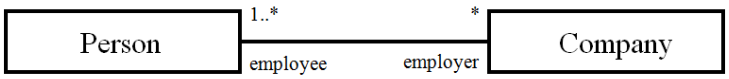
- An association is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa
- Graphically, an association is rendered as a solid line connecting the same of different classes
- Use associations when you want to show structural relationships
- An association can have four adornments
  - ✓ Name
  - ✓ Role
  - ✓ Aggregation
    - Captures the 'whole-part' relationship
    - Composition – a stronger 'whole-part' relationship
  - ✓ Multiplicity
- Association Relationship:



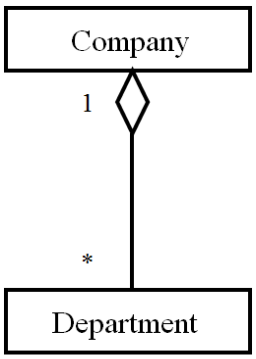
Association Names



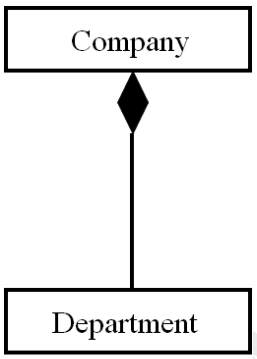
Roles



Multiplicity



Aggregation



Composition

Hints and Tips

- Use dependencies only when the relationships you are modeling are not structural
- Use generalization only when you have 'is-a-kind-of' relationship; multiple inheritance can often be replaced with aggregation
- Keep your generalization relationships generally balanced; inheritance latices should not be too deep (more than five levels or so should be questioned) nor too wide (instead, look for the possibility of intermediate abstract classes)
- Beware of introducing cyclical generalization relationships
- Use associations primarily where there are structural relationships among objects

Class Inheritance and Object Composition

- Class Inheritance: Advantages
  - ✓ Class inheritance is defined statically at compile-time
  - ✓ Class inheritance is straightforward to use, as it is supported directly by object-oriented programming languages
  - ✓ Class inheritance makes it easy to modify the implementation being reused
- Class Inheritance: Disadvantages
  - ✓ You cannot change the implementations inherited from parent class at run-time, because inheritance is defined at compile-time
  - ✓ Parent classes often define at least part of their subclasses' physical representation. Any change in the parent's implementation will force the subclass to change
  - ✓ Inheritance breaks encapsulation
  - ✓ Implementation dependencies can cause problems when you're trying to reuse a subclass
- Object Composition: Advantages
  - ✓ Object composition is defined dynamically at run-time through objects acquiring references to other objects
  - ✓ Composition requires objects to respect each others' interfaces, that requires you to carefully define interfaces of classes
  - ✓ Encapsulation is not broken

- ✓ Very few implementation dependencies
- ✓ Object composition has a positive affect on the system design
    - Classes are encapsulated and focused on one task
    - Classes and class hierarchies will remain small
    - There will be more objects than classes, and the system's behavior will depend on their interrelationships instead of being defined in one class

Object-Oriented Static Modeling of the Banking System

Steps in Object-Oriented Analysis

- Identify classes within the problem domain
- Define the attributes and methods of these classes
- Define the behavior of those classes
- Model the relationship between those classes

Identification of Objects and Classes

- Examine structures in the real world
- Identify other systems with which the current or proposed system will interact
- Identify those things in the real world that need to be remembered for later retrieval
- Identify specific roles played by individuals
- Identify physical locations that need to be known
- Identify organizations that humans belong to
- Identify catalogs that have to record quantities of repetitive, static information about things

Structure

- Classification
- Assembly

External Device I/O Objects

- A concrete entity in the application domain is an entity that exists in the real world and has some physical attributes
- For every concrete entity in the real world that is relevant to the application domain, there should be a corresponding software object in the system
- Each software object hides the details of the interface to the real world entity that it receives input from or provides outputs to. However, a software object models the events experienced by the concrete entity to which it corresponds. The events experienced by the entity are inputs to the system, particularly to the software object that models the entity
- Examples
    - ✓ Engine sensor
    - ✓ Brake sensor
    - ✓ Buttons

User Role Objects

- A user role object models a role played in the application domain, typically by a user. A role is a sequence of related actions performed sequentially by a user
- If a user can play two or more independent roles, then this may be represented by a different object for each role
- Examples: Machine Operator, Loan Officer

Control Objects

- A control object is an active abstract object in the problem domain that has different states and controls the behavior of other objects and functions. A control object is defined by means of a finite state machine, which is represented by a state transition diagram
- A control object receives incoming events that cause state transitions
- It generates output events that control other objects or functions
- In a real-time system, there are usually one or more control objects
- Examples: Elevator control

Data Abstraction Objects

- For every entity in the application domain that needs to be remembered, there should be a corresponding data abstraction object. These objects model the real world entities by encapsulating the data that needs to be remembered as well as supporting the operations on that data
- Locations and organizations are examples of objects that need to be remembered
- A data abstraction object is a passive object
- The basis for a data abstraction object is a data store

Algorithm Objects

- An algorithm object encapsulates an algorithm used in the problem domain
- This kind of object is more prevalent in real-time domains
- Example: Scheduler object

Banking System Case Study

Problem Description

- A bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained

by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.

- If the PIN is validated satisfactorily, the customer is prompted for a withdrawal, query, or transfer transaction. Before withdrawal transaction can be approved, the system determines that sufficient funds exist in the requested account, that the maximum daily limit will not be exceeded, and that there are sufficient funds available at the local cash dispenser. If the transaction is approved, the requested amount of cash is dispensed, a receipt is printed containing information about the transaction, and the card is ejected. Before a transfer transaction can be approved, the system determines that the customer has at least two accounts and that there are sufficient funds in the account to be debited. For approved query and transfer requests, a receipt is printed and card ejected. A customer may cancel a transaction at any time; the transaction is terminated and the card is ejected. Customer records, account records, and debit card records are all maintained at the server.
- An ATM operator may start up and close down the ATM to replenish the ATM cash dispenser and for routine maintenance. It is assumed that functionality to open and close accounts and to create, update, and delete customer and debit card records is provided by an existing system and is not part of this problem.
- 'Designing Concurrent, Distributed, and Real-Time Applications with UML' by H. Gomaa, Addison-Wesley, 2000

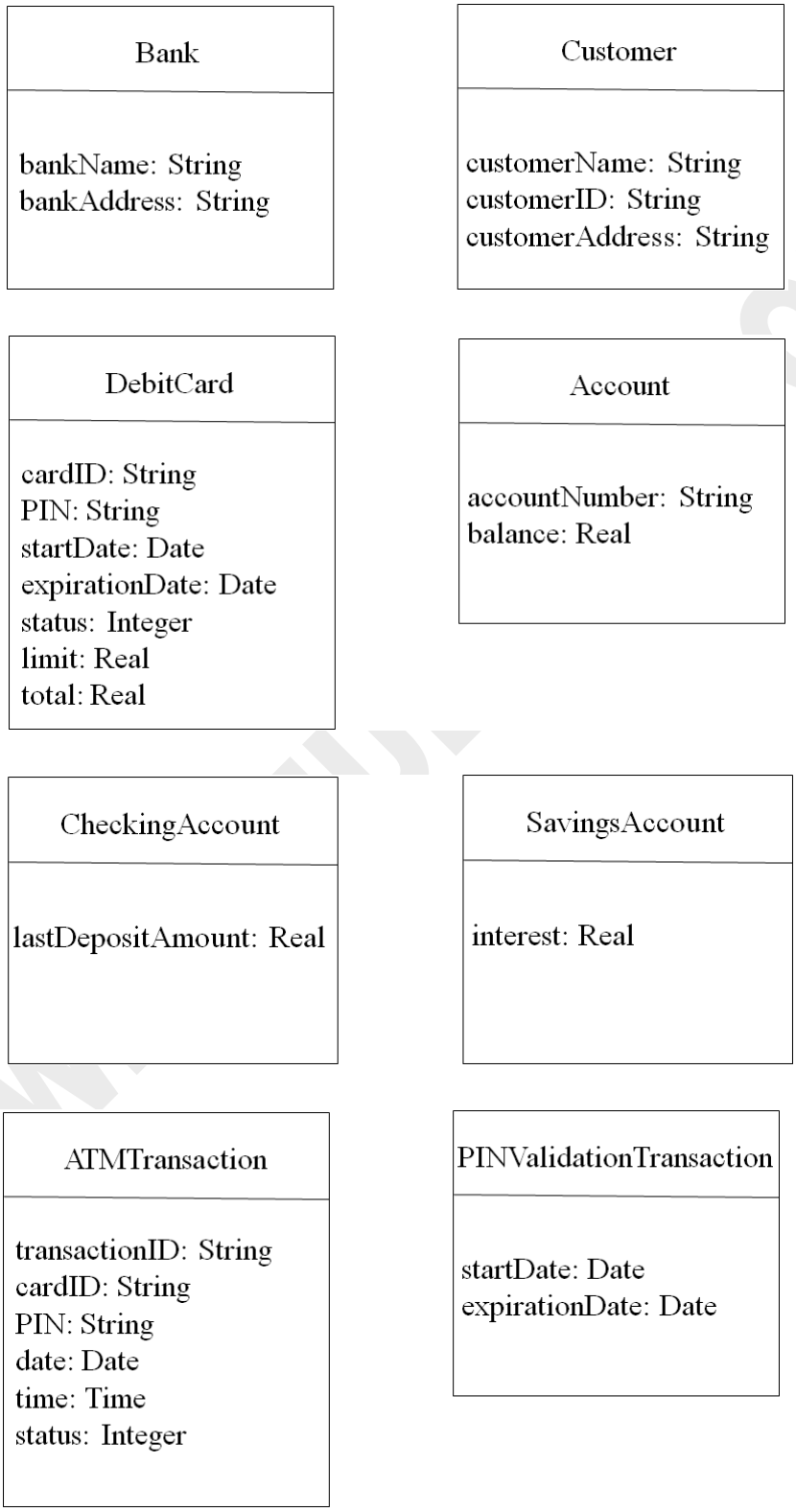
Observations

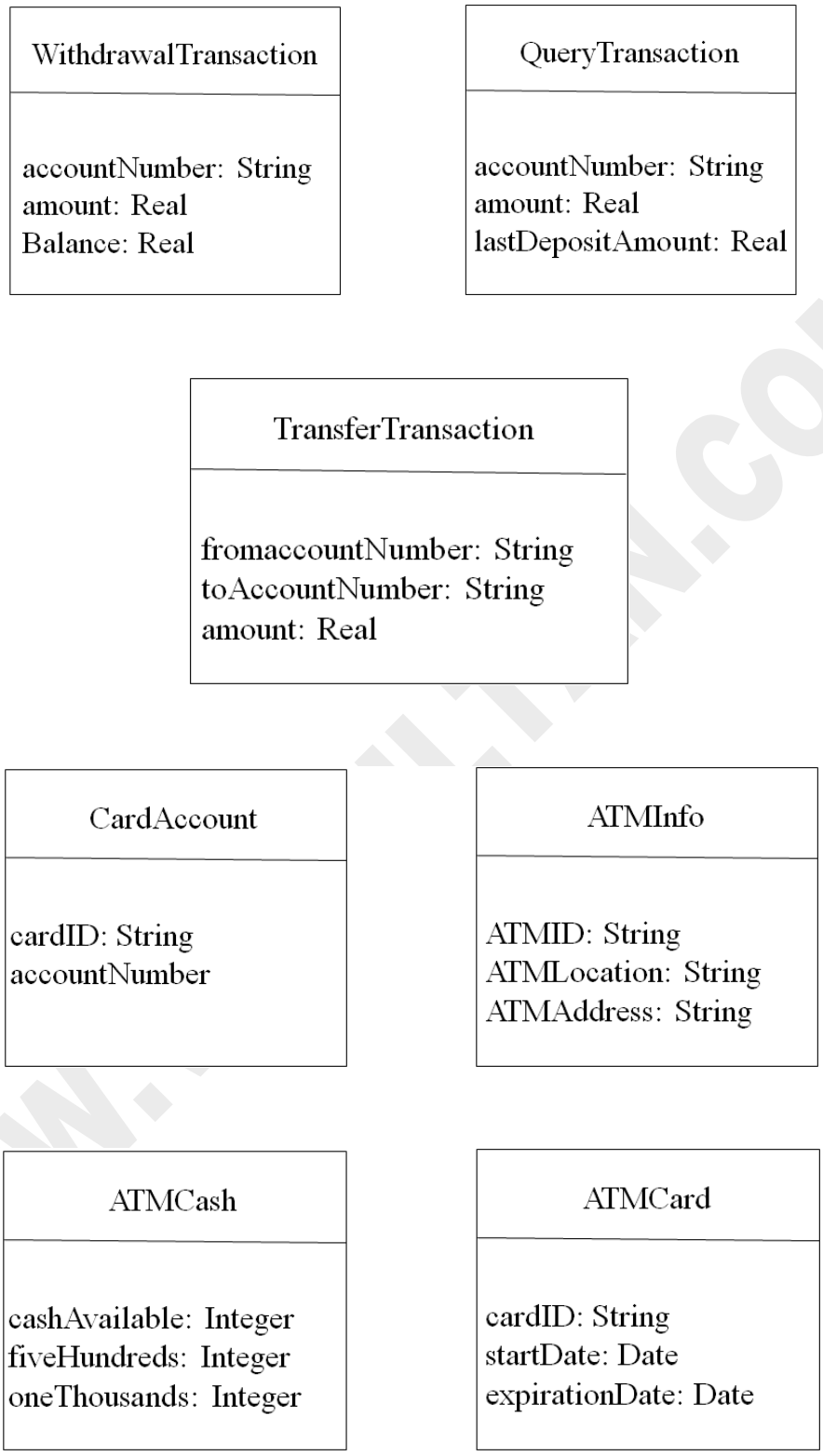
- A bank has several ATMs
- Each ATM has a card reader, a cash dispenser, a receipt printer, and a user who interacts with the ATM through a keyboard/display unit
- The card reader reads an ATM card - physical thing
- Dispensed cash and receipt are also physical entities
- ATM operator maintains an ATM

Possible Objects in the ATM Domain

- External device I/O objects
  - ✓ Card Reader
  - ✓ Cash Dispenser
  - ✓ Keyboard/Display
  - ✓ Receipt Printer
- External user
  - ✓ ATM Operator
- User role/entity objects
  - ✓ ATM Customer
  - ✓ ATM (ATM Info)
  - ✓ Debit Card
  - ✓ ATM Card
  - ✓ Bank Account
  - ✓ ATM Cash
  - ✓ ATM Transaction
- User role/entity objects (contd.)
  - ✓ PIN Validation Transaction
  - ✓ Withdrawal Transaction
  - ✓ Query Transaction
  - ✓ Transfer Transaction
  - ✓ Checking Account
  - ✓ Savings Account

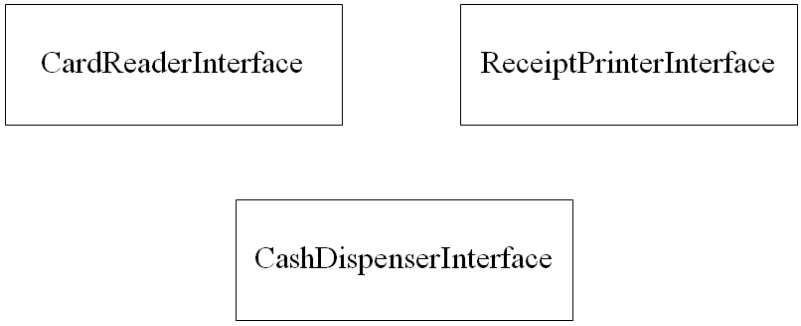
Entity Classes in Banking System



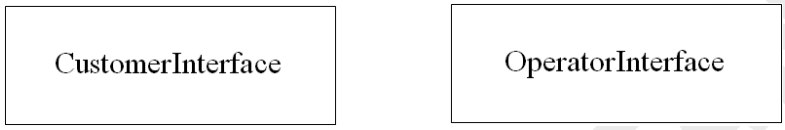

WithdrawalTransaction
accountNumber: String
amount: Real
Balance: Real
QueryTransaction
accountNumber: String
amount: Real
lastDepositAmount: Real
TransferTransaction
fromaccountNumber: String
toAccountNumber: String
amount: Real
CardAccount
cardID: String
accountNumber
ATMInfo
ATMID: String
ATMLocation: String
ATMAddress: String
ATMCash
cashAvailable: Integer
fiveHundreds: Integer
oneThousands: Integer
ATMCard
cardID: String
startDate: Date
expirationDate: Date

Interface Classes for External Objects

- Output Device Interface Classes in the Banking System

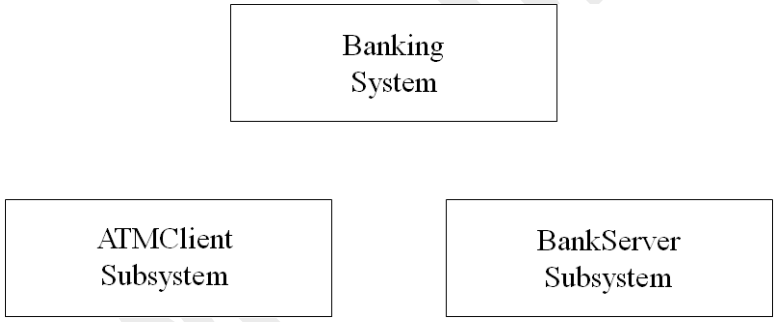


- User Interface Classes in the Banking System



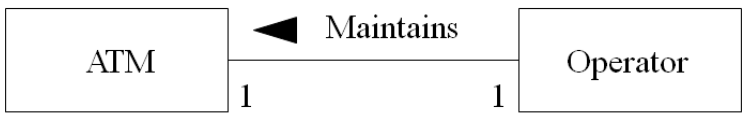
System and Subsystem Classes

- System and Subsystem Classes in the Banking System

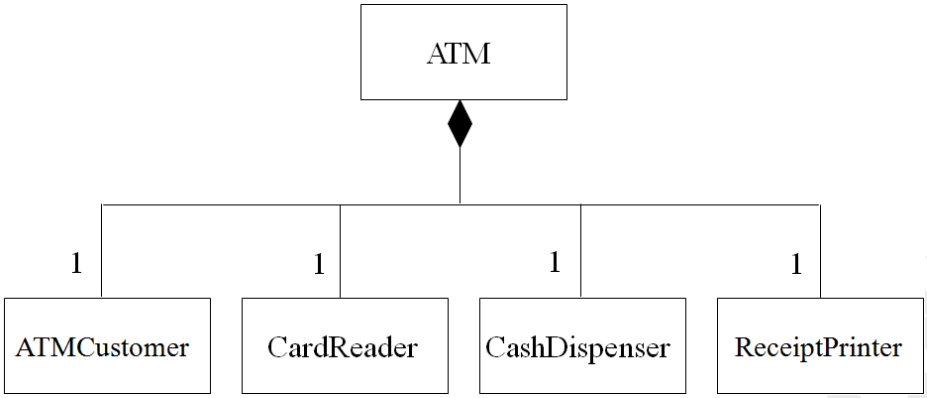


- A Bank Has Many ATMs

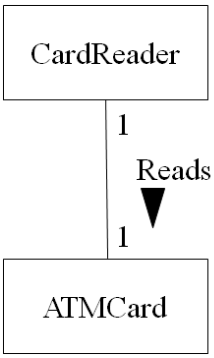
- An Operator Maintains an ATM



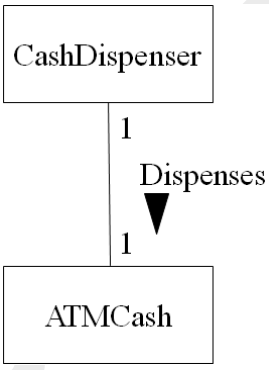
An ATM Has Other Objects



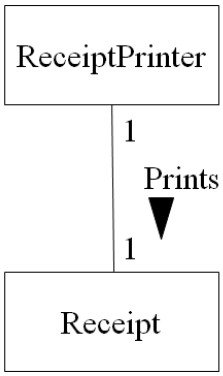
Relation Between CardReader and ATMCard



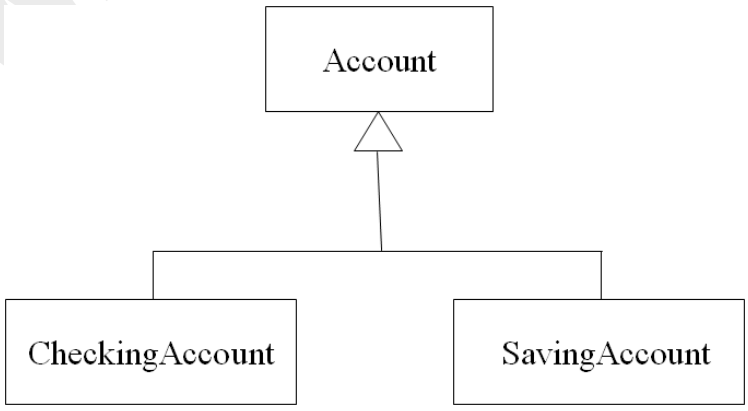
Relation Between CashDispenser and ATMCash



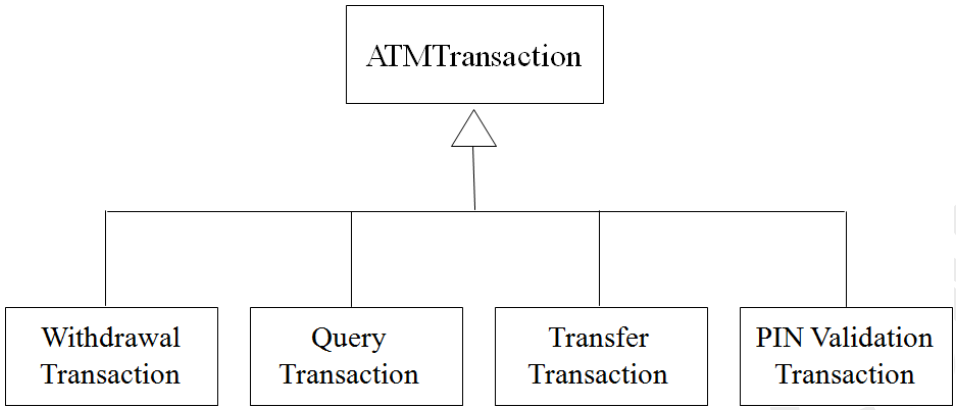
Relation Between ReceiptPrinter and Receipt



Relationship Between Account and CheckingAccount & SavingsAccount

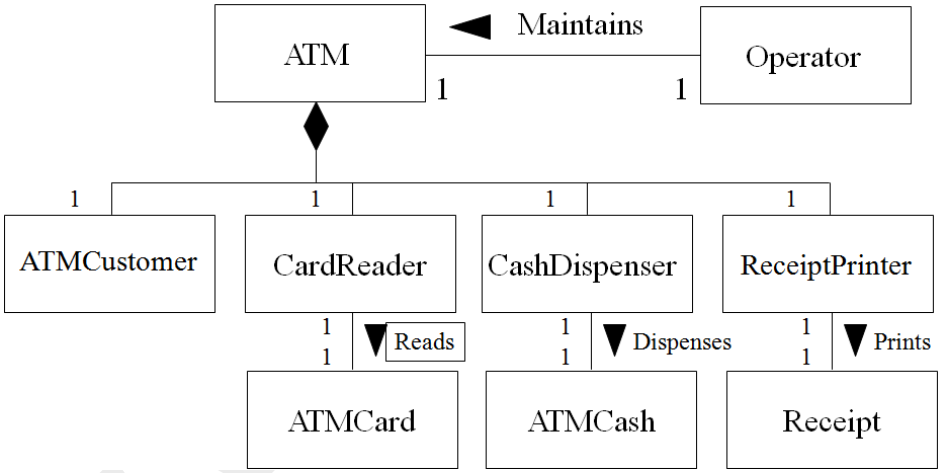


ATMTransaction and its Subclasses



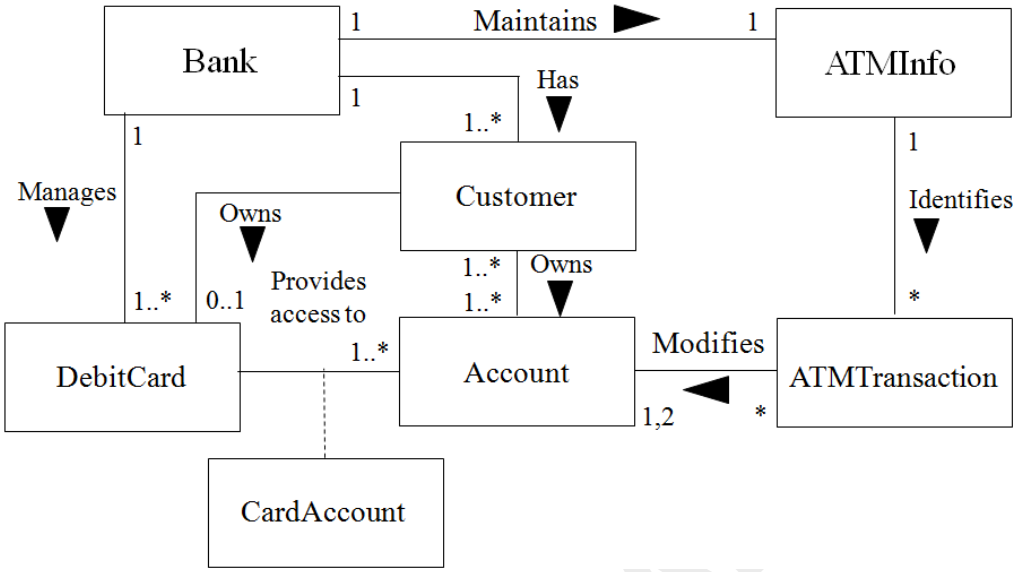
Putting the Classes Together

Conceptual Static Model for Problem Domain: Physical Classes



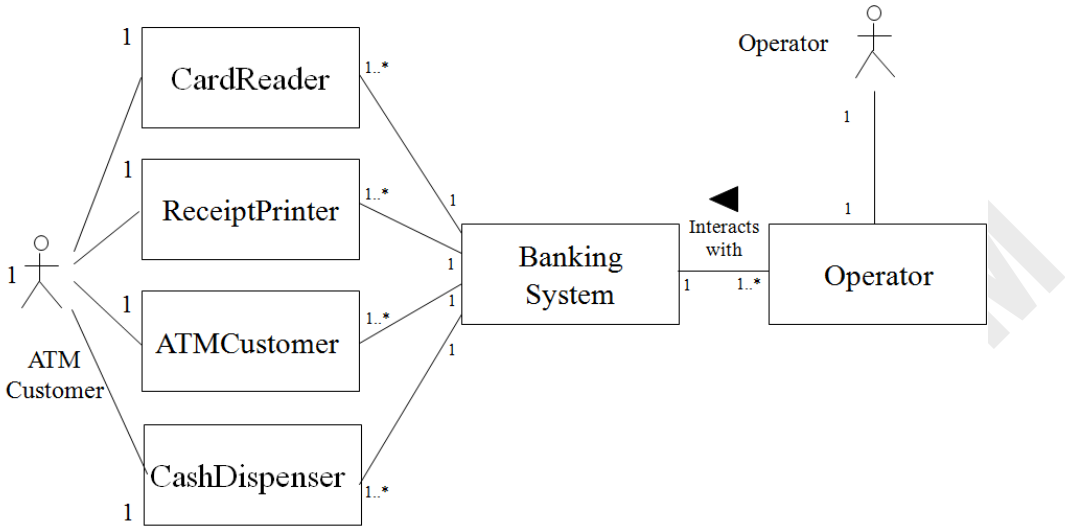
- A bank has several ATMs.
- Each ATM is a composite class consisting of a Card Reader, a Cash Dispenser, a Receipt Printer, and a user, the ATM Customer, who interacts with the system by using a keyboard/display
- The Card Reader reads an ATM Card, which is a plastic card and hence a physical entity
- The Cash Dispenser dispenses ATM Cash, which is also a physical entity in terms of paper money of given denominations
- The Receipt Printer prints a Receipt, which is a paper physical entity
- The physical entities represent classes in the problem domain and are usually modeled by software entity classes
- In addition, the Operator is also a an external user, whose job is to maintain the ATM – who interacts with the system via a keyboard/displaytemuuteat

Conceptual Static Model for Problem Domain: Entity Classes



- The Bank entity class has a one-to-many relationship with the Customer class and the Debit Card class
- The Bank class has only one instance
- The Customer has many to many relationship with Account class, which has its subclasses (Checking Account, Savings Account)
- An Account is modified by an ATM Transaction, which is also specialized to depict different types of transactions (Withdrawal Transaction, Query Transaction, Transfer Transaction, or PIN Validation Transaction)
- There is also a Card Account association class. Association classes are needed in cases where the attributes are of the association, rather than of the classes connected by the association. Thus, in many-to-many association between Debit Card and Account, the individual accounts that can be accessed by a given debit card are attributes of the Card Account class and not of either Debit Card or Account
- Entity classes are also required to model the physical classes in the problem domain
- These include ATM Card, representing the information read off the magnetic strip on the plastic card
- ATM Cash holds the amount of cash maintained at an ATM, in five-hundred and one-thousand rupee notes
- The Receipt holds information about a transaction, and because is holds similar information to the Transaction class described earlier, a separate entity class is unnecessary

Banking System Context Class Diagram

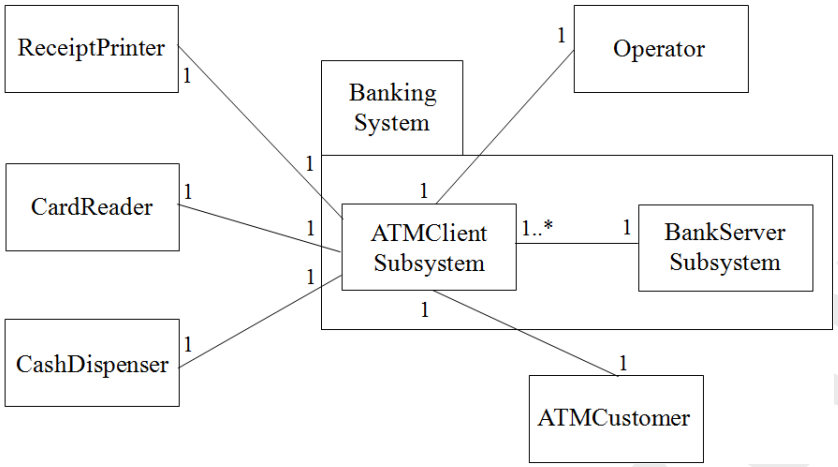


- The system context diagram is developed to show the external classes to which the Banking system has to interface
- We develop the context class diagram by considering the physical classes
- The external classes correspond to the users and I/O devices depicted to the classes in the application domain
- They are the Card Reader, the Cash Dispenser, the Receipt Printer, the ATM Customer who interacts with the system via a keyboard/display, and the Operator who also interacts with the system via a keyboard/display
- There is one instance of each of these external classes for each ATM
- The system context class diagram for the Banking system depicts the system as one aggregate class that receives input from and provides output to the external classes

Client/Server Subsystem Structuring

- The Banking system is a client/server application, some of the objects reside at the ATM client and some reside at the bank server
- The ATM Client Subsystem
- The Bank Server Subsystem

Banking System: Major Subsystems

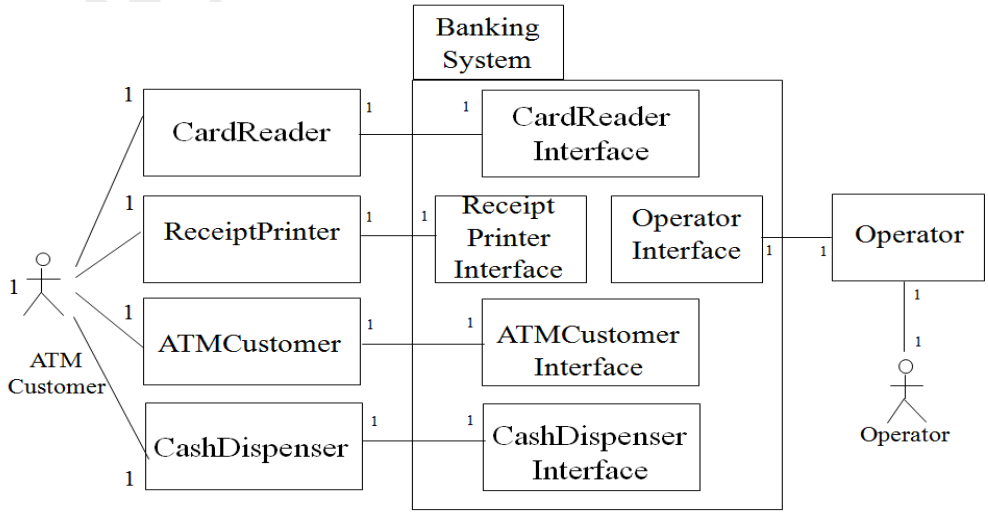


- The ATM Client Subsystem is a composite class, and one instance of this class is located at each ATM machine
- All the external classes interface to this aggregate class
- The Bank Server Subsystem has one instance
- This aggregate class has a one-to-many association with the ATM Client Subsystem
- This is an example of geographically distributed system

ATM Client Object Structuring

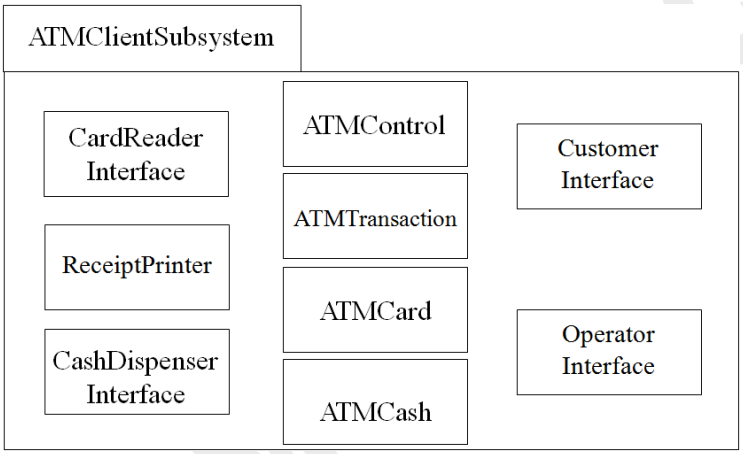
- Consider the interface classes for the ATM Client Subsystem. These classes are determined from the system context diagram
- We design one interface class for each external class, which can be a device interface class or user interface class

Banking System External Classes and Interfaces Classes



- The device interface classes are the Card Reader Interface, through which ATM cards are read, the Cash Dispenser Interface, which dispenses cash, and the Receipt Printer Interface, which prints receipts
- There is also the Customer Interface, which is the user interface class that interacts with the customer via the keyboard/display, displaying textual messages, prompting the customer, and receiving the customer's inputs
- The Operator Interface class provides the user interface to the ATM operator, who replenishes the ATM machine with cash
- There is one instance of each of these interface classes for each ATM

ATM Client Subsystem Classes



- To control the sequence in which actions take place, we identify the need for a control object/class
- This is a state-dependent object

ATM Client Structuring: Objects in Use Cases

- Validate PIN Abstract Use Case
  - ✓ Card Reader Interface
  - ✓ ATM Card
  - ✓ Customer Interface
  - ✓ ATM Transaction (PIN Validation Transaction)
  - ✓ ATM Control (Controls the Sequence)
- Withdraw Funds Concrete Use Case
  - ✓ ATM Transaction (Withdrawal Transaction)
  - ✓ Cash Dispenser Interface
  - ✓ ATM Cash
  - ✓ Receipt Printer Interface
  - ✓ ATM Control controls the sequence

Use Cases and Classes

- Query Account and Transfer Funds use cases are associated with the same set of classes/objects

Operator-Specific Use Cases

- Operator Interface
- ATM Control

Object Structuring in Bank Server Subsystem

- Several entity objects are bank-wide and need to be stored to be accessible from any ATM. Therefore, these objects must be stored at the server
- These objects include: Account objects and Debit Card objects
- In the Bank Server Subsystem, the entity classes are Customer, the Account super-class, Checking Account and Savings Account subclasses, and Debit Card
- There is also the ATM Transaction object, which migrates from the client to the server. The client sends the transaction request to the server, which sends a response to the client. The transaction is also stored at the server as an entity in the form of a Transaction Log, so that a transaction history is maintained
- The transient data sent as part of the ATM Transaction message might differ from the persistent transaction data; for example transaction status is known at the end of the transaction but not during it
- Business logic objects are also needed at the server to define the business-specific application logic for processing client requests. Each ATM transaction type needs a transaction manager to specify the business rules for handling the transaction

Function-oriented Modeling

- In function-oriented modeling, a hierarchy of functions (also known as processes, transforms, transactions, bubbles, and activities) is created
- At the root of the hierarchy is the most abstract function, while the leaf nodes of the hierarchy are least abstract
- Function-oriented modeling is based on the concept of functions or processes, so they become the most important element in this approach
- The functional model describes computations within a system, i.e., what happens
- What is a function or a transform or process?
- Each function is a sequential activity that potentially may execute concurrently with other functions
- The functional model specifies the result of a computation without specifying how or when they are computed

Types of Functions

- Asynchronous Function
  - ✓ An asynchronous function can be activated by another object or function to perform some action
- Asynchronous State Dependent Function
  - ✓ An asynchronous state-dependent function is usually a "one-shot" action, which is executed during a transition from one state to another state
  - ✓ This function is activated by a control transformation
- Periodic Function
  - ✓ A periodic function is activated at regular intervals to perform some action
  - ✓ The frequency with which a specific function is activated is application dependent
- Periodic State-Dependent Function
  - ✓ A periodic function is activated at regular intervals to perform some action
  - ✓ The frequency with which a specific function is activated is application dependent
  - ✓ This function is activated by a control transformation

Functional Modeling

- Non-interactive programs, such as compilers, usually are used for computations. The functional model is the main model for such programs
- On the other hand, databases often have a trivial functional model, since their purpose is to store and organize data, not to transform it
- Fundamental to most of techniques used for functional modeling, is some combination of data flow diagrams and data dictionaries

Data Flow Diagrams

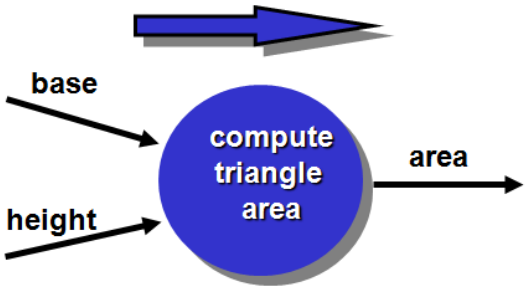
- Data flow diagrams are composed of data on the move, transformations of data into other data, sources and destinations of data, and data in static storage
- Data flow diagrams show the flow of data through a system

Observations About DFDs

- All names should be unique
- A DFD is not a flow chart
- Suppress logical decisions
- Do not become bogged down in details

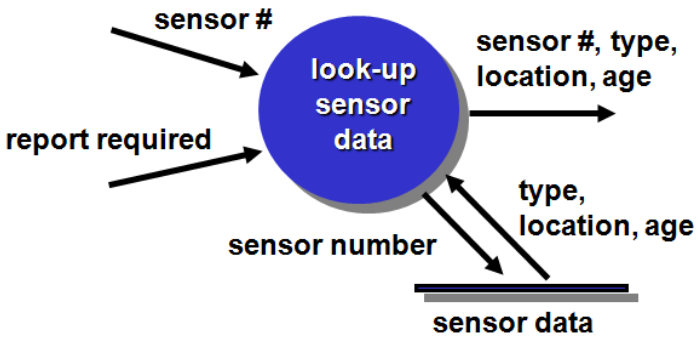
Data Flow

- Data flows through a system, beginning as input and be transformed into output.



Data Stores

- Data is often stored for later use.

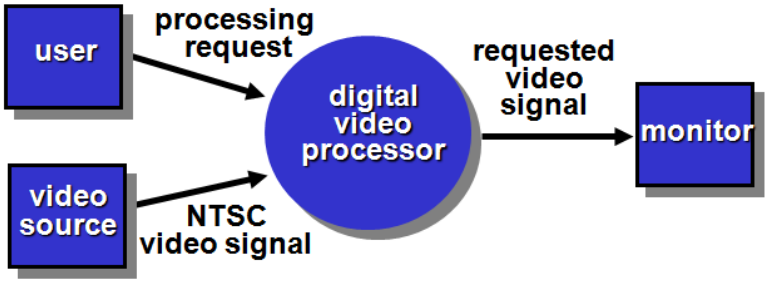


Data Flow Diagramming: Guidelines

- All icons must be labeled with meaningful names
- The DFD evolves through a number of levels of detail
- Always begin with a context level diagram (also called level 0)
- Always show external entities at the context-level or level zero 0
- Always label data flow arrows
- Do not represent procedural logic

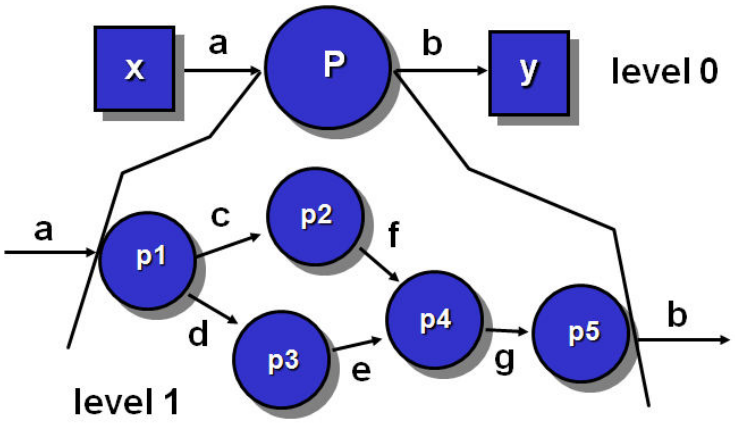
Constructing a DFD

- Determine data objects and associate operations
- Determine external entities
  - ✓ Producers of data
  - ✓ Consumers of data
- Create a level 0 DFD
- Level 0 DFD Example



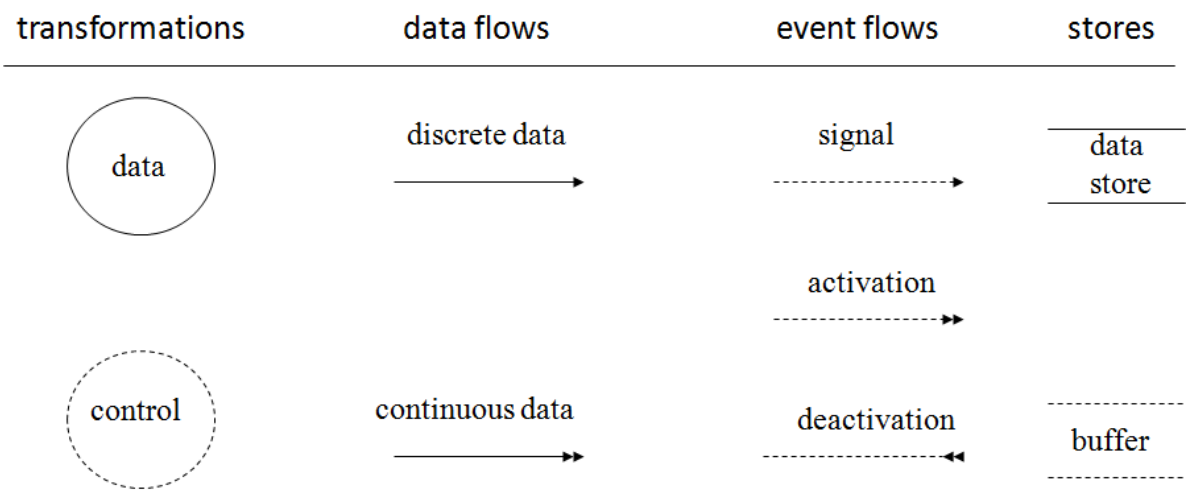
- Write a narrative describing the transform
- Parse to determine next level transforms
- "Balance" the flow to maintain data flow continuity
- Develop a level 1 DFD
- Use a 1:5 (approx.) expansion ratio

The Data Flow Hierarchy



Ward Notations for DFD Extensions

- Ref: Software Requirements by Alan Davis, PH 1993. Copyright 1986 IEEE



Description of Ward's Extensions

- Discrete data
  - ✓ A single item of data
- Continuous data
  - ✓ A source of constantly available and perhaps continuously changing data

- Signal
  - ✓ Reporting an event
- Activation
  - ✓ A direct overt action to initiate another process
- Deactivation
  - ✓ A direct overt action to stop another process
- Data transformation
  - ✓ This is an example of a process, which is used to transform data values
  - ✓ The lowest-level functions are pure functions without side effects
  - ✓ A process may have side effects if it contains nonfunctional components, such as data stores or external objects
- Control transformation
  - ✓ Control transformations are modeled in the dynamic model, as they deal with sequencing and control issues

Data Dictionaries

- Data dictionaries are simply repositories in which to store information about all data items defined in DFDs
- Contents of Data Dictionaries
  - ✓ Name of the data item
  - ✓ Aliases
  - ✓ Description/purpose
  - ✓ Related data items
  - ✓ Range of values
  - ✓ Data flows
  - ✓ Data structure definition/for

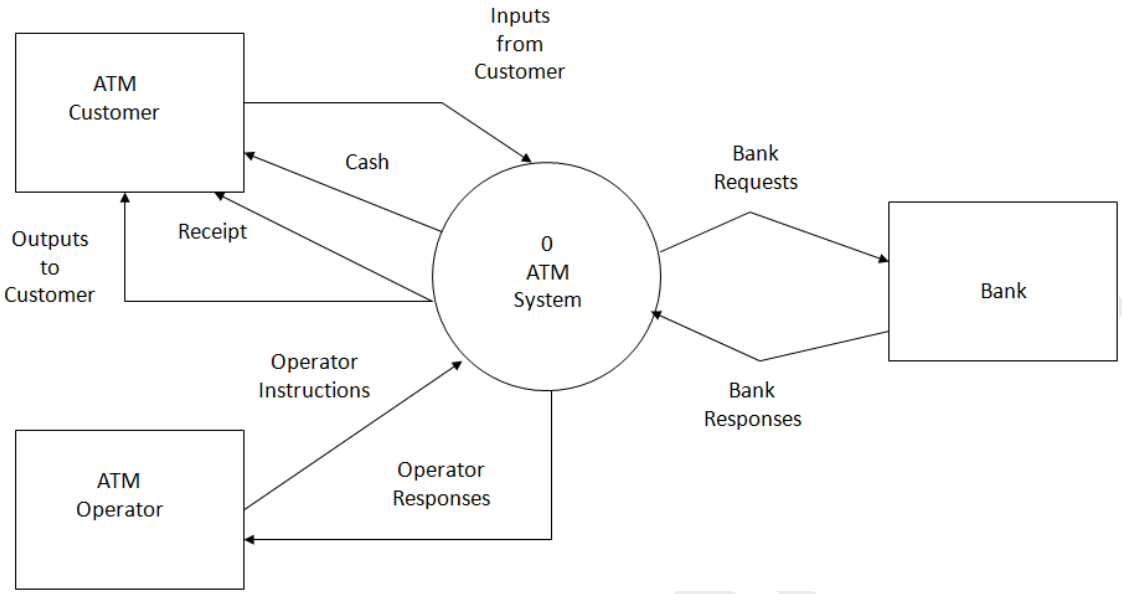
Function-oriented Modeling Techniques

- Structured requirements definition
- Structured analysis and system specification
- Modern structured analysis
- Real-time structured analysis and structured design
- Structured analysis and design technique
- PSL/PSA

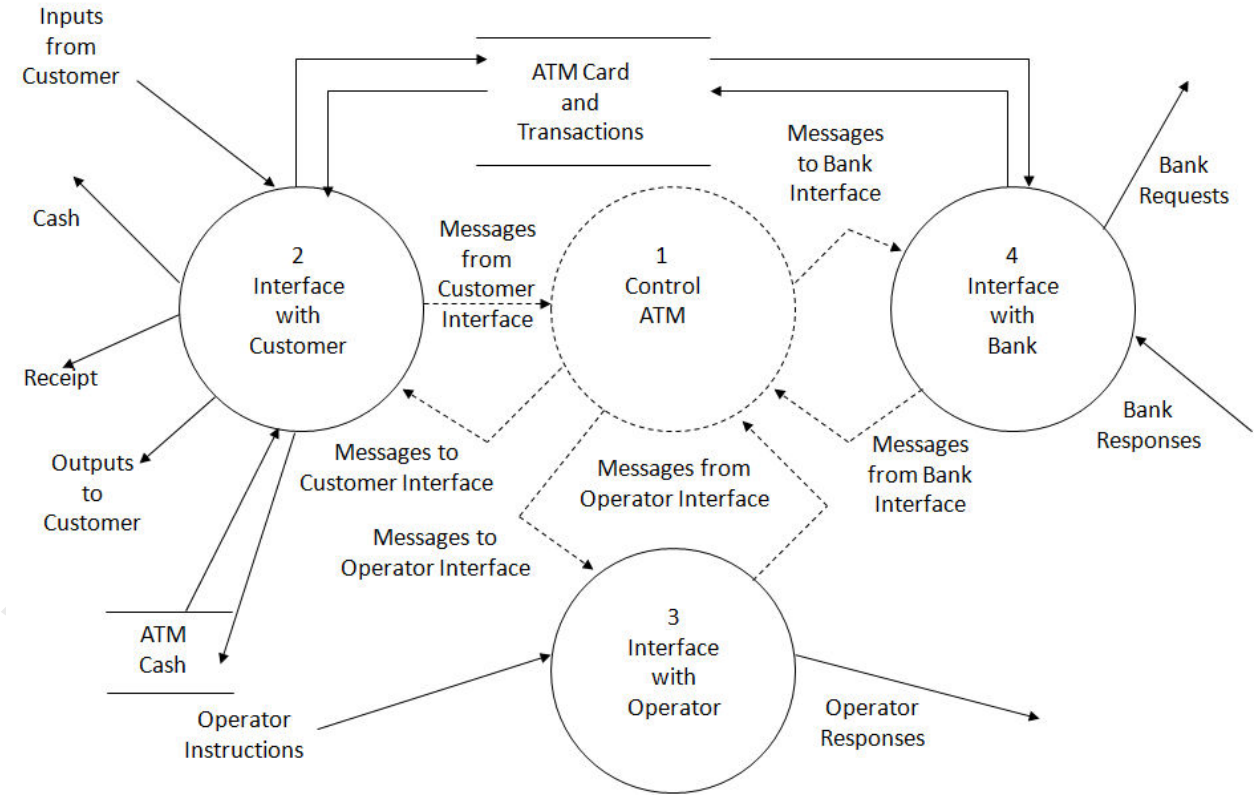
Real-Time Structured Analysis and Structured Design (RSTAD)

- Develop the system context diagram
- Perform data flow/control flow decomposition
- Develop control transformations or control specifications
- Define mini-specifications (process specifications)
- Develop data dictionary

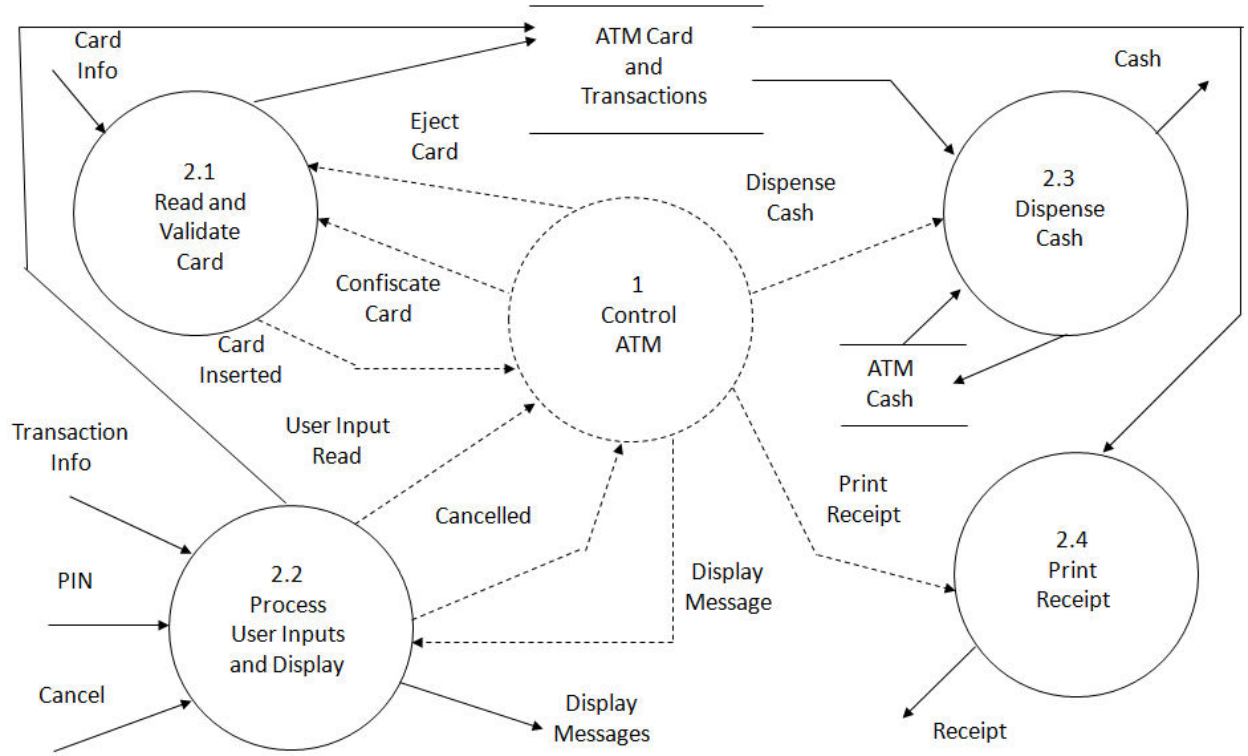
System Context Diagram



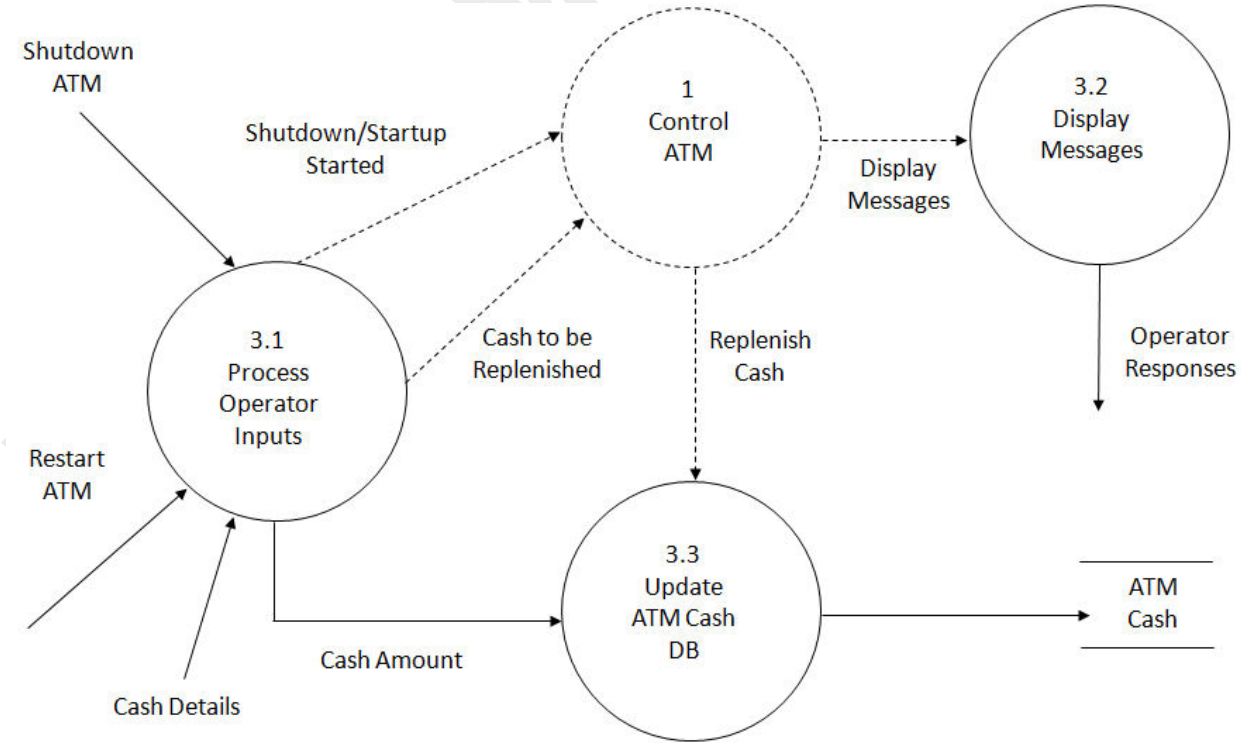
Data Flow Diagram – Level 1



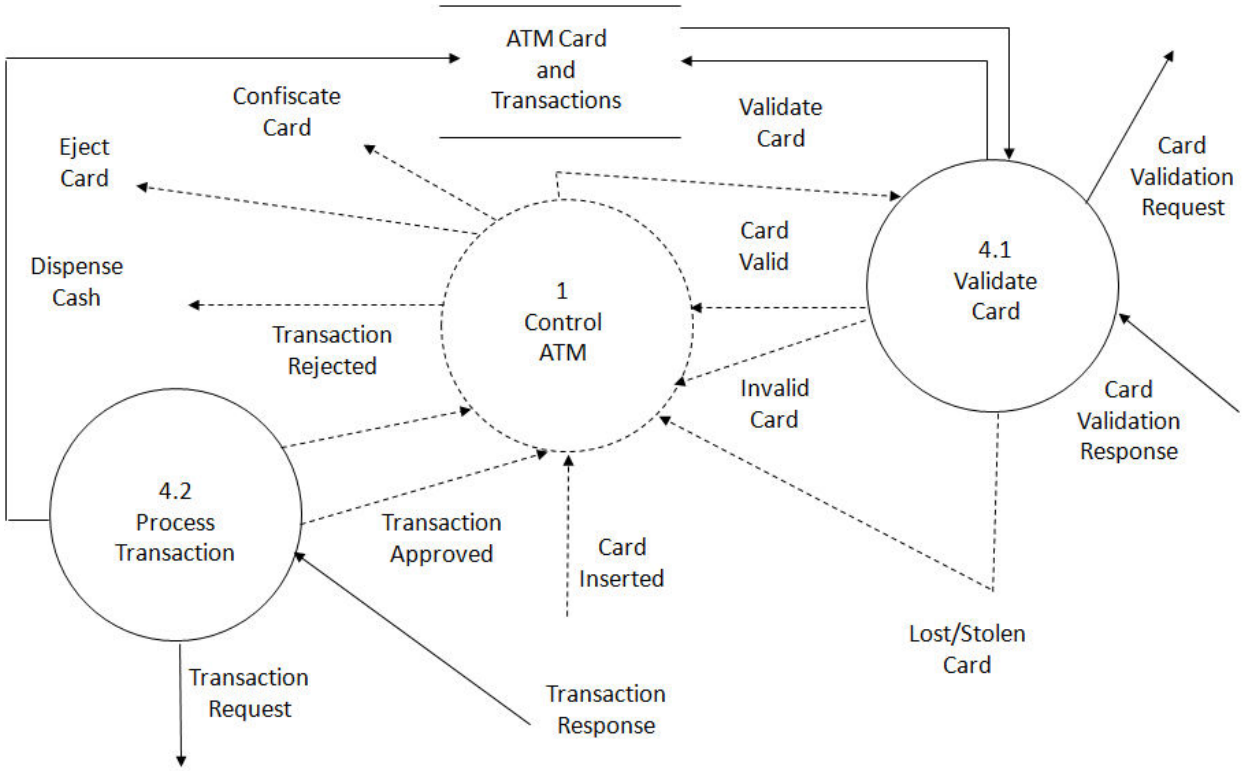
Level 2 DFD: Interface with Customer



Level 2 DFD: Interface with Operator



Level 2 DFD: Interface with Bank



Control Flow Specification

- There is only function here, which has a control flow: Control ATM
- We will discuss control flow in dynamic modeling

Mini Specification

- These are almost equivalent to the use cases
- We had discussed these in quite detail when we talked about use cases

Contents of Data Dictionary

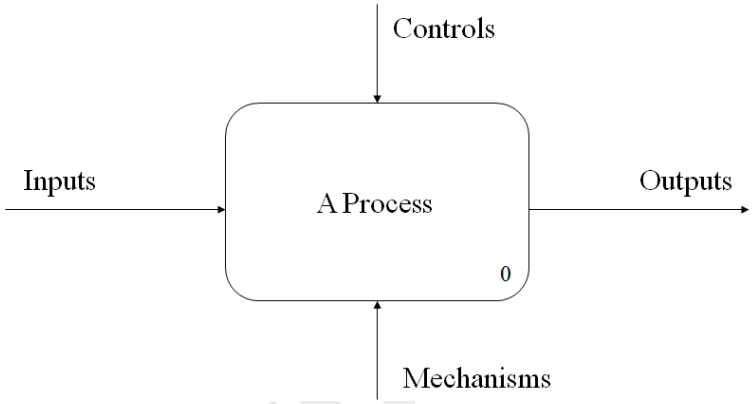
- Name of the data item
- Aliases
- Description/purpose
- Related data items
- Range of values
- Data flows
- Data structure definition/form

Structured Analysis and Design Technique (SADT)

- A model of the problem is constructed, which is composed of hierarchy of diagrams
- Each diagram is composed of boxes and arrows
- The topmost diagram, called the context diagram, defines the problem most abstractly
- As the problem is refined into sub-problems, this refinement is documented into other diagrams
- Boxes should be given unique names that should always be verb phrases, because they represent functions

- All boxes should be numbered in the lower right corner, to reflect their relative dominance
- Arrows may enter top, left, or bottom sides of the box, and can exit only from the right side of the box
- An arrow pointing into the left side of a box represents things that will be *transformed* by the box. These are inputs
- An arrow pointing down into the top of the box represents *control* that affects how the box transforms the things entering from left side
- Arrows entering the bottom of a box represent *mechanism* and provide the analyst with the ability to document how the function will operate, who will perform it, or what physical resources are needed to perform the function

An SADT Context Diagram



IDEF0 Fundamentals

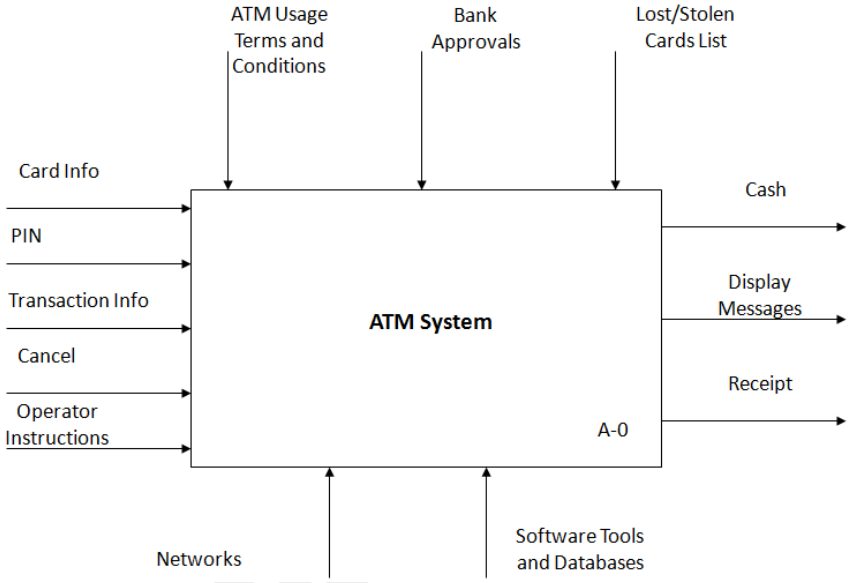
- Diagrams based on simple box and arrow graphics, arrows convey data or objects
- Text labels to describe boxes and arrows and glossary and text to define the precise meanings of diagram elements
- Box name shall be a verb or verb phrase, such as "Perform Inspection", arrows are nouns
- Gradual exposition of detail, with the major functions at the top and with successive levels of sub functions revealing well-bounded detail breakout
- The limitation of detail to no more than six sub functions on each successive function
- A "node chart" that provides a quick index for locating details within the hierarchic structure of diagrams

IDEF0 – Rules

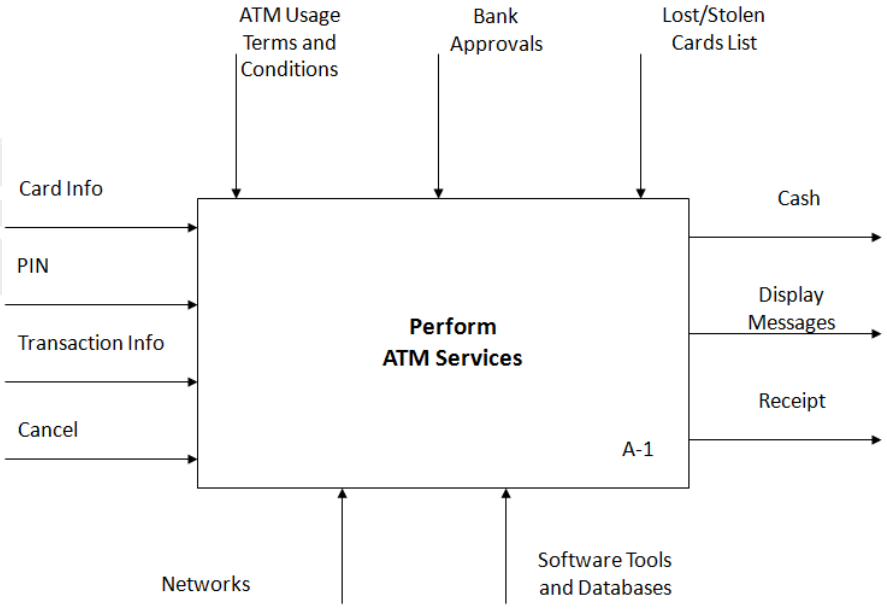
- There is always an A-0 context diagram, and its box number is always 0. This is a strict part of the standard that helps identify the overall description of the system
- A non-context diagram has from 3 to 6 boxes. This helps us manage detail
- Each box is numbered in its lower right corner, generally going upper left to lower right on the diagram. This gives us a consistent way to lay out the diagram
- Arrows have horizontal and/or vertical segments, never diagonal; make diagrams more readable
- Each box has at least one control and output. This keeps us from using boxes with little purpose

- Successive detail diagrams are numbered by "building up" diagram and box numbers. This is one of the most important concepts, that leads to a collection of easy-to-understand diagrams rather than a single confusing one
- Unconnected ends of boundary arrows are identified by their ICOM codes. This helps identify arrows when moving from one diagram to another.
- Fork and/or join arrows, rather than using parallel arrows for the same object. This reduces clutter, and reduces the likelihood that an arrow could be overlooked because it is in another part of the diagram

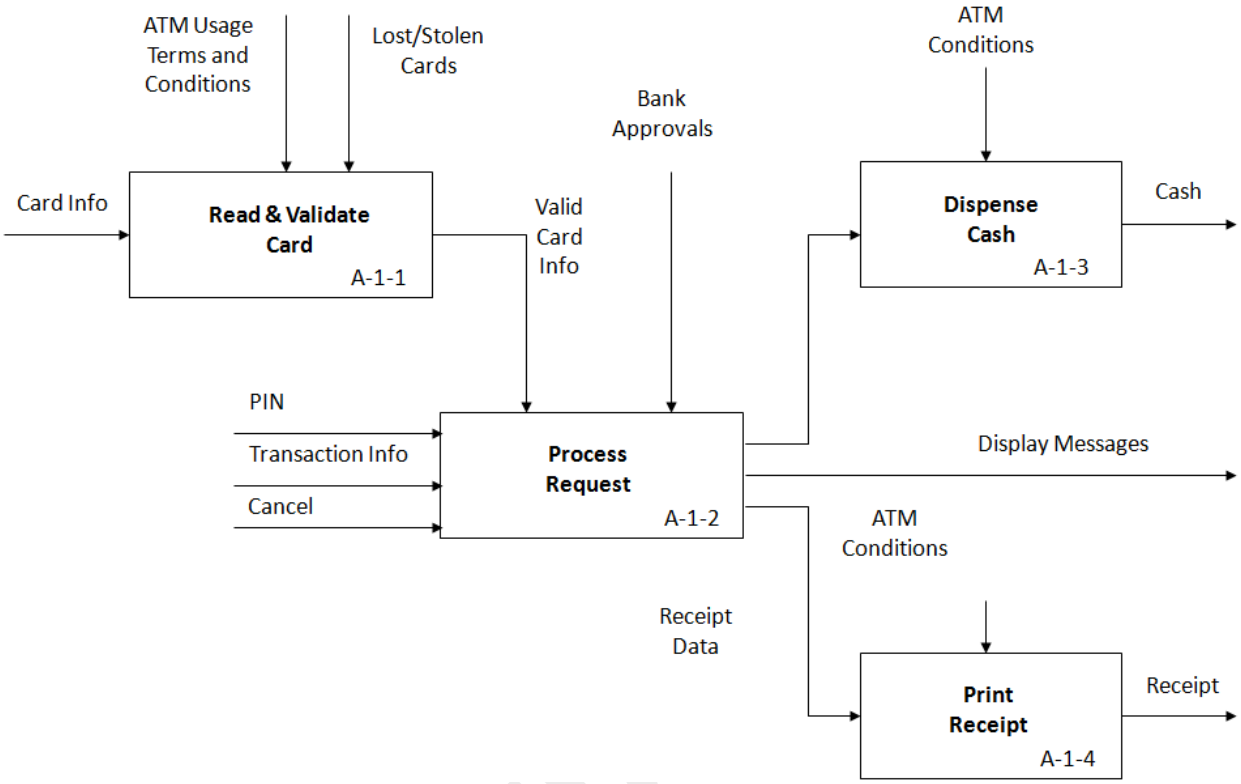
Banking System Context Diagram



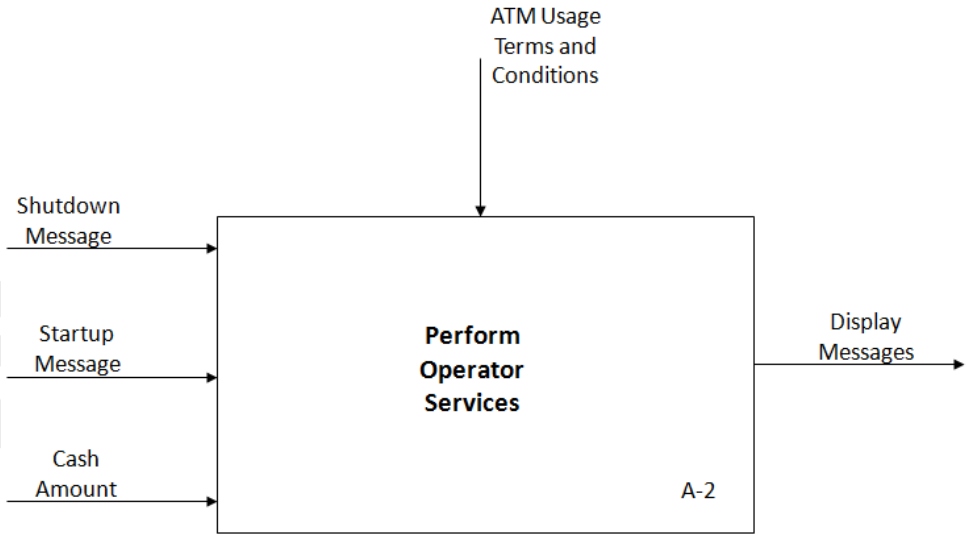
SADT Level 1 Diagram



SADT Level 2



SADT Level 1 Diagram



Application of SADT

- This is an application of SADT technique on the ATM system. This can be decomposed further

Dynamic Modeling

- Temporal relationships are difficult to understand
- A system can first be understood by examining its static structure and the relationships that exist among different elements at single moment in time
- Then we examine changes in these elements and their relationships over time
- Those aspects of a system that are concerned with time and changes are the dynamic model
- Control is an aspect of a system that describes the sequences of operations that occur in response to external stimuli, without consideration of what the operations do, what they operate on, or how they are implemented
- Dynamic modeling deals with flow of control, interactions, and sequencing of operations in a system of concurrently-active objects
- Major dynamic modeling concepts are *events*, which represent external stimuli, and *states*, which represent values of objects
- There are two ways to model dynamic behavior
- One is the life history of one object as it interacts with the rest of the world; the other is the communication patterns of a set of connected objects as they interact to implement behavior
- The view of an object in isolation is a state machine – a view of an object as it responds to events based on its current state, performs actions as part of its response, and transitions to a new state
- This is displayed in state chart diagrams in UML
- The view of a system of interacting objects is a collaboration, a context-dependent view of objects and their links to each other, together with the flow of messages between objects across data links
- Collaboration and sequence diagrams are used for this view in UML

Techniques for Dynamic Modeling

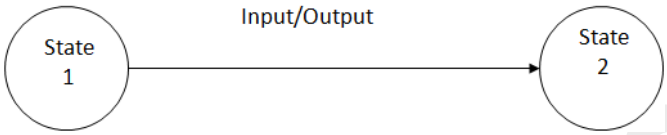
1. Finite state machines (FSM)
2. Statecharts
3. Petri nets
4. Decision tables and decision trees
5. Collaboration diagrams
6. Sequence diagrams

1 – Finite State Machines

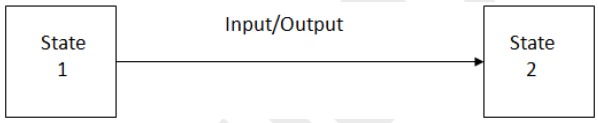
- A finite state machine (FSM) is a hypothetical machine that can be in only one of a given number of states at any specific time
- In response to an input, the machine generates an output and changes state
- Both the output (O) and the next state (S_N) are purely functions of the current state (S_C) and the input (I)
  - ✓ $S_N = F(S_C, I)$
  - ✓ $O = G(S_C, I)$
- There are two notations commonly used for finite state machines
  - ✓ State transition diagrams (STDs)
  - ✓ State transition matrices (STMs)

State Transition Diagrams

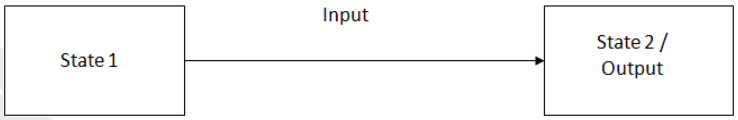
- A circle denotes a state
- A directed arc connecting two states denotes the potential to transition between the two indicated states
- A label on the arc, which has two parts separated by a slash, means the input that triggers the transition and the output with which the system responds
- Mealy model of state transition diagrams



- Most structured analysis tools use a slightly different notation to describe state transition diagrams
- They use boxes instead of circles
- This notation has become very popular in software engineering

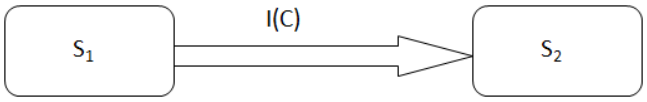


- Another model exists for state transition diagrams, Moore model, in which system responses are associated with the state rather than the transition between states
- On Moore STDs, arcs are labeled with only the stimulus name, and circles are labeled with the state name and the system response

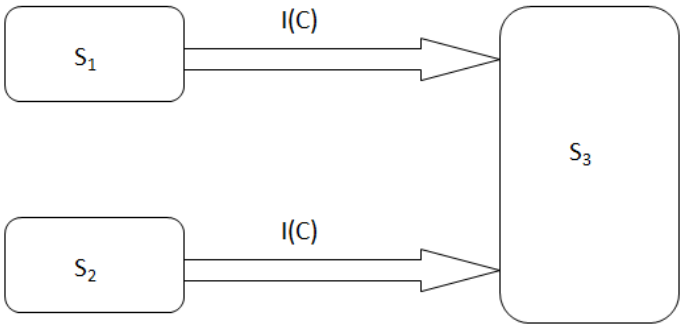


2 – Statecharts

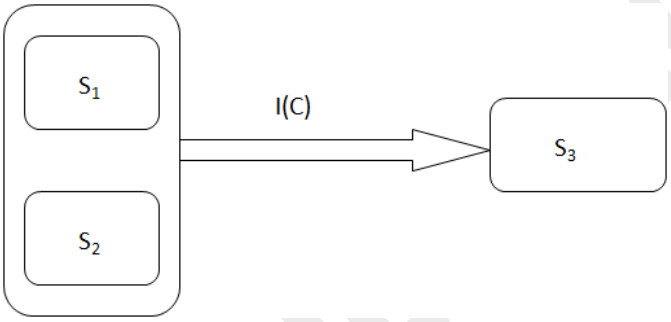
- Statecharts are an extension to finite state machines, proposed by Harel
- They provide a notation and a set of conventions that facilitate the hierarchical decomposition of finite state machines and a mechanism for communication between concurrent finite state machine
- Statecharts allow a transition to be a function of not only an external stimulus but also the truth of a particular condition



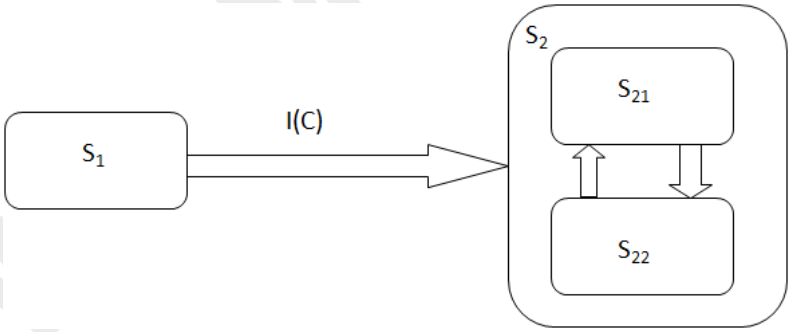
- The concept of superstate, which can be used to aggregate sets of states with common transitions

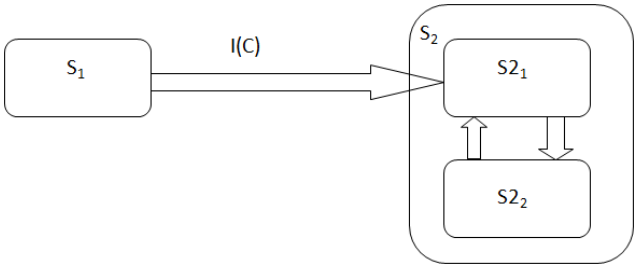


- A super-state can be used an abstract state, in which all states transition to a particular state

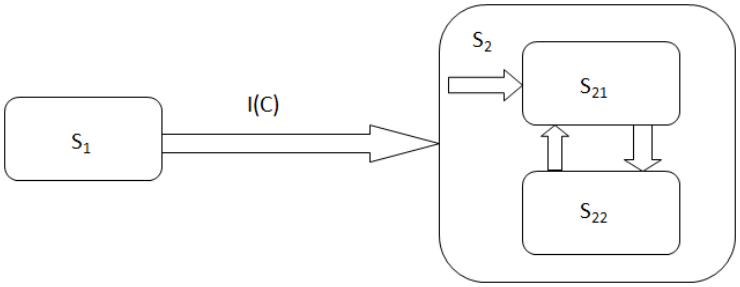


- A super-state can be used to transition into from more than one states

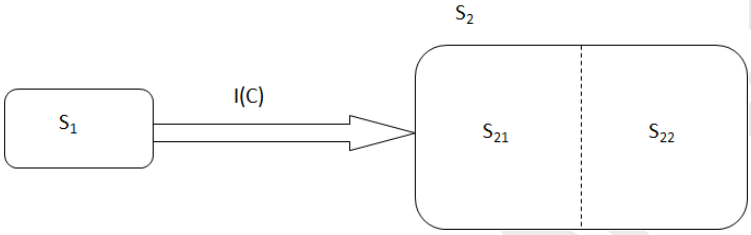




- Default entry state/ 'or' decomposition

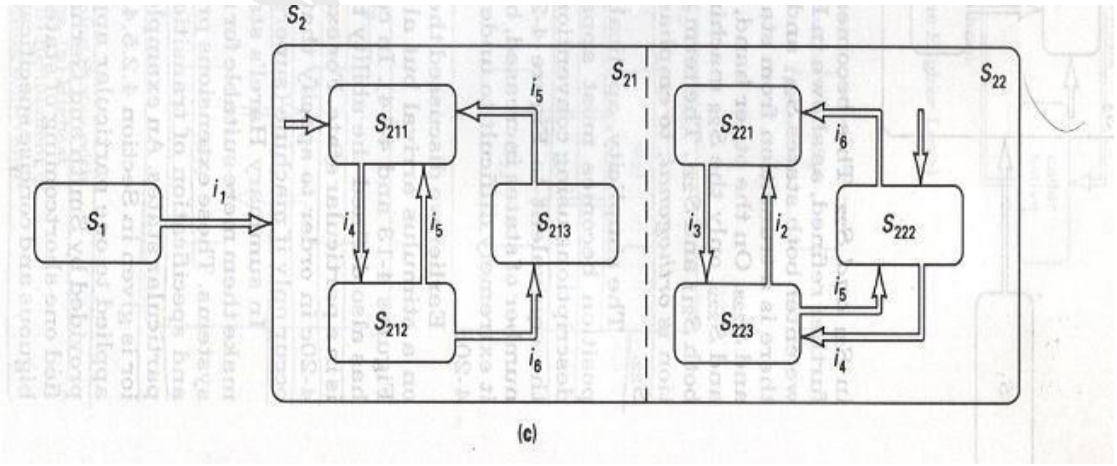


- Statecharts describes the 'and' function of state refinement

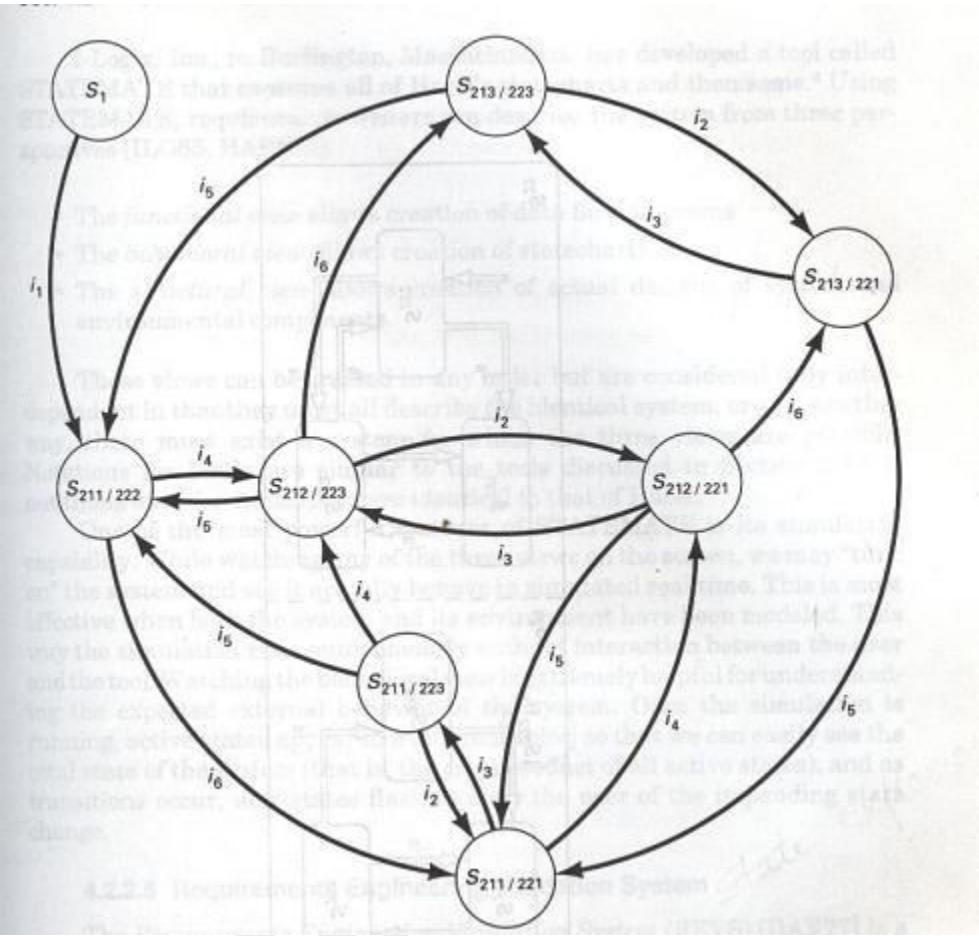


- The simplicity, applicability, and elegance of Harel's orthogonal decomposition becomes most apparent when we compare equivalent behavioral descriptions using conventional state transition diagrams
- Statecharts provide natural extensions to FSMs to make them more suitable for specifying external behavior of real-time systems
- These extensions provide for hierarchical decomposition of states and specification of transitions dependent on global conditions and being in particular state

Orthogonal Decomposition using Statecharts



Explosion of States



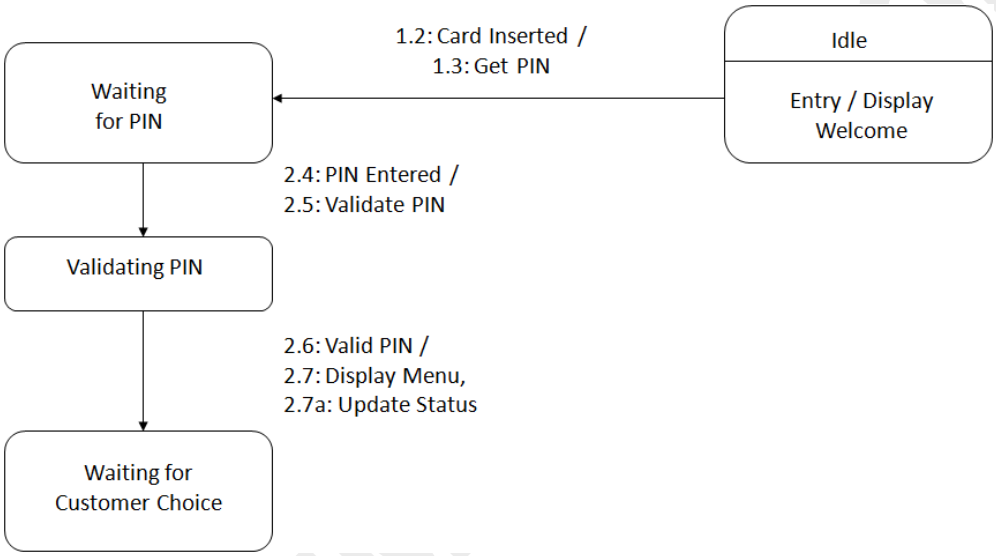
Dynamic Modeling (Continued)

- There are two ways to model dynamic behavior
- One is the life history of one object as it interacts with the rest of the world; the other is the communication patterns of a set of connected objects as they interact to implement behavior
- The view of an object in isolation is a state machine – a view of an object as it responds to events based on its current state, performs actions as part of its response, and transitions to a new state
- This is displayed in state chart diagrams in UML
- The view of a system of interacting objects is a collaboration, a context-dependent view of objects and their links to each other, together with the flow of messages between objects across data links
- Collaboration and sequence diagrams are used for this view in UML
- The dynamic model depicts the interaction among the objects that participate in each use case
- The starting point for developing the dynamic model is the use case and the objects determined during object structuring

Dynamic Modeling of Banking System Case Study

- The Client Validate PIN and Client Withdraw Funds client use cases are state-dependent use cases. The state-dependent aspects of the use case are defined by the ATM Control object, which executes the ATM statechart
- The Client Validate PIN use case starts with the customer inserting the ATM card into the card reader
- The statechart for ATM Control for the Validate PIN use case is shown next

Statechart for ATM Control: Validate PIN Use Case

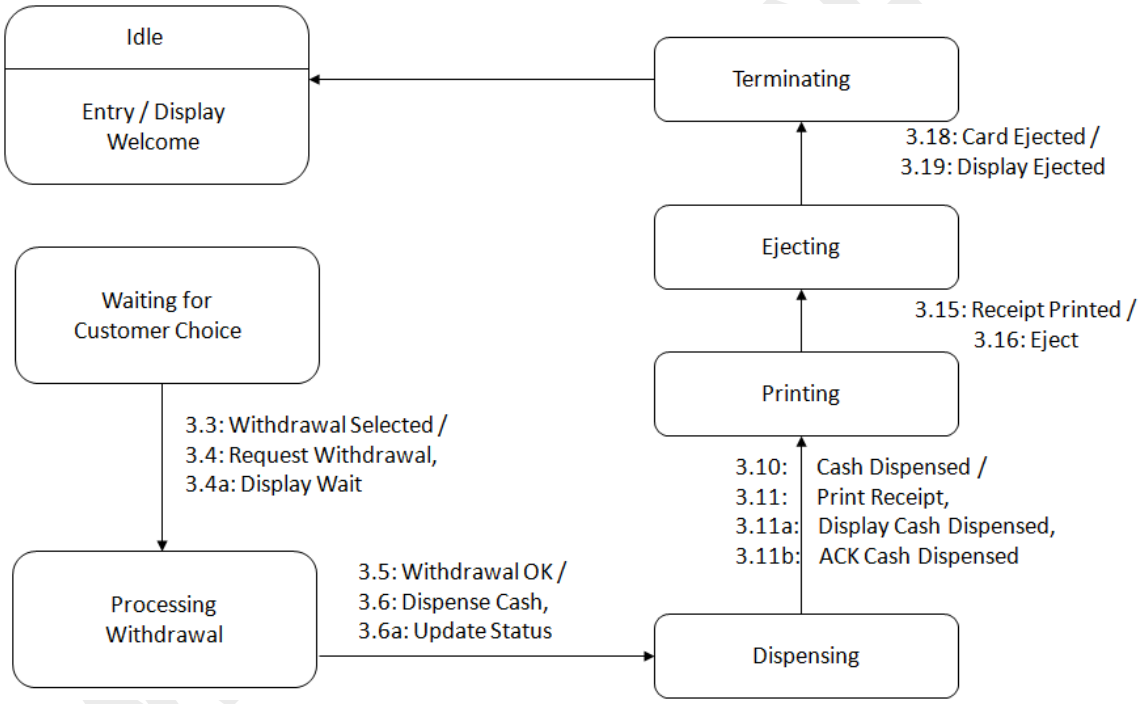


Statechart for ATM Control: Validate PIN Use Case

- **1:** The ATM Customer actor inserts the ATM card into the Card Reader. The Card Reader Interface object reads the card input
- **1.1:** The Card Reader Interface object sends the Card Input Data, containing card ID, start Date, and expiration Date, to the entity ATM Card
- **1.2:** Card Reader Interface sends the Card Inserted event to ATM Control. As a result, the ATM Control statechart transitions from Idle state (the initial state) to Waiting for PIN state. The output event associated with this transition is Get PIN
- **1.3:** ATM Control sends the Get PIN event to Customer Interface
- **1.4:** Customer Interface displays the Pin Prompt to the ATM Customer actor
- **2:** ATM Customer inputs the PIN to the Customer Interface object
- **2.1:** Customer Interface requests Card Data from ATM Card
- **2.2:** ATM Card provides the Card Data to the Customer Interface
- **2.3:** Customer Interface sends the Customer Info, containing card ID, PIN, start Date, and expiration Date, to the ATM Transaction entity object

- **2.4:** Customer Interface sends the PIN Entered (Customer Info) event to ATM Control. This causes ATM Control to transition from Waiting for PIN state to Validating PIN state. The output event associated with this transition is Validate PIN
- **2.5:** ATM Control sends a Validate PIN (Customer Info) request to the Bank Server
- **2.6:** Bank Server validates the PIN and sends a Valid PIN response to ATM Control. As a result of this event, ATM Control transitions to Waiting for Customer Choice state. The output events for this transition are Display Menu and Update Status
- **2.7:** ATM Control sends the Display Menu event to the Customer Interface
- **2.7a:** ATM Control sends an Update Status message to the ATM Transaction
- **2.8:** Customer Interface displays a menu showing the Withdraw, Query, and Transfer options to the ATM Customer actor

Statechart for ATM Control: Withdraw Funds Use Case



- **3:** ATM Customer actor inputs withdrawal selection to Customer Interface, together with the account number for checking or savings account and withdrawal amount
- **3.1:** Customer Interface sends the customer selection to ATM Transaction
- **3.2:** ATM Transaction responds to Customer Interface with Transaction Details. Transaction Details contains transaction ID, card ID, PIN, date, time, account Number, and amount
- **3.3:** Customer Interface sends the Withdrawal Selected (Transaction Details) request to ATM Control. ATM Control transitions to Processing Withdrawal state. Two output events are associated with this transition, Request Withdrawal and Display Wait
- **3.4:** ATM Control sends a Request Withdrawal transaction containing the Transaction Details to

the Bank Server

- **3.4a:** ATM Control sends a Display Wait message to Customer Interface
- **3.4a.1:** Customer Interface displays the Wait Prompt to the ATM Customer
- **3.5:** Bank Server sends a Withdrawal OK (Cash Details) response to ATM Control. Cash Details contains the amount to be dispensed and the account balance. This event causes ATM Control to transition to Dispensing state. The output events are Dispense Cash and Update Status

- **3.6:** ATM Control sends a Dispense Cash (Cash Details) message to Cash Dispenser Interface
- **3.6a:** ATM Control sends an Update Status (Cash Details) message to ATM Transaction
- **3.7:** Cash Dispenser Interface sends the Cash Withdrawal Amount to ATM Cash
- **3.8:** ATM Cash sends a positive Cash Response to the Cash Dispenser Interface
- **3.9:** Cash Dispenser Interface sends the Dispenser Output command to the Cash Dispenser external output device to dispense cash to the customer
- **3.10:** Cash Dispenser Interface sends the Cash Dispensed event to ATM Control. As a result, ATM Control transitions to Printing state. The three output events associated with this transition are Print Receipt, Display Cash Dispensed, and ACK Cash Dispensed

- **3.11:** ATM Control sends Print Receipt event to Receipt Printer
- **3.11a:** ATM Control requests Customer Interface to Display Cash Dispensed message
- **3.11a.1:** Customer Interface displays Cash Dispensed prompt to ATM Customer
- **3.11b:** ATM Control sends an Acknowledge Cash Dispensed message to the Bank Server
- **3.12:** Receipt Printer Interface requests Transaction Data from ATM Transaction
- **3.13:** ATM Transaction sends the Transaction Data to the Receipt Printer Interface
- **3.14:** Receipt Printer Interface sends the Printer Output to the Receipt Printer external output device

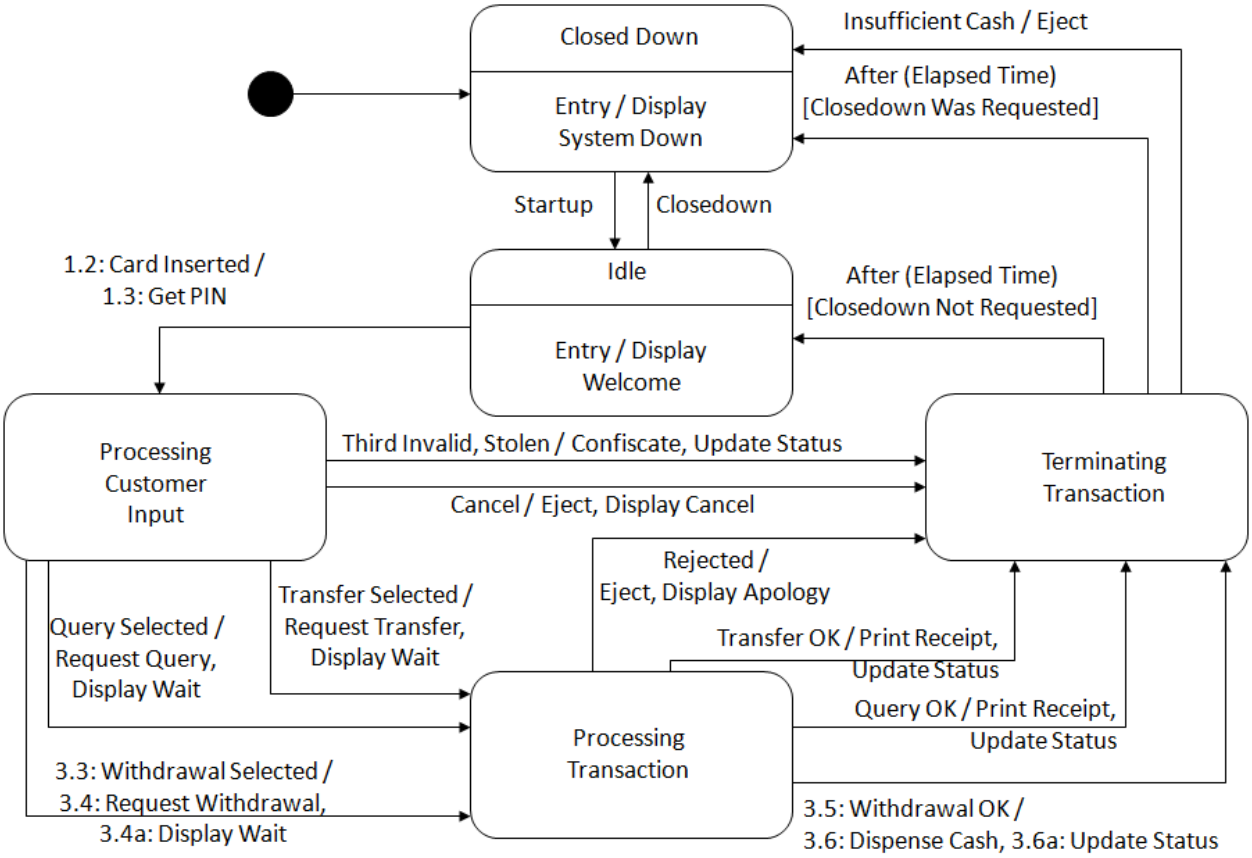
- **3.15:** Receipt Printer Interface sends the Receipt Printed event to ATM Control. As a result, ATM Control transitions to Ejecting state. The output event is Eject
- **3.16:** ATM Control sends the Eject event to Card Reader Interface
- **3.17:** Card Reader Interface sends the Card Reader Output to the Card Reader external I/O device

- **3.18:** Card Reader Interface sends the Card Ejected event to ATM Control. ATM Control transitions to Terminating state. The output event is Display Ejected
- **3.19:** ATM Control sends the Display Ejected event to the Customer Interface
- **3.20:** Customer Interface displays the Card Ejected prompt to the ATM Customer

ATM Statecharts

- A hierarchical statechart for the ATM Control class is needed, which can be decomposed further

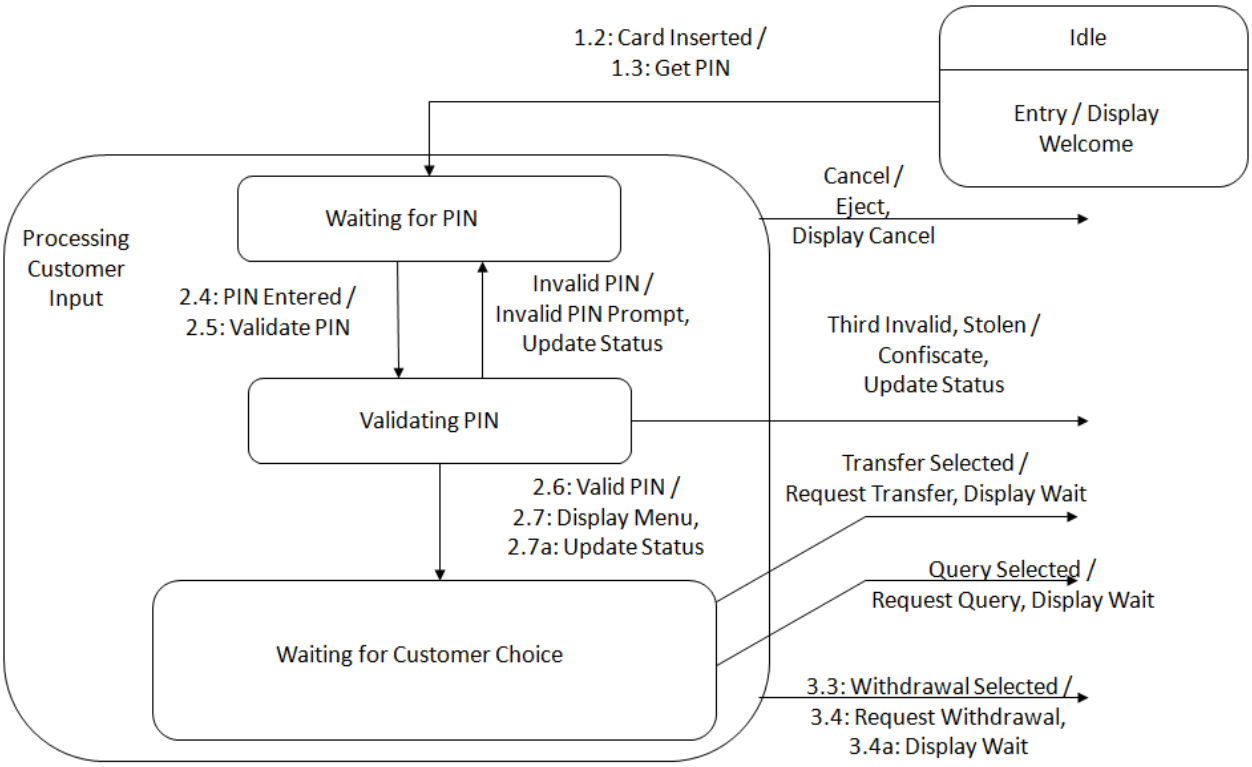
Top-Level ATM Control Statechart



- Five states are shown on the top-level statechart
  - ✓ Closed Down (initial state)
  - ✓ Idle
  - ✓ Processing Customer Input (superstate)
  - ✓ Processing Transaction (superstate)
  - ✓ Terminating Transaction (superstate)

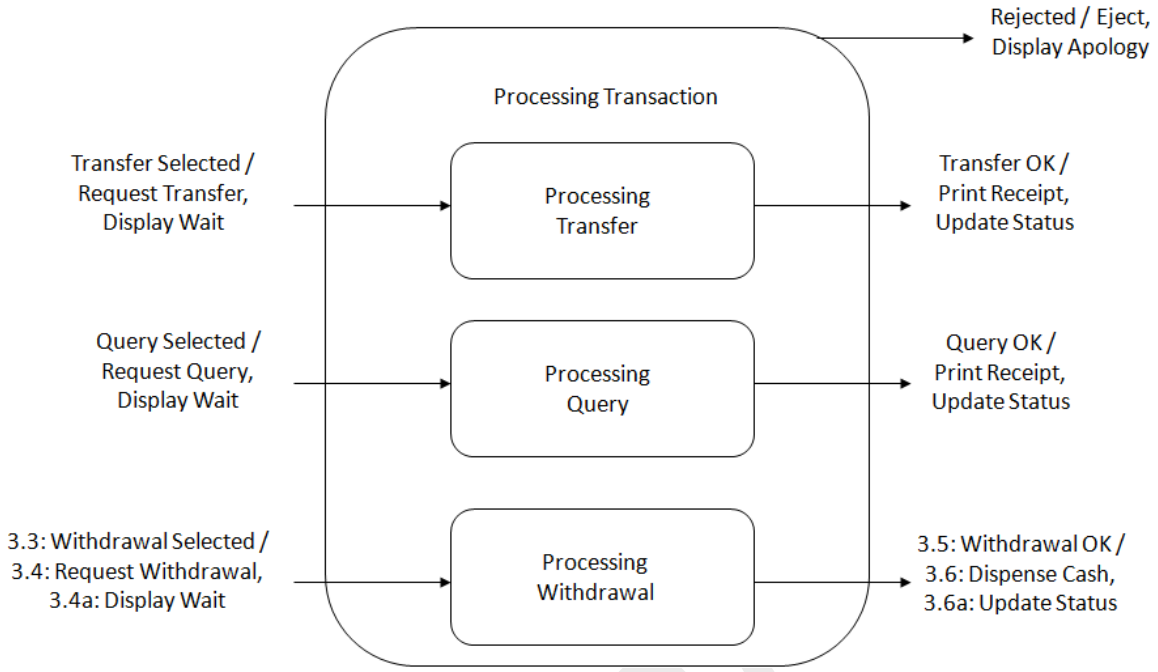
- At system initialization time, given by the event Startup, the ATM transitions from the initial Closed Down state to Idle state. The event Display Welcome message is triggered on entry into idle state. In Idle state, the ATM is for a customer-initiated event

ATM Control Statechart: Processing Customer Input Superstate

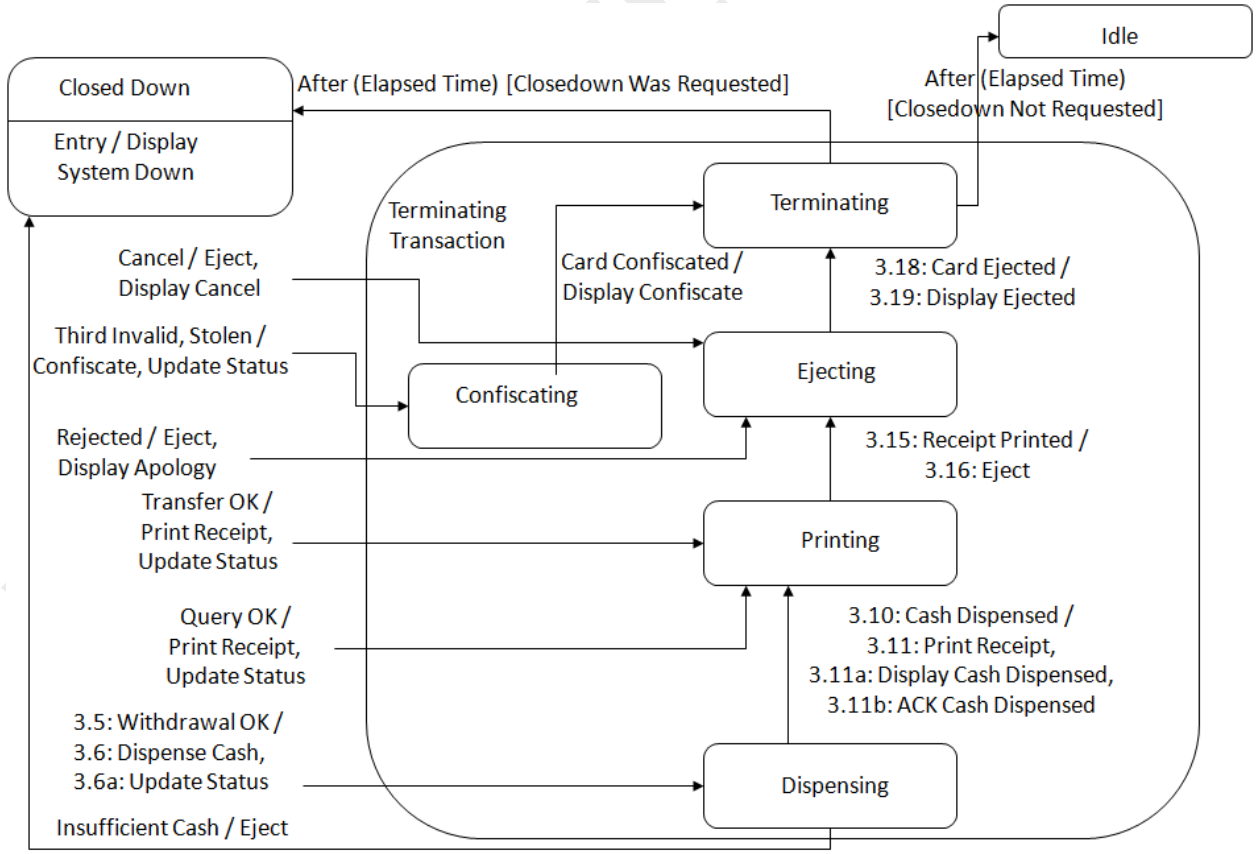


- Processing Customer Input Superstate
  - ✓ The Processing Customer Input superstate is decomposed into three substates
  - ✓ Waiting for PIN
  - ✓ Validating PIN
  - ✓ Waiting for Customer Choice
- Waiting for PIN Substate
  - ✓ This substate is entered from Idle state when the customer inserts the card in the ATM, resulting in the Card Inserted event. In this state, the ATM waits for the customer to enter the PIN
- Validating PIN Substate
  - ✓ This substate is entered when the customer enters the PIN. In this substate, the Bank Server validates the PIN
- Waiting for Customer Choice Substate
  - ✓ This substate is entered as a result of a Valid PIN event, indicating a valid PIN was entered. In this state, the customer enters a selection: Withdraw, Transfer, or Query

ATM Control Statechart: Processing Transaction Superstate



ATM Control Statechart: Terminating Transaction Superstate



Processing Transaction Superstate

- This superstate is also decomposed into three substates
  - ✓ Processing Withdrawal
  - ✓ Processing Transfer
  - ✓ Processing Query
- Depending on customer's selection the appropriate substate within Processing Transaction is entered, during which the customer's request is processed

Terminating Transaction Superstate

- This superstate has five substates
  - ✓ Dispensing
  - ✓ Printing
  - ✓ Ejecting
  - ✓ Confiscating
  - ✓ Terminating

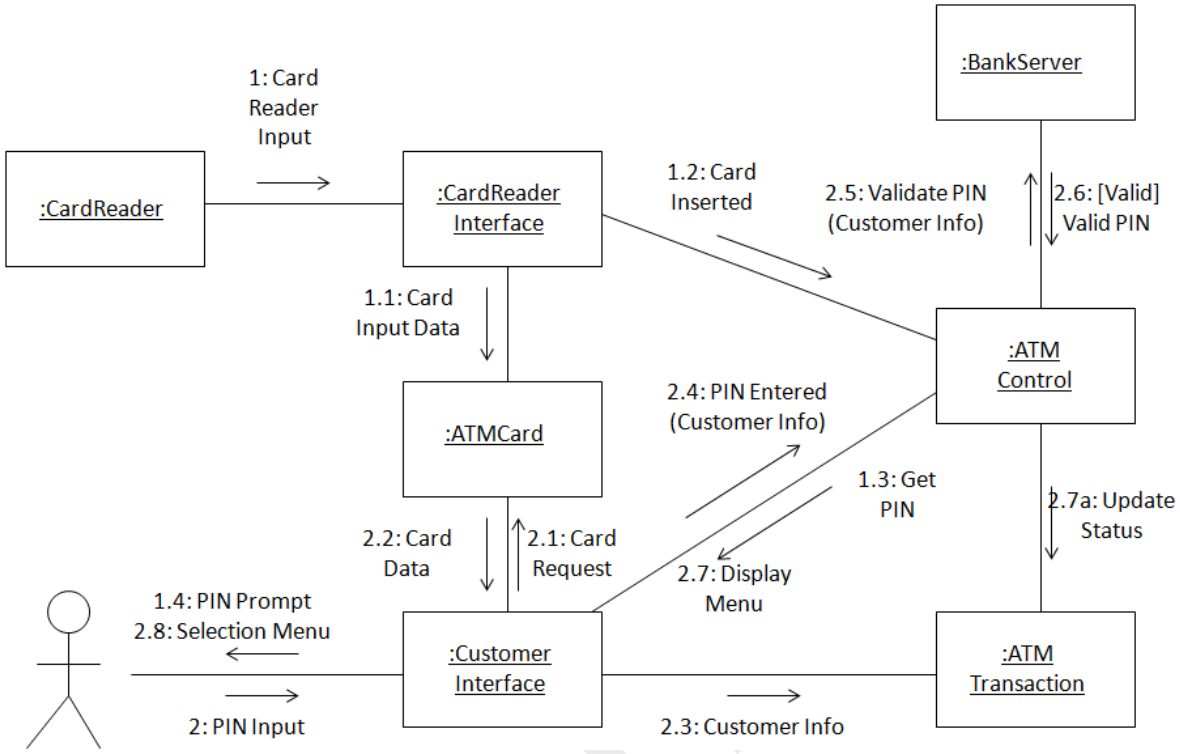
Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects a system. For the most part, this involves modeling concrete or prototypical instances of classes, interfaces, components, and nodes, along with messages that are dispatched among them, all in the context of a scenario that illustrates a behavior
- Interaction diagrams may stand alone to visualize, specify, construct, and document the dynamics of a particular society of objects, or they may be used to model one particular flow of control of a use case
- There are types of Interaction Diagrams
- Sequence diagrams
  - ✓ A sequence diagram is an interaction diagram that emphasizes the time ordering of messages
  - ✓ Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis
- Collaboration diagrams
  - ✓ A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages
  - ✓ Graphically, a collaboration diagram is a collection of vertices and arcs

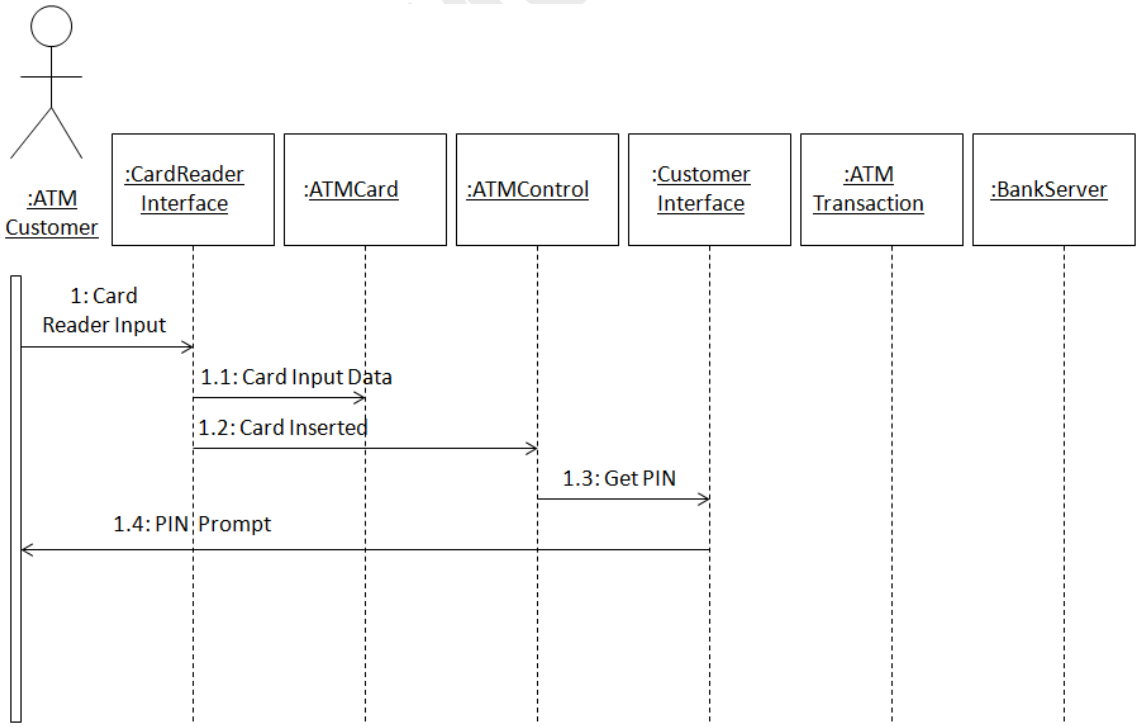
Hints and Tips on Interaction Diagrams

- Give it a name that communicates its purpose
- Use a sequence diagram if you want to emphasize the time ordering of messages
- Use a collaboration diagram if you want to emphasize the organization of the objects involved in the interaction
- Lay out its elements to minimize lines that cross
- Use notes and color as visual cues to draw attention to important features of your diagram

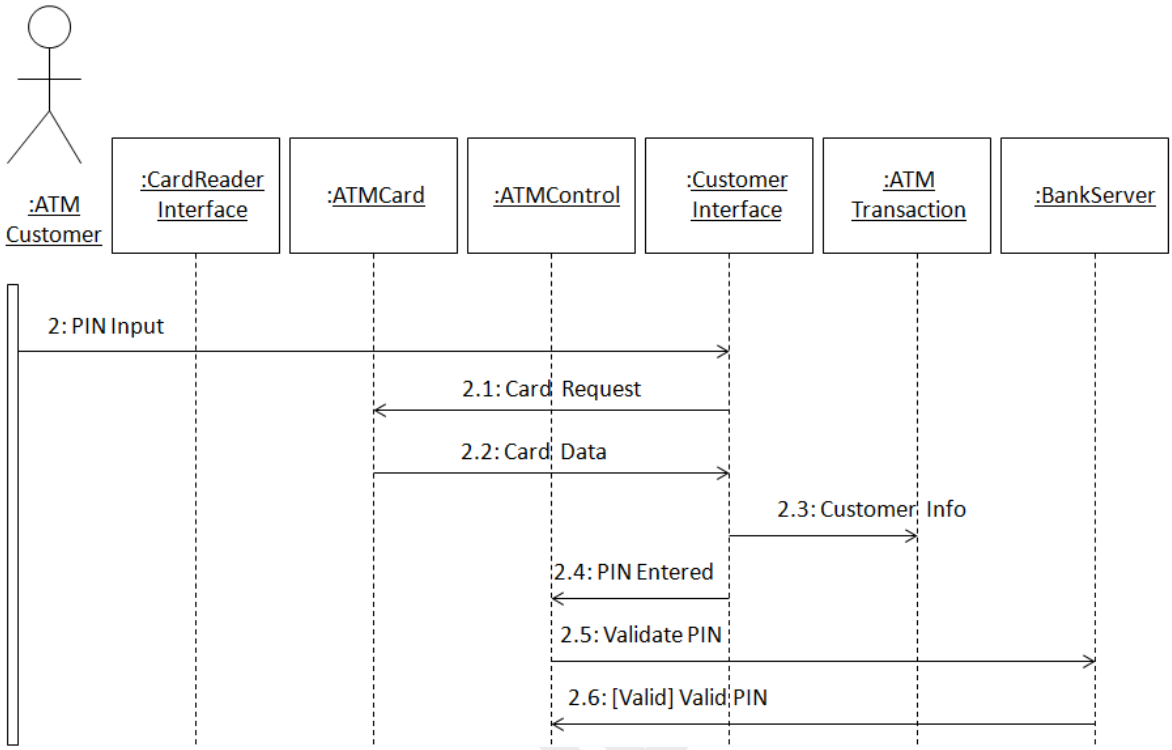
Collaboration Diagram: ATM Client Validate PIN Use Case



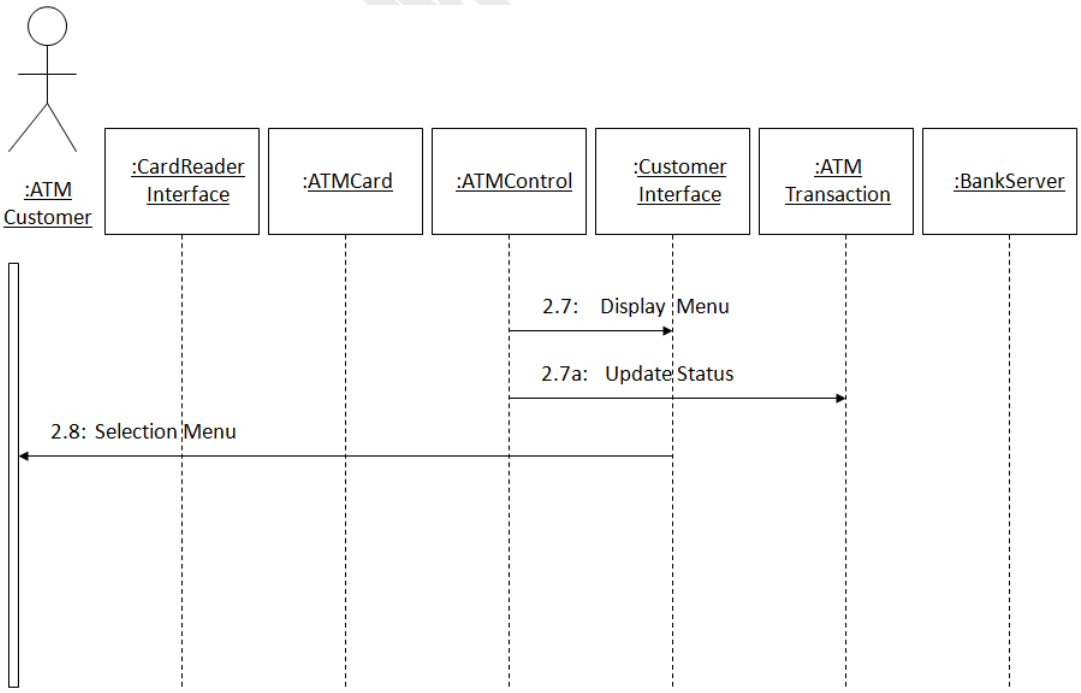
Sequence Diagram: ATM Client Validate PIN Use Case – 1



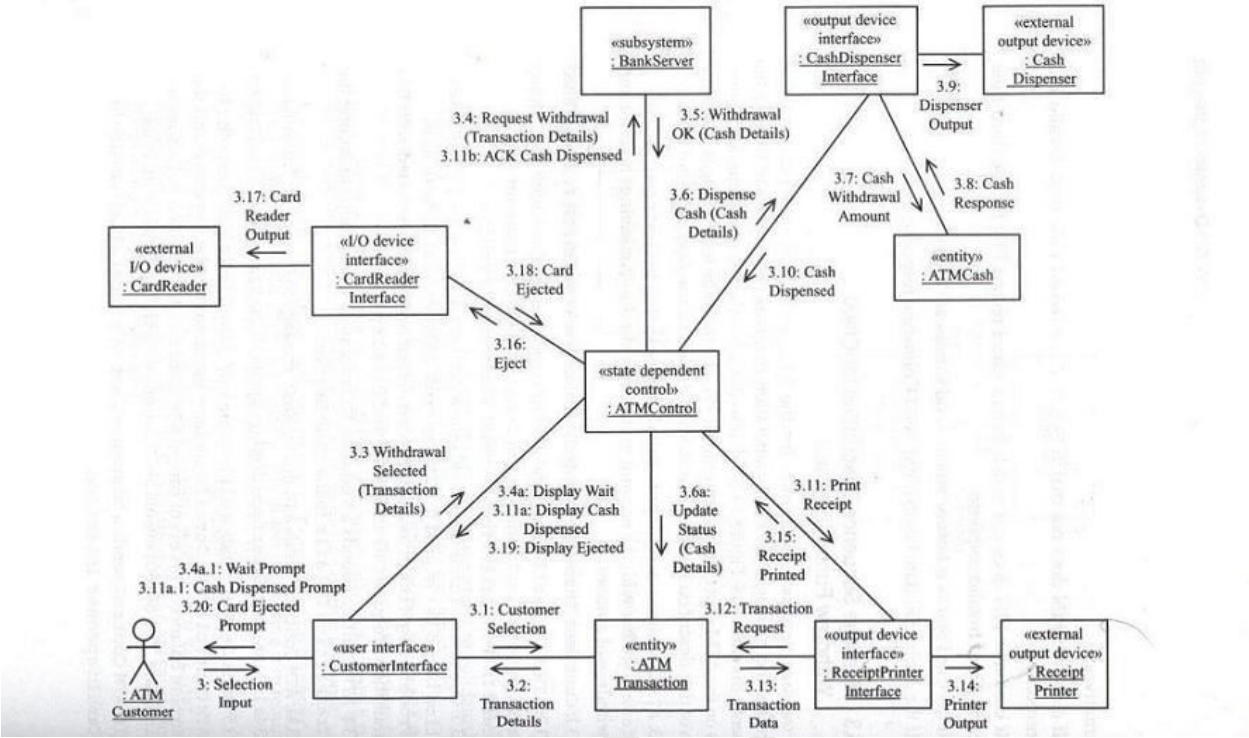
Sequence Diagram: ATM Client Validate PIN Use Case – 2



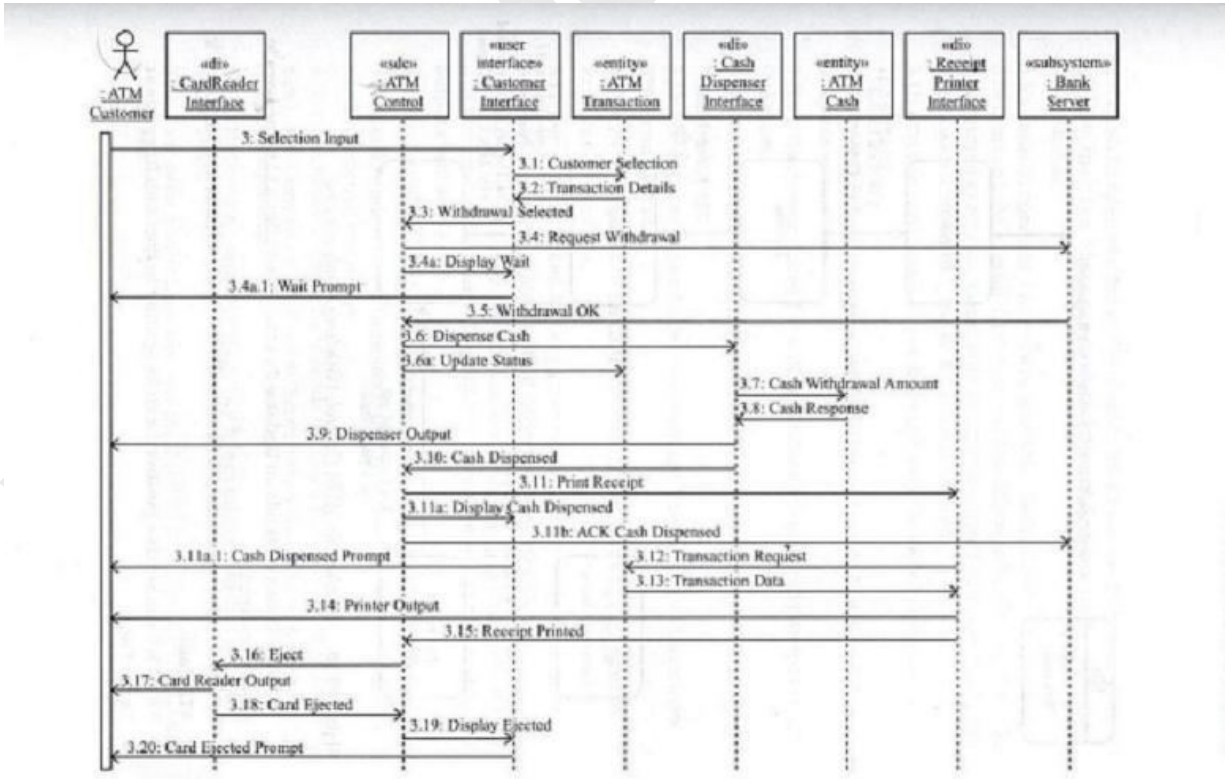
Sequence Diagram: ATM Client Validate PIN Use Case – 3



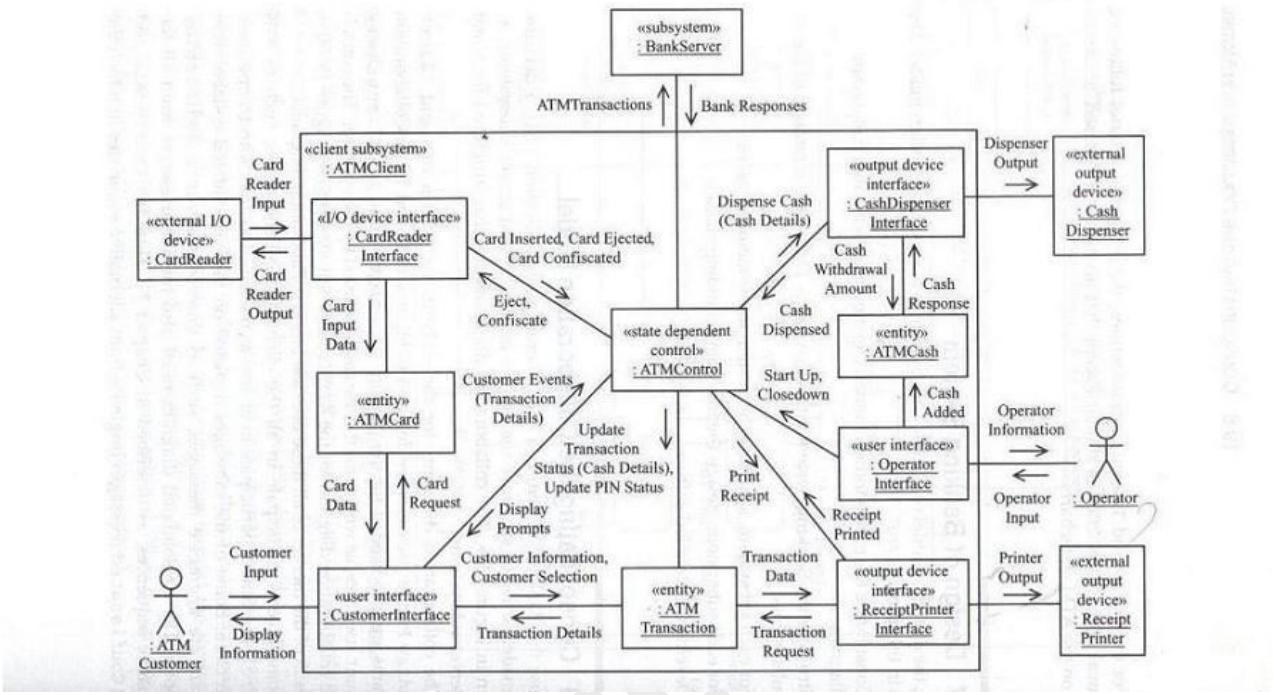
Collaboration Diagram: ATM Client Withdraw Funds Use Case



Sequence Diagram: ATM Client Withdraw Funds Use Case



Consolidated Collaboration Diagram for ATM Client Subsystem



Requirements Document for the Banking System

Requirements Document (Review)

- The requirements document is a formal document used to communicate the requirements to customers, engineers and managers
- It is also known as software requirements specifications or SRS
- The services and functions which the system should provide
- The constraints under which the system must operate
- Overall properties of the system i.e., constraints on the system's emergent properties

SRS for the Banking System

- Preface
  - ✓ This should define the expected readership of the document and describe its version history including a rationale for creation of a new version and a summary of the changes made in each version
- Introduction
  - ✓ This should define the product in which the software is embedded, its expected usage and present an overview of the functionality of the control software
- Glossary
  - ✓ This should define all technical terms and abbreviations used in the document
- Specific requirements

- ✓ This should define specific requirements for the system using natural language with the help of diagrams, where appropriate

- Appendices
  - ✓ Use-case model
  - ✓ Object model
  - ✓ Data-flow model

Software Requirements Specifications for the Banking System

1. Preface

- This document, Software Requirements Specification (SRS), is created to document the software requirements for the Banking System, as described in section 2, Introduction, of this document
- This document was created on the request of the 'XYZ Bank Inc.' – the 'Client'. The creator of this document is 'A Software House Inc.' – 'Vendor'. The 'Client' has asked the 'Vendor' to develop an SRS for the Banking System. The 'Vendor' will also be responsible for the development of the software based on this SRS
- This is the first version of the SRS.

2. Introduction

- This section documents an overview of the functionality expected from the software for the Banking System
- We'll review the functionality of the software to be developed

- A bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.
- If the PIN is validated satisfactorily, the customer is prompted for a withdrawal, query, or transfer transaction. Before withdrawal transaction can be approved, the system determines that sufficient funds exist in the requested account, that the maximum daily limit will not be exceeded, and that there are sufficient funds available at the local cash dispenser. If the transaction is approved, the requested amount of cash is dispensed, a receipt is printed containing information about the transaction, and the card is ejected. Before a transfer transaction can be approved, the system determines that the customer has at least two

accounts and that there are sufficient funds in the account to be debited. For approved query and transfer requests, a receipt is printed and card ejected. A customer may cancel a transaction at any time; the transaction is terminated and the card is ejected. Customer records, account records, and debit card records are all maintained at the server.

- An ATM operator may start up and close down the ATM to replenish the ATM cash dispenser and for routine maintenance. It is assumed that functionality to open and close accounts and to create, update, and delete customer and debit card records is provided by an existing system and is not part of this problem.

3. Glossary

- ATM: Automated Teller Machine
- PIN: Personal Identification Number

4. Specific Requirements

1. The XYZ Bank Inc. can have many automated teller machines (ATMs), and the new software system shall provide functionality on all ATMs.
2. The system shall enable the customers of XYZ Bank Inc., who have valid ATM cards, to perform three types of transactions; 1) withdrawal of funds, 2) Query of account balance, and 3) transfer of funds from one bank account to another account in the same bank.
3. An ATM card usage shall be considered valid if it meets the following conditions:
    a. The card was issued by an authorized bank.
    b. The card is used after the start date, i.e., the date when the card was issued.
    c. The card is used before the expiration date, i.e., the date when the card expires.
    d. The card has not been reported lost or stolen by the customer, who had been issued that card.
    e. The customer provides correct personal identification number (PIN), which matches the PIN maintained by the system.
4. The system shall confiscate the ATM card if it detects that a lost or stolen card has been inserted by a customer. The system shall also display an apology to the customer.
5. The system shall allow the customer to enter the correct PIN in no more three attempts. The failure to provide correct PIN in three attempts shall result in the confiscation of the ATM card.
6. The system shall ask for the transaction type after satisfactory validation of the customer PIN. The customer shall be given three options: withdrawal transaction, or query transaction, or transfer transaction.
7. If a customer selects withdrawal transaction, the system shall prompt the customer to enter account number and amount to be dispensed.
8. For a withdrawal transaction, the system shall determine that sufficient funds exist in the requested account, that the maximum daily limit has not be exceeded, and that there are sufficient funds available at the local cash dispenser.
9. If a withdrawal transaction is approved, the requested amount of cash shall be dispensed, a receipt shall be printed containing information about the transaction, and the card shall be

ejected. The information printed on the receipt includes transaction number, transaction type, amount withdrawn, and account balance.

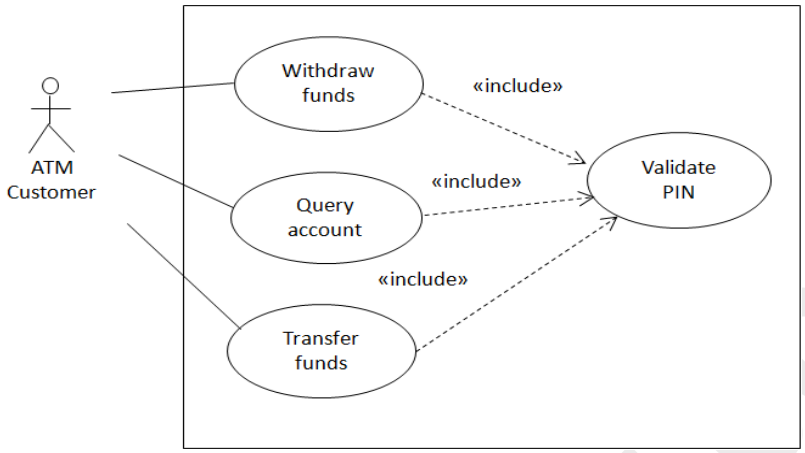
10. If a customer selects query transaction, the system shall prompt the customer to enter account number.
11. If a query transaction is approved, the system shall print a receipt and eject the card. The information contained on the receipt includes transaction number, transaction type, and account balance.
12. If a customer selects transfer transaction, the system shall prompt the customer to enter from account number, to account number, and amount to be transferred.
13. The system shall check if there are enough funds available in the from account, which are being requested for transfer to the to account.
14. If the transfer transaction is approved, a receipt shall be printed and card shall be ejected. The information printed on the receipt includes transaction number, transaction type, amount transferred, and account balance.
15. The system shall cancel any transaction if it has not been completed if the customer presses the Cancel button
16. The customer records, account records, and debit card records will all be maintained at the server and shall not be the responsibility of the system.
17. The system shall enable an ATM operator to shutdown or start up an ATM for routine maintenance.
18. The system shall enable an ATM operator to add cash to the cash dispenser.
19. The system shall not be responsible for opening or closing of accounts, and to create, update, and delete customer and debit card records. These tasks are performed elsewhere by a bank.
20. The system shall be linked with the bank server through communication systems, which are beyond the scope of the current system. It is assumed that this facility is always available.
21. The system shall not be responsible for the maintenance of the hardware devices of the ATM or network facilities.

5. Appendices

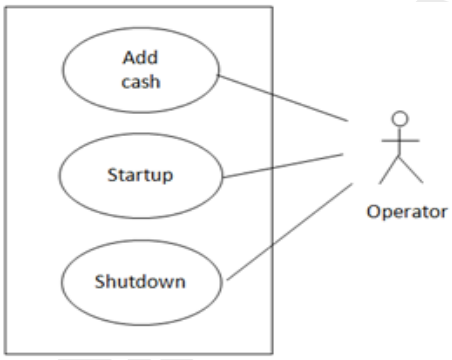
5.1 Use-case model
5.2 Object model
5.3 Functional model
    5.3.1 Data-flow model
    5.3.2 SADT model
5.4 Dynamic model
    5.4.1 Statecharts
    5.4.2 Interaction diagrams

Use Case Model

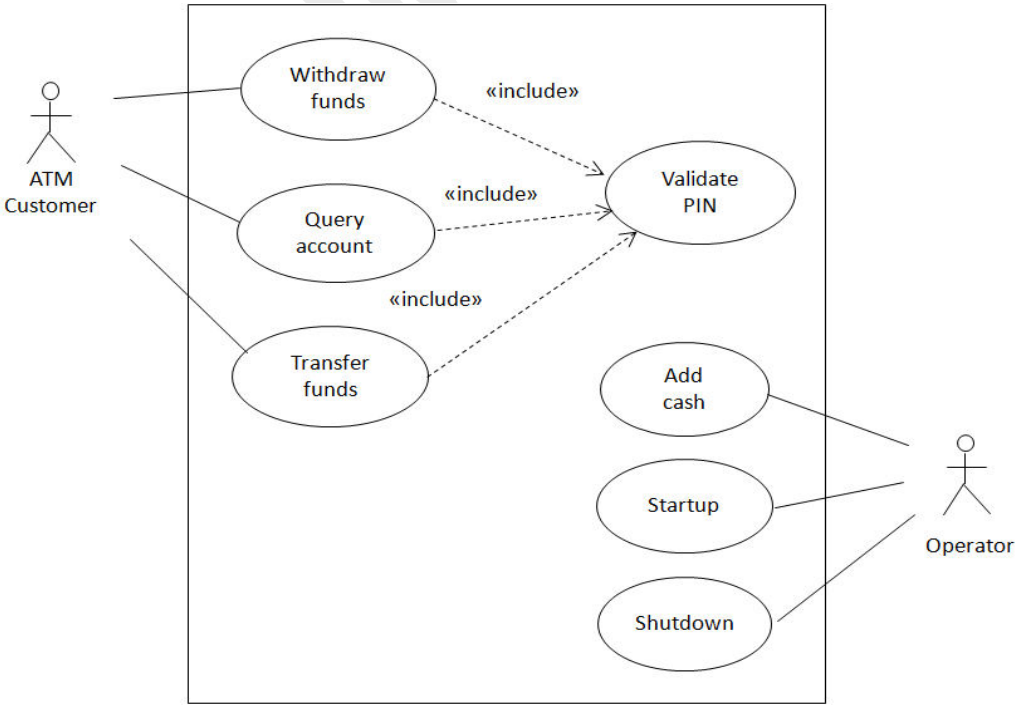
Uses Case Diagram for ATM Customer



Use Case Diagram for ATM Operator

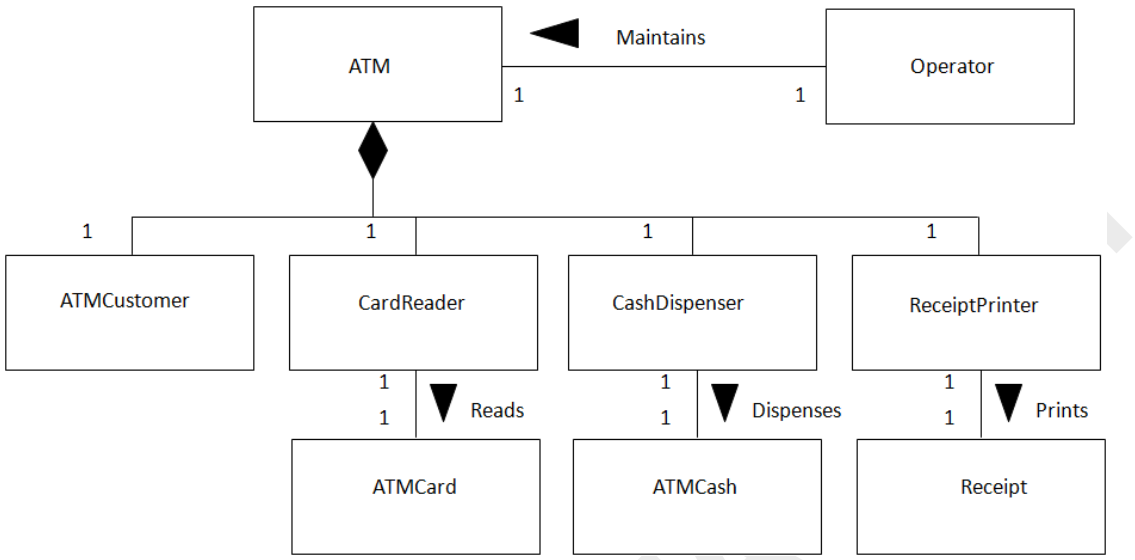


Use Case Diagram for ATM

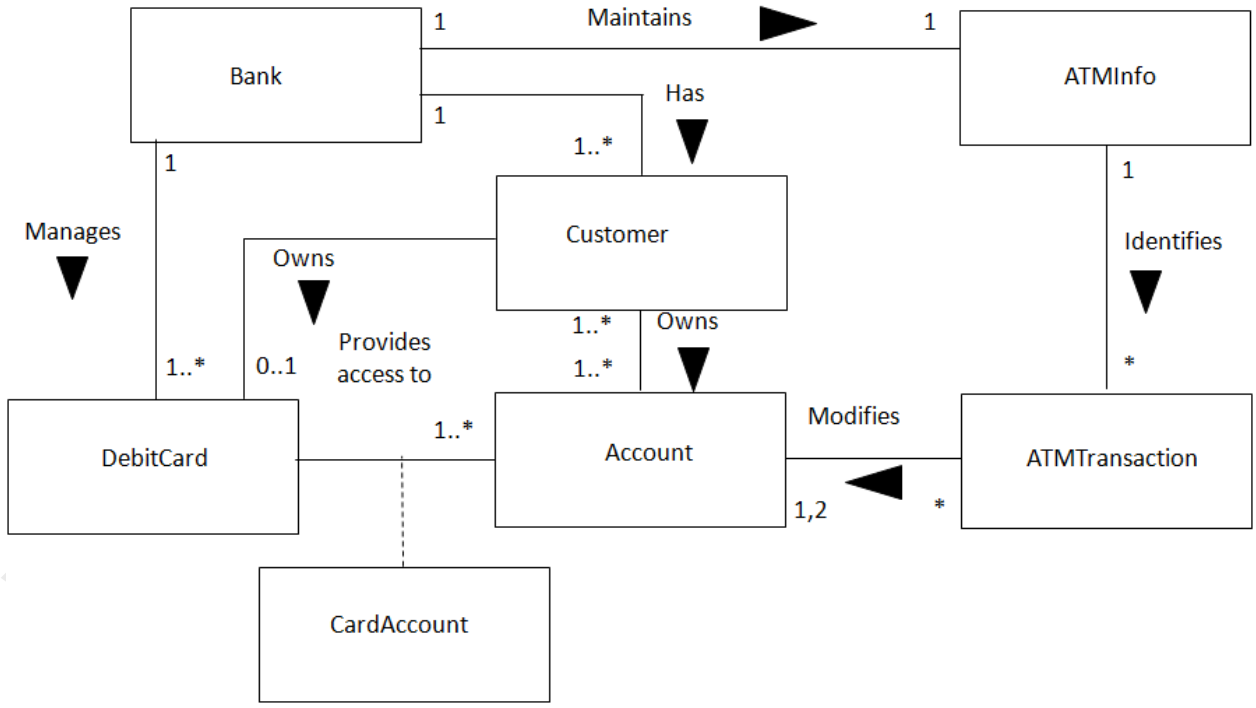


Object Model

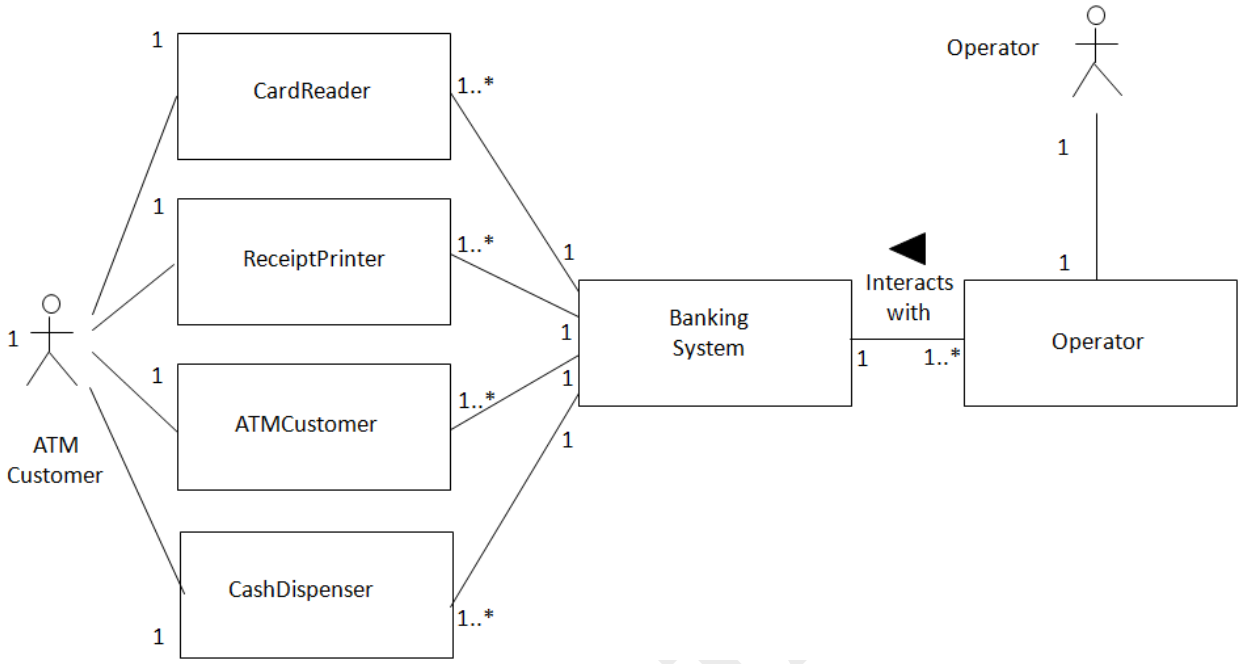
Conceptual Static Model for Problem Domain: Physical Classes



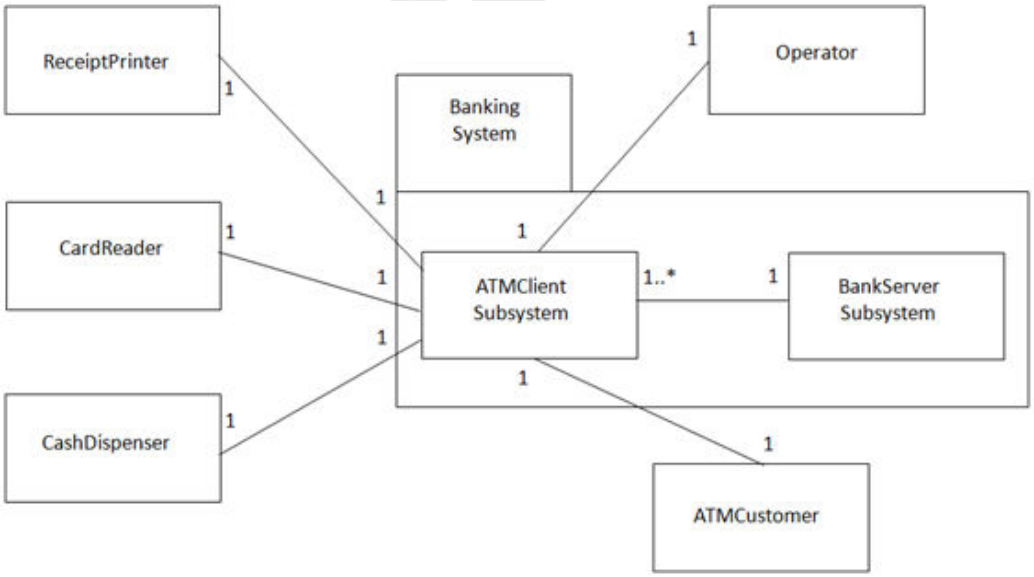
Conceptual Static Model for Problem Domain: Entity Classes



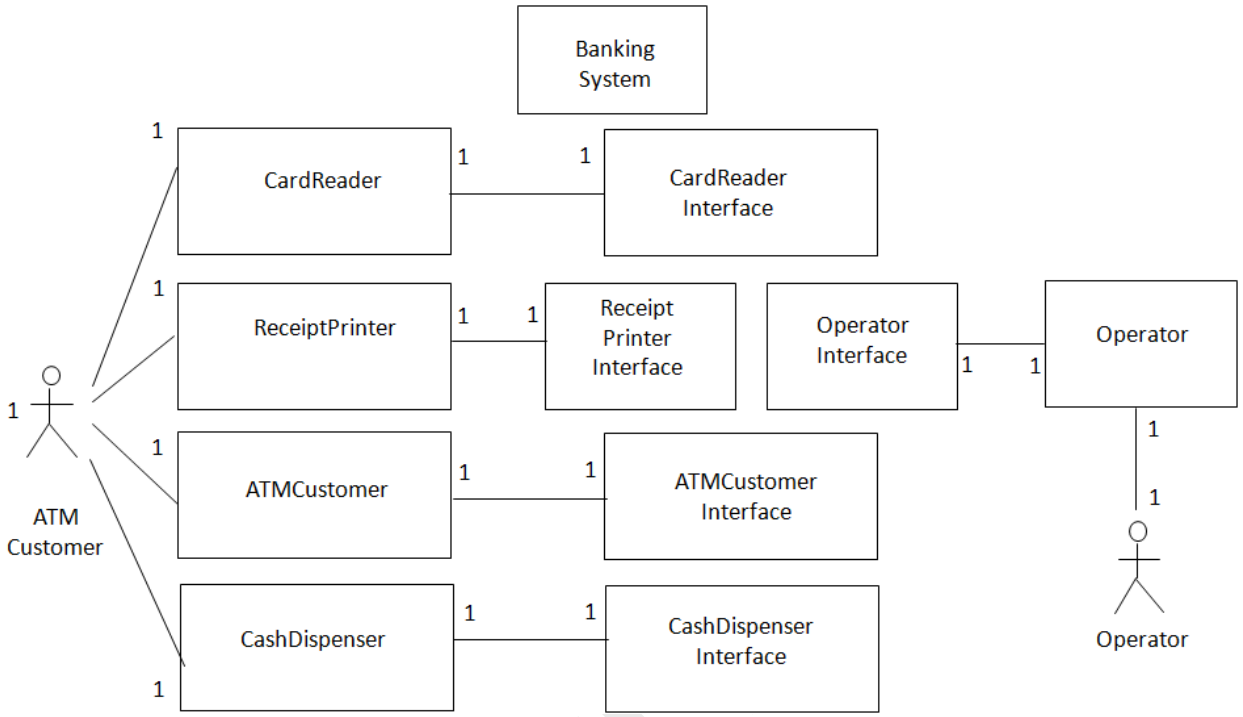
Banking System Context Class Diagram



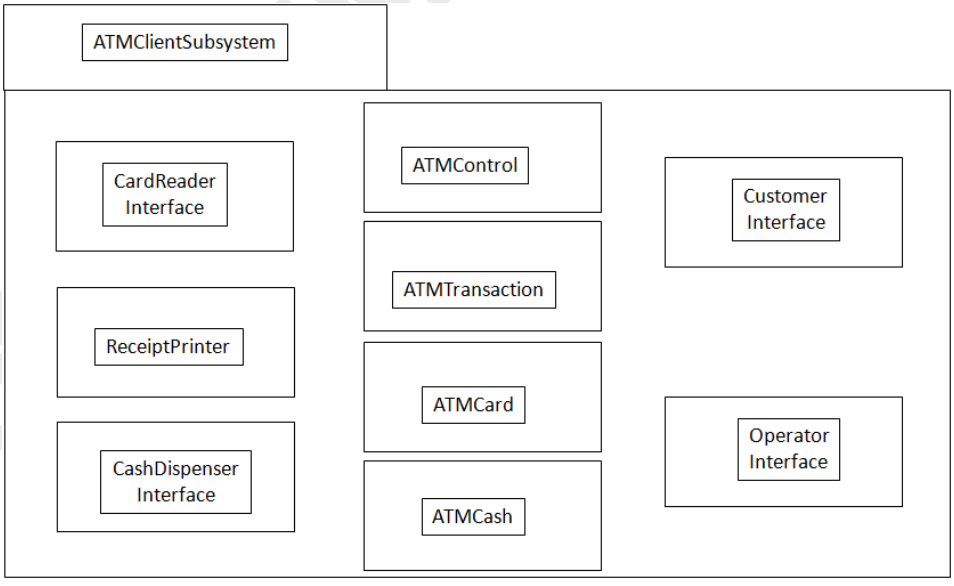
Banking System: Major Subsystems



Banking System External Classes and Interfaces Classes

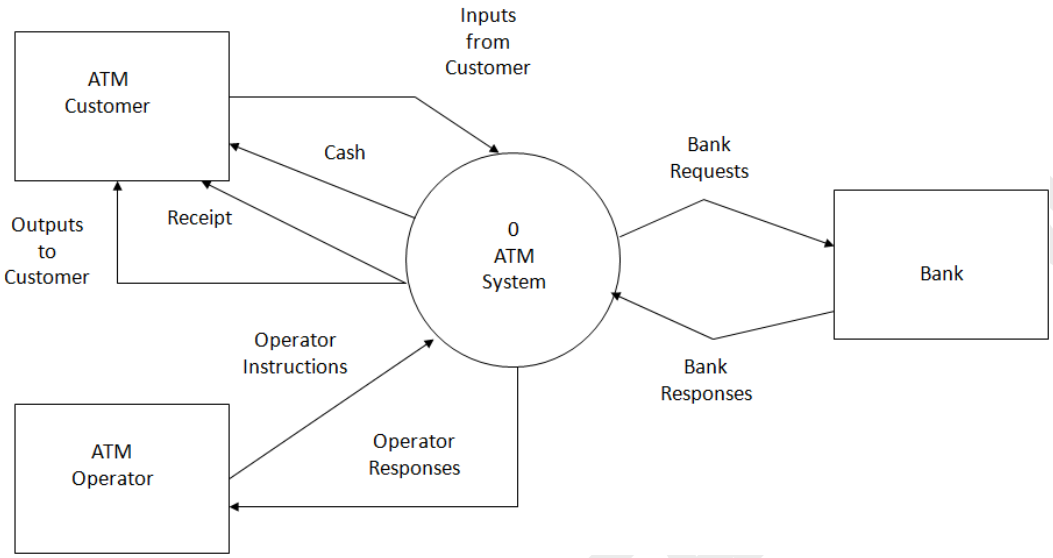


ATM Client Subsystem Classes

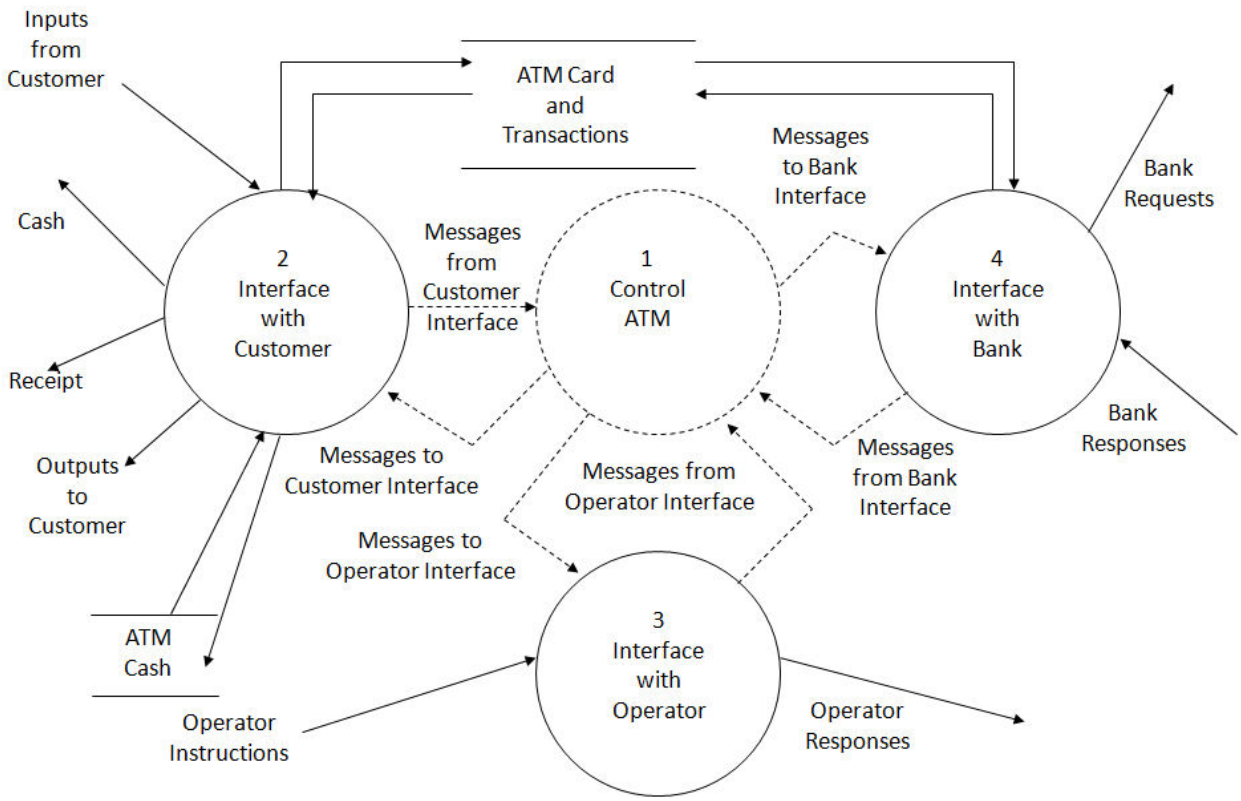


Data Flow Diagrams

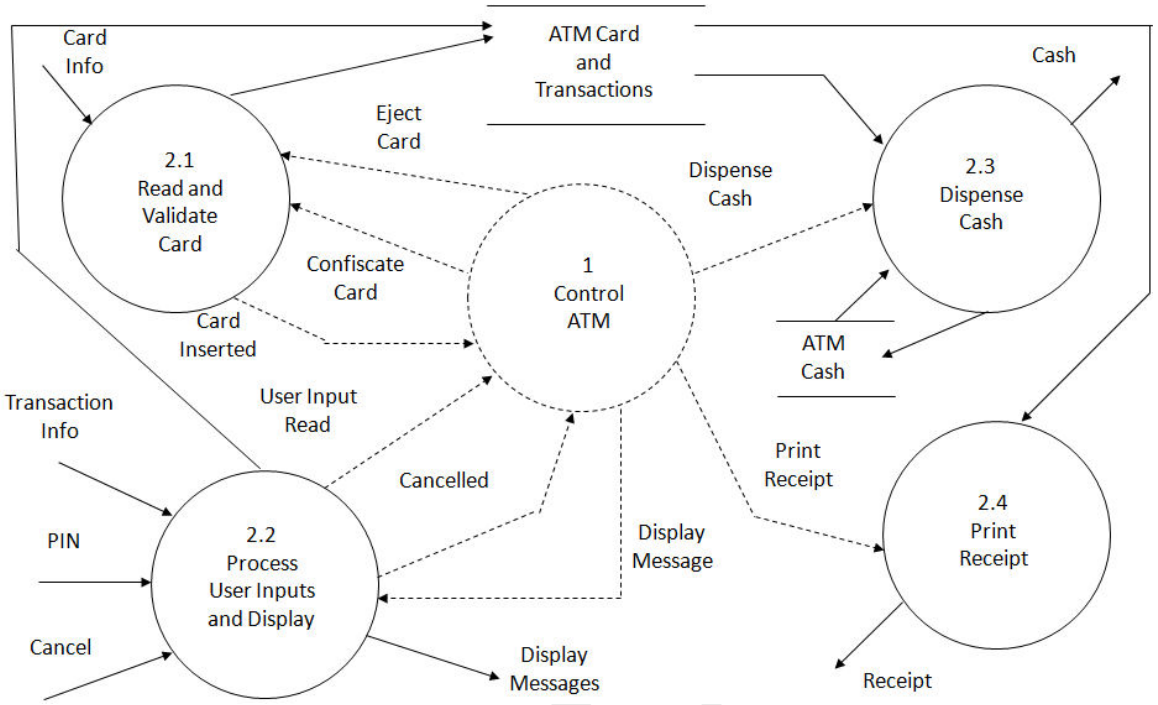
System Context Diagram



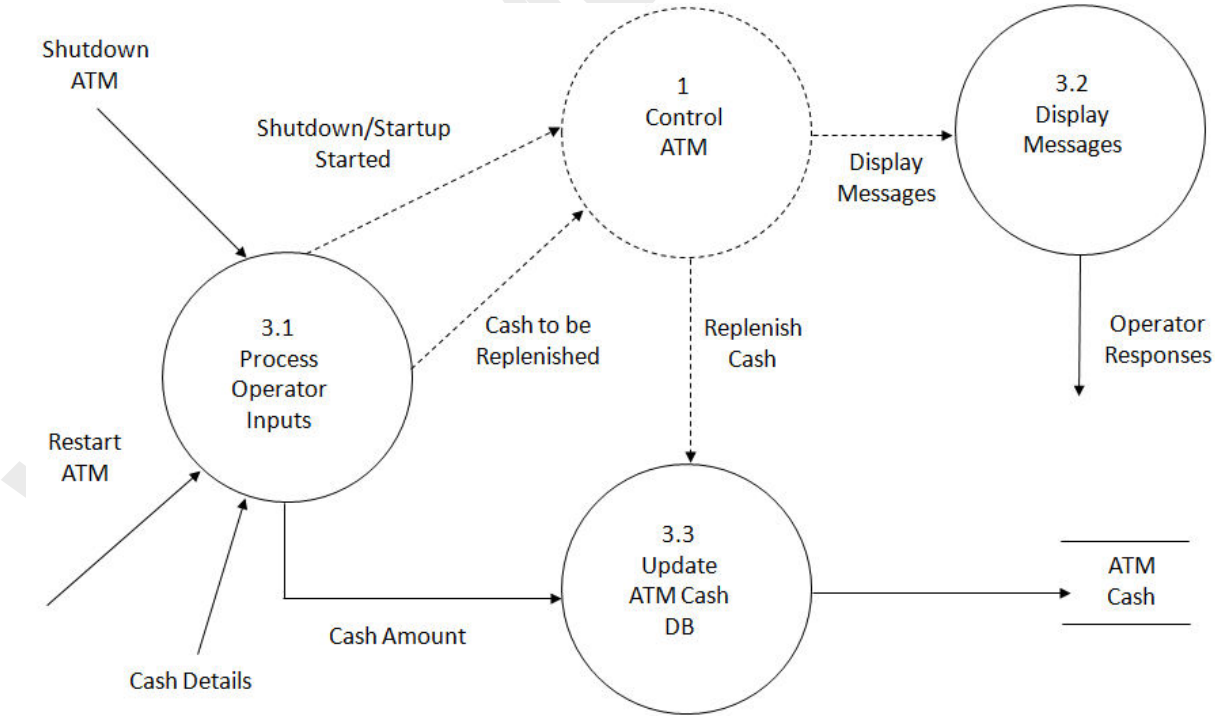
Data Flow Diagram – Level 1



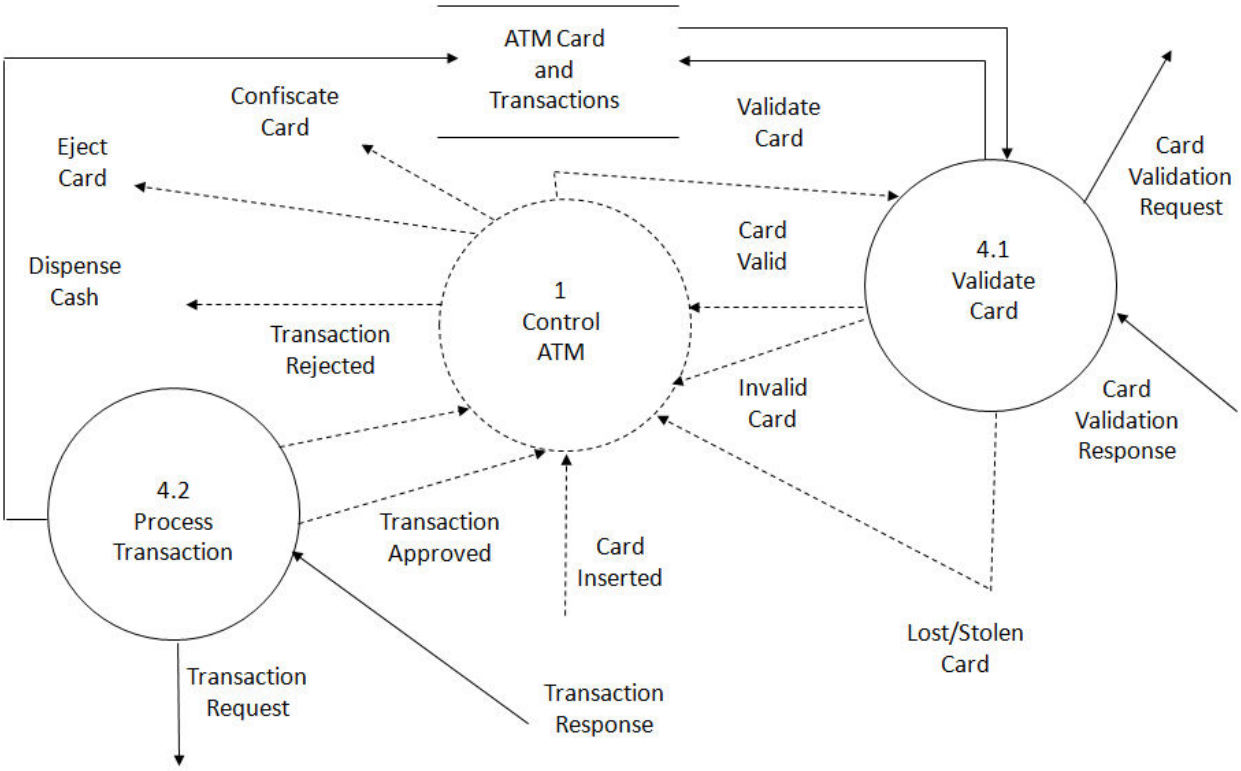
Level 2 DFD: Interface with Customer



Level 2 DFD: Interface with Operator

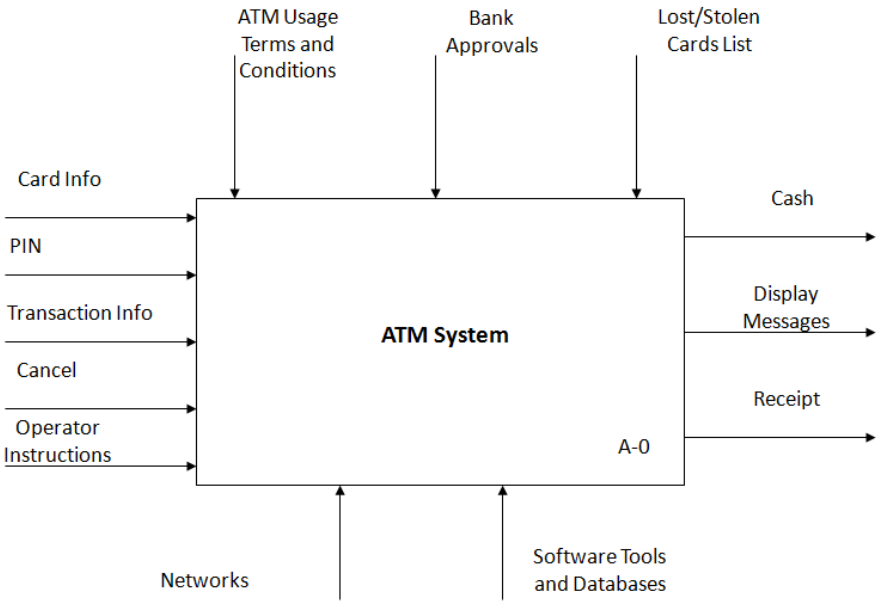


Level 2 DFD: Interface with Bank

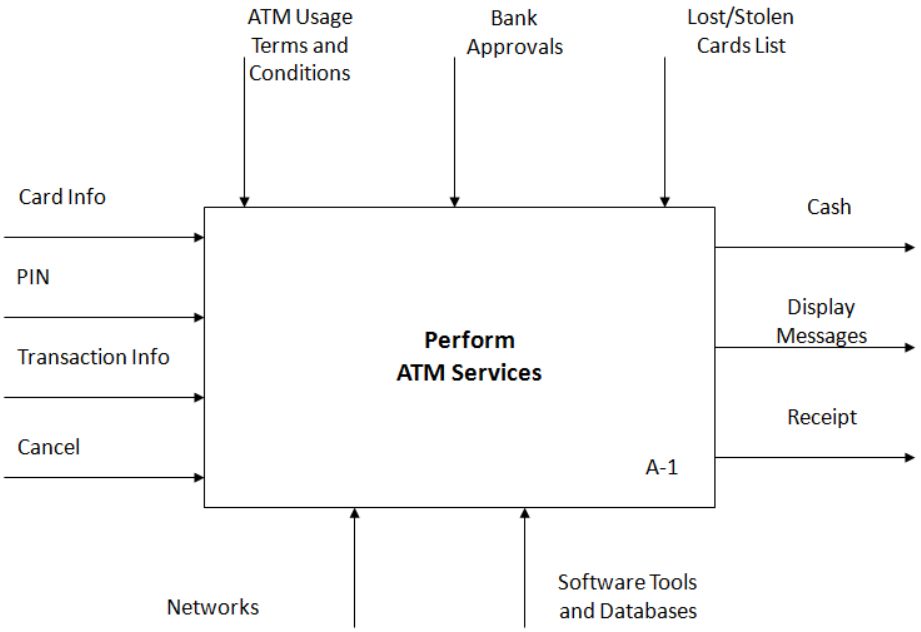


SADT Diagrams

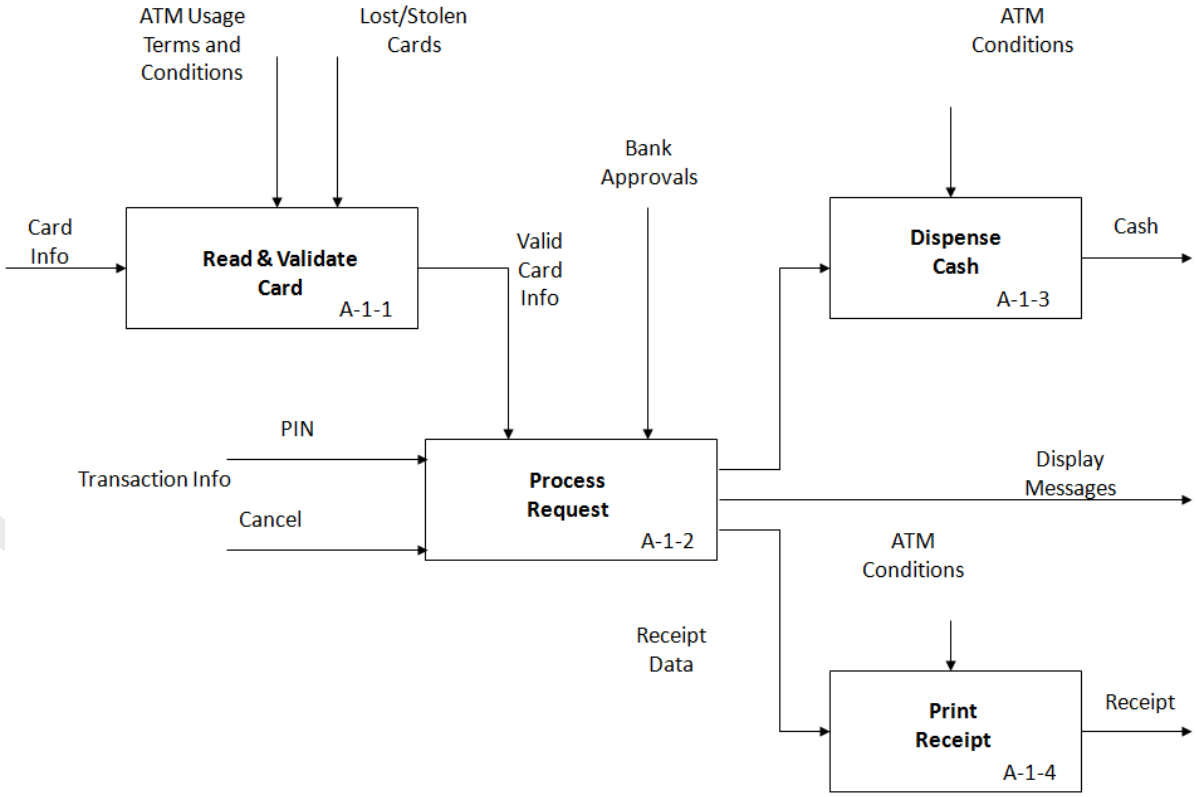
Banking System Context Diagram



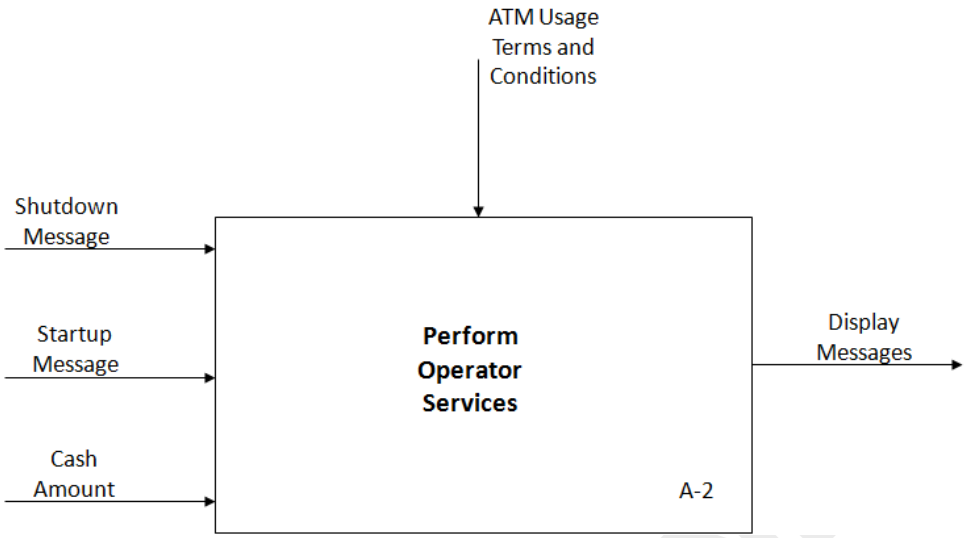
SADT Level 1 Diagram



SADT Level 2

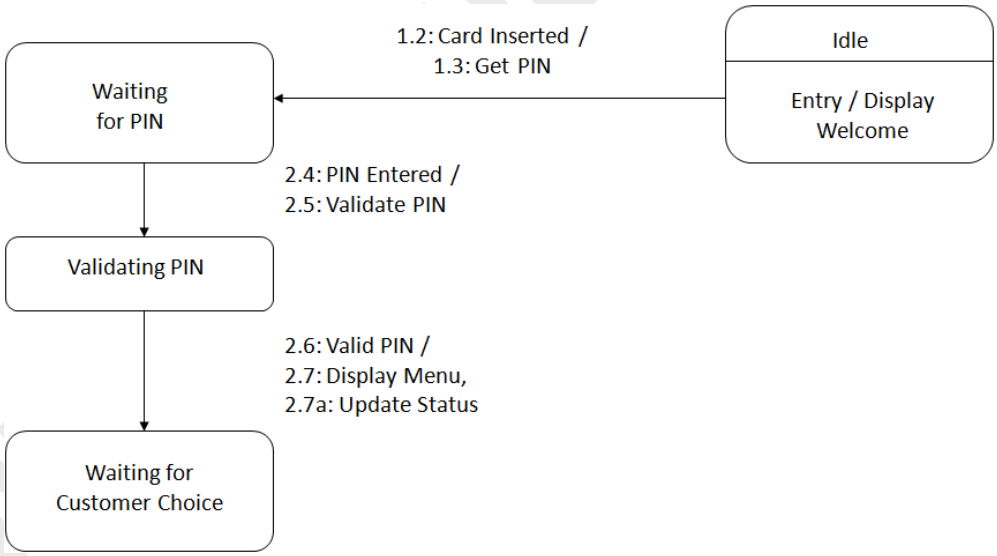


SADT Level 1 Diagram

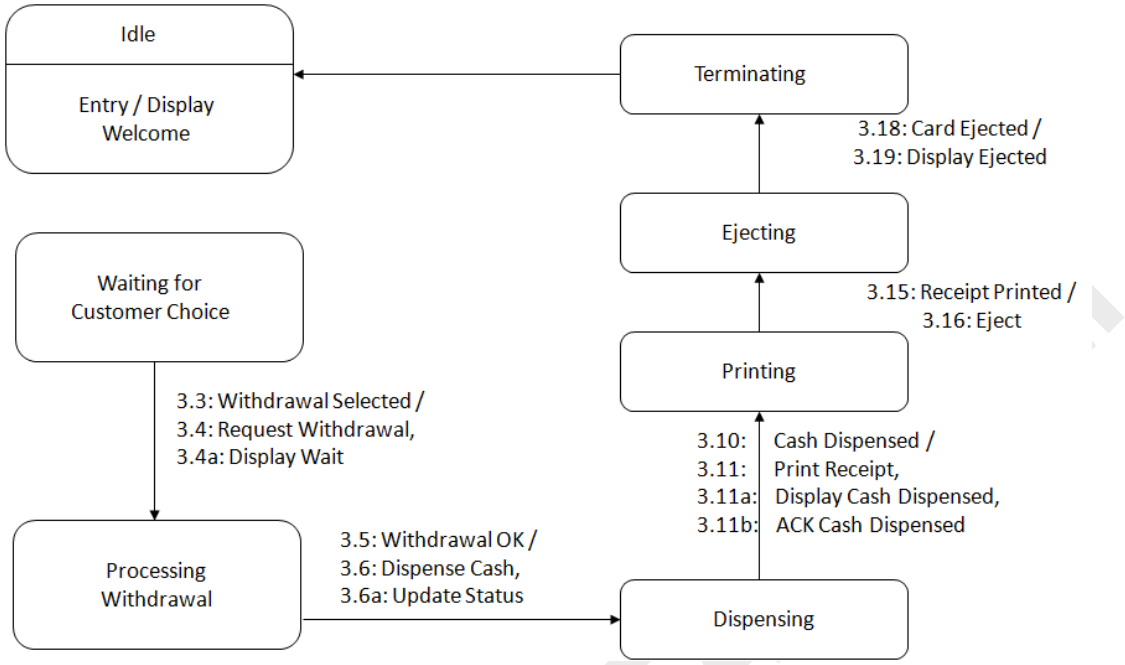


Statecharts Diagrams

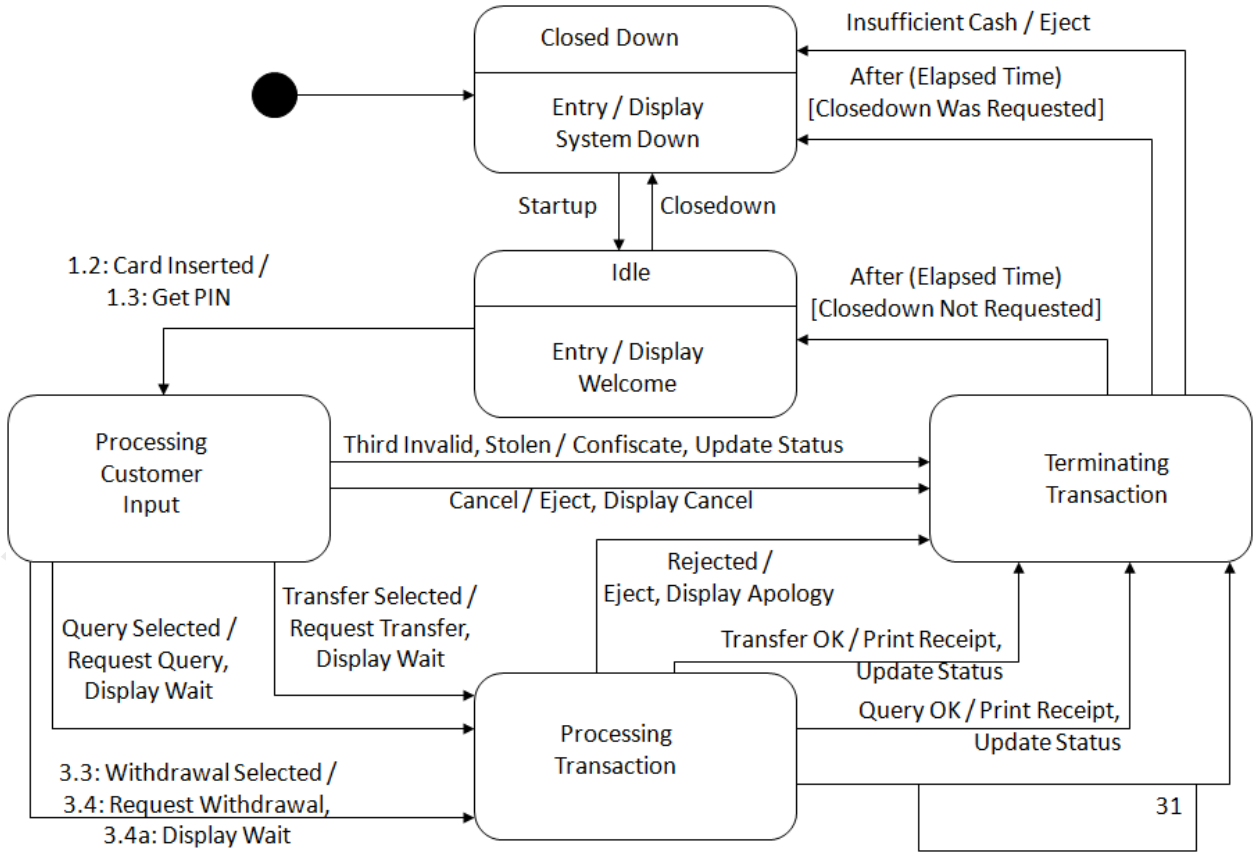
Statechart for ATM Control: Validate PIN Use Case



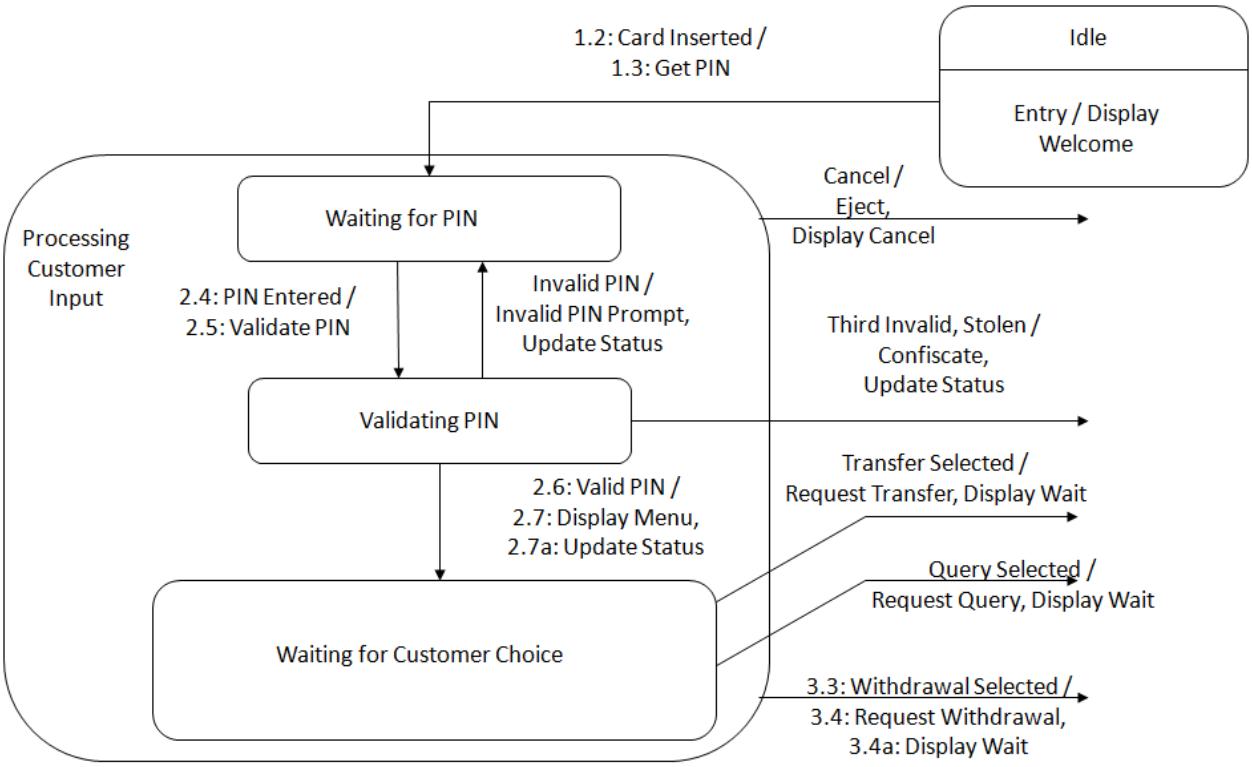
Statechart for ATM Control: Withdraw Funds Use Case



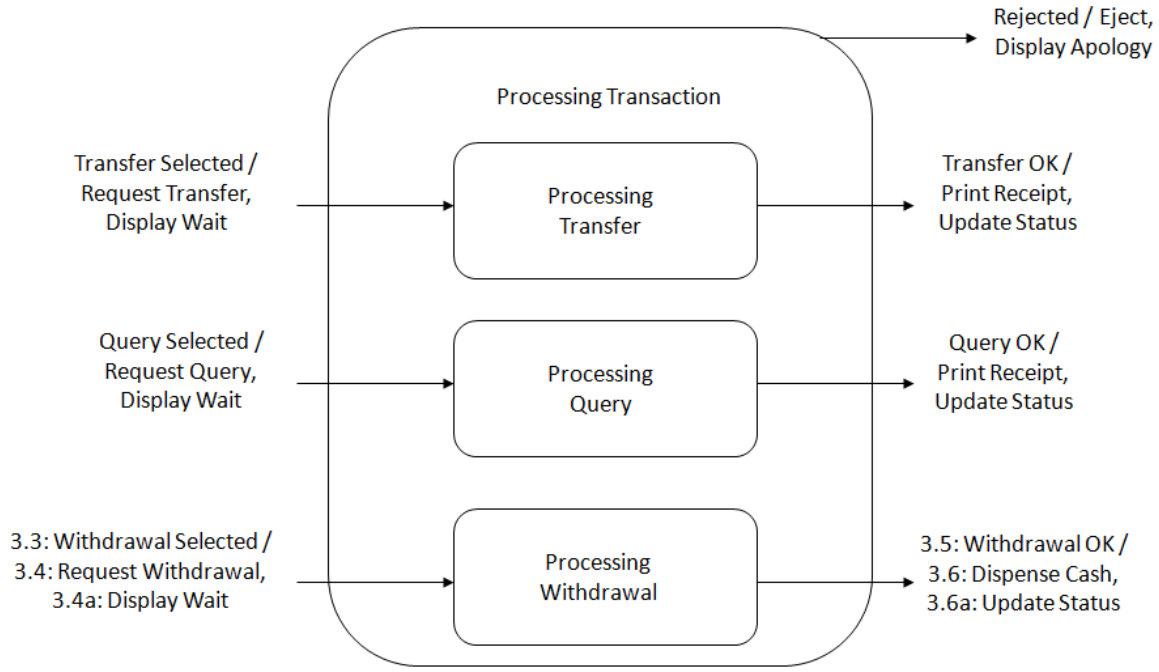
Top-Level ATM Control Statechart



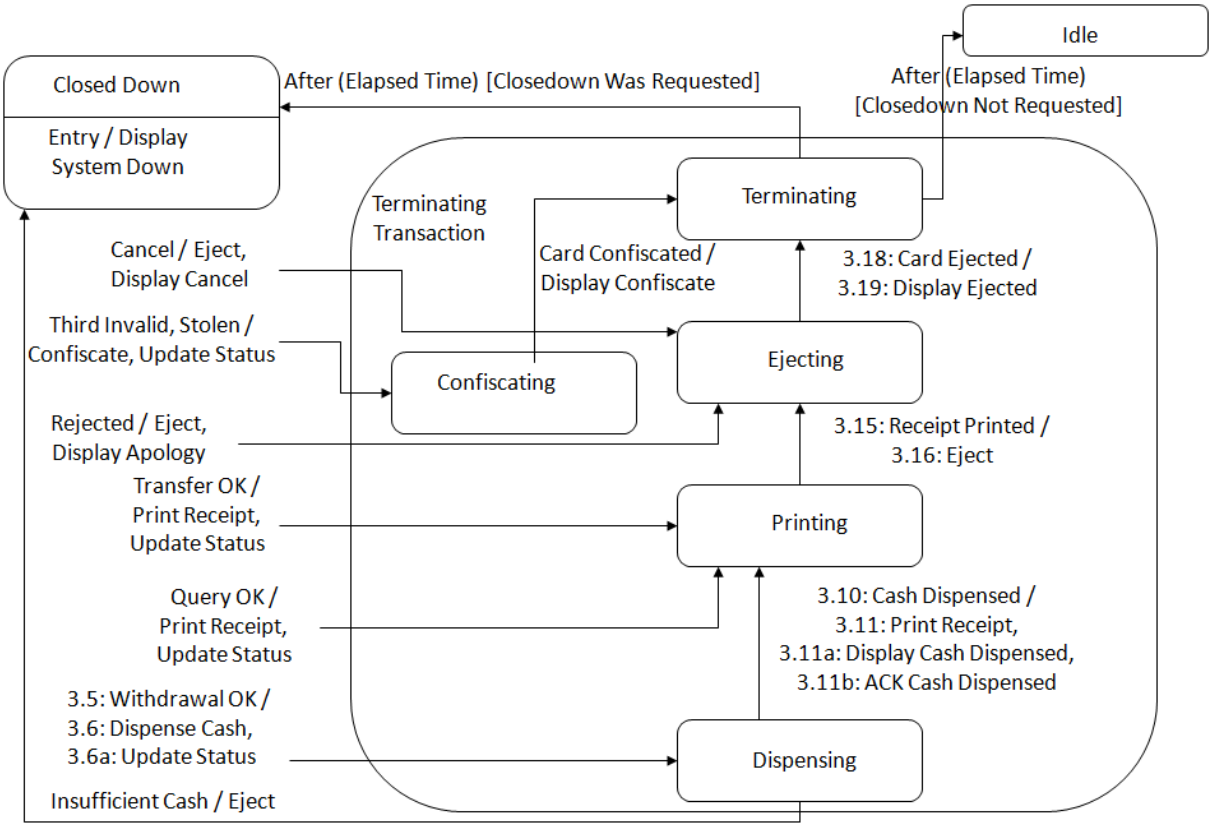
ATM Control Statechart: Processing Customer Input Superstate



ATM Control Statechart: Processing Transaction Superstate

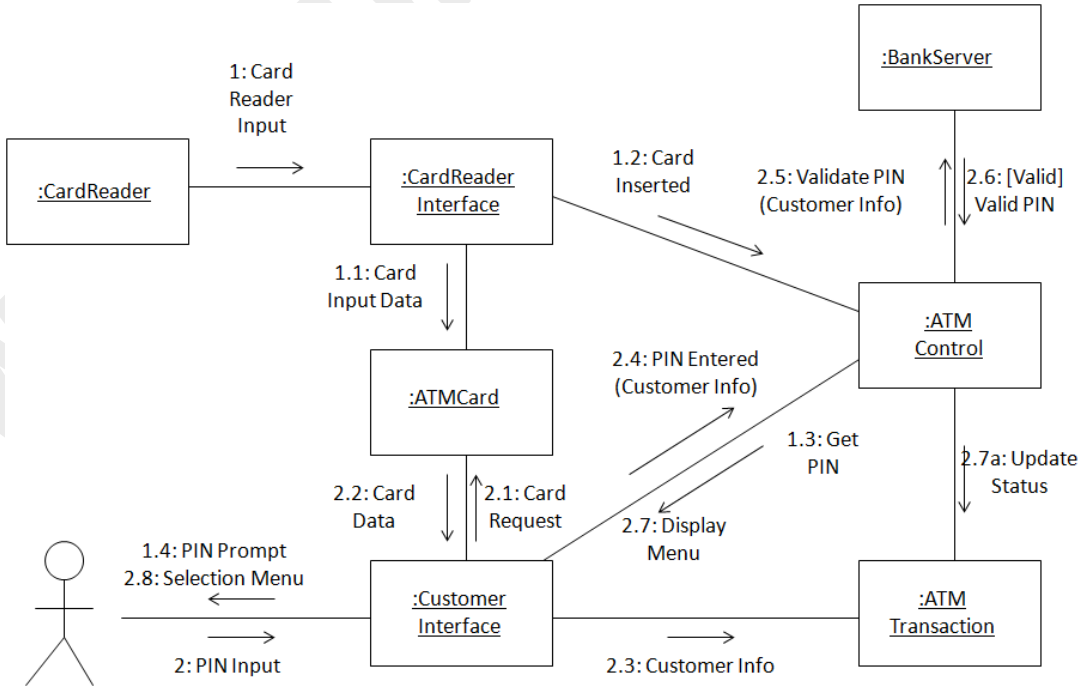


ATM Control Statechart: Terminating Transaction Superstate

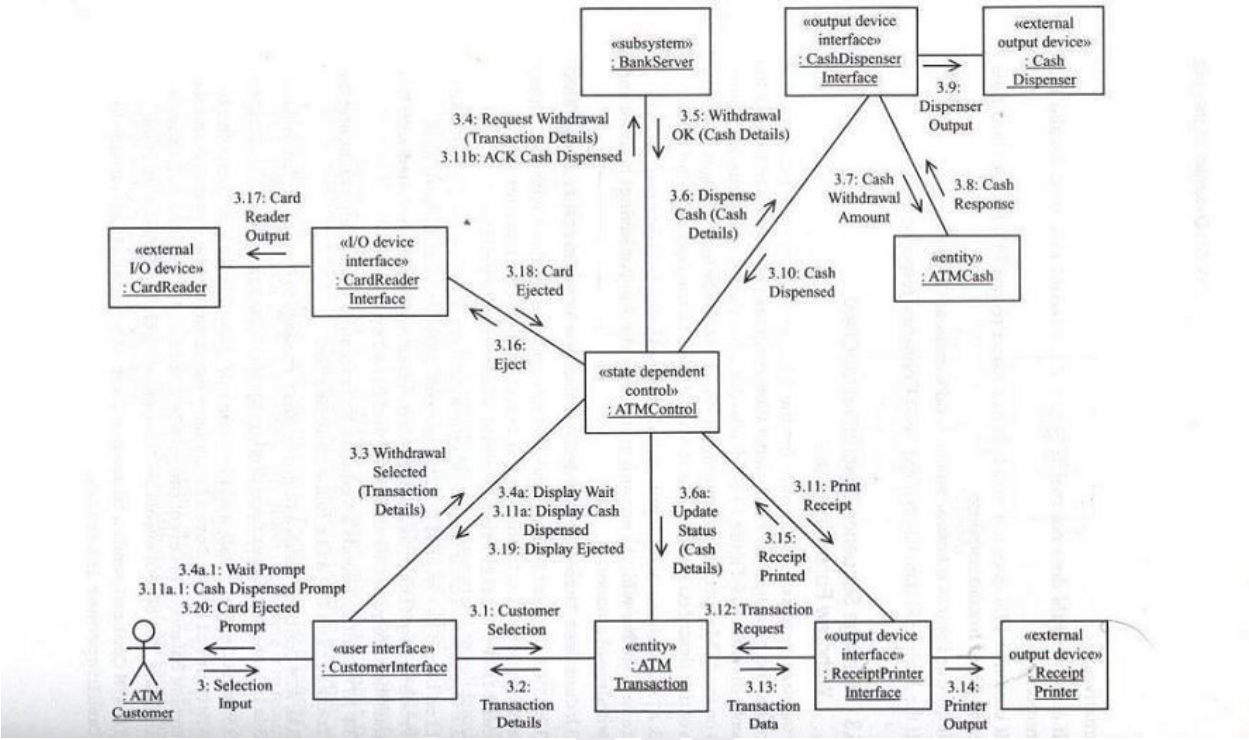


Collaboration Diagrams

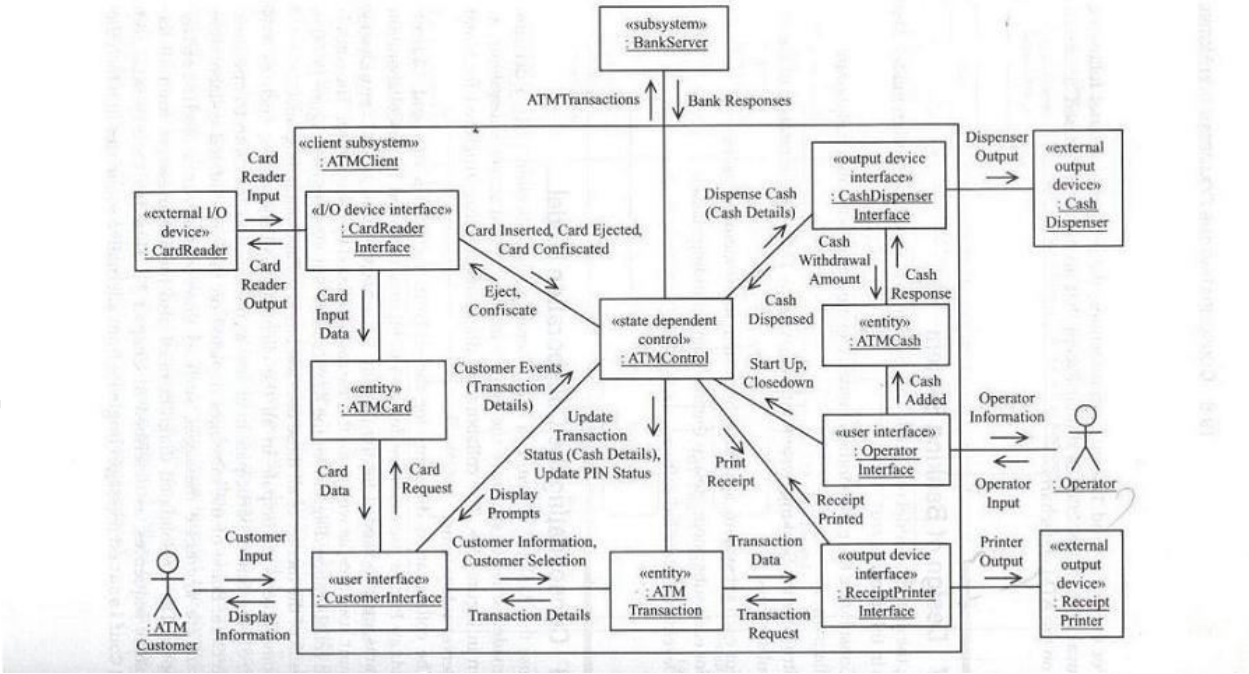
Collaboration Diagram: ATM Client Validate PIN Use Case



Collaboration Diagram: ATM Client Withdraw Funds Use Case

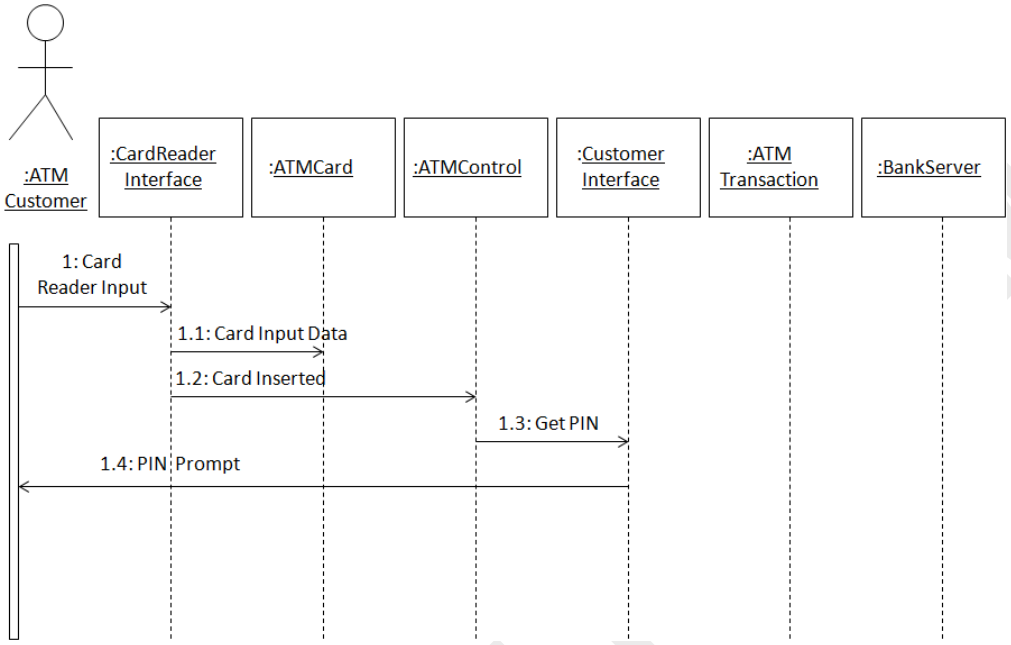


Consolidated Collaboration Diagram for ATM Client Subsystem

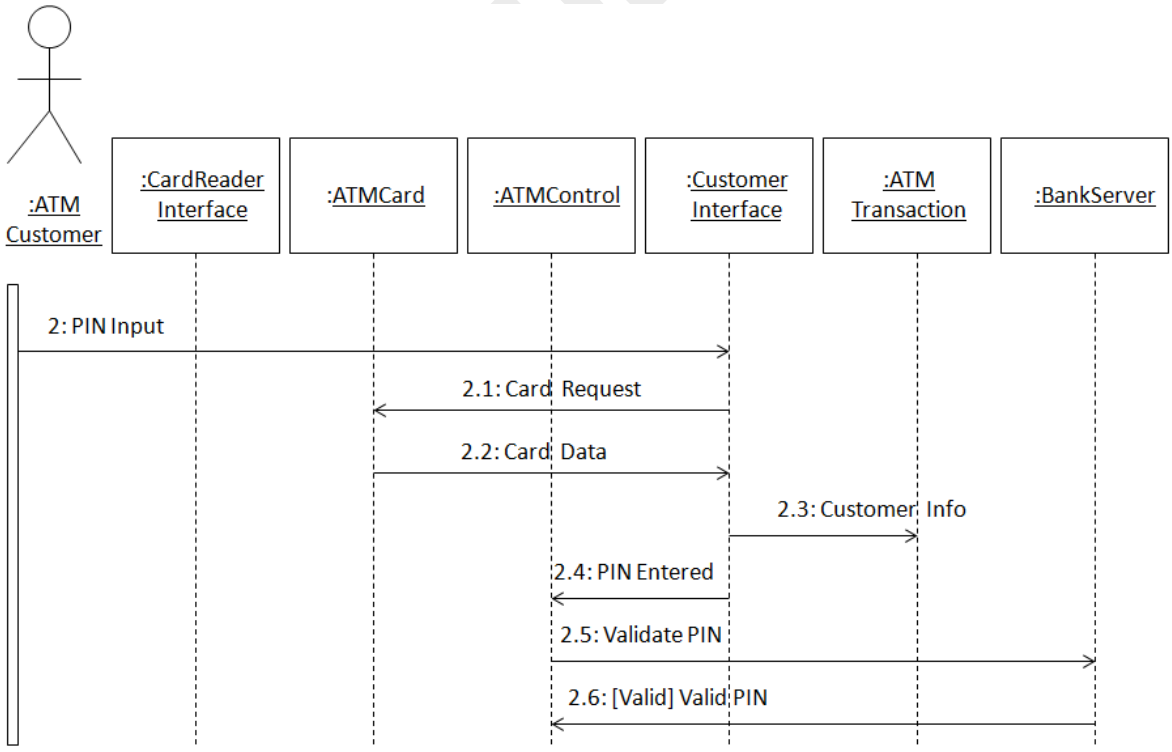


Sequence Diagram

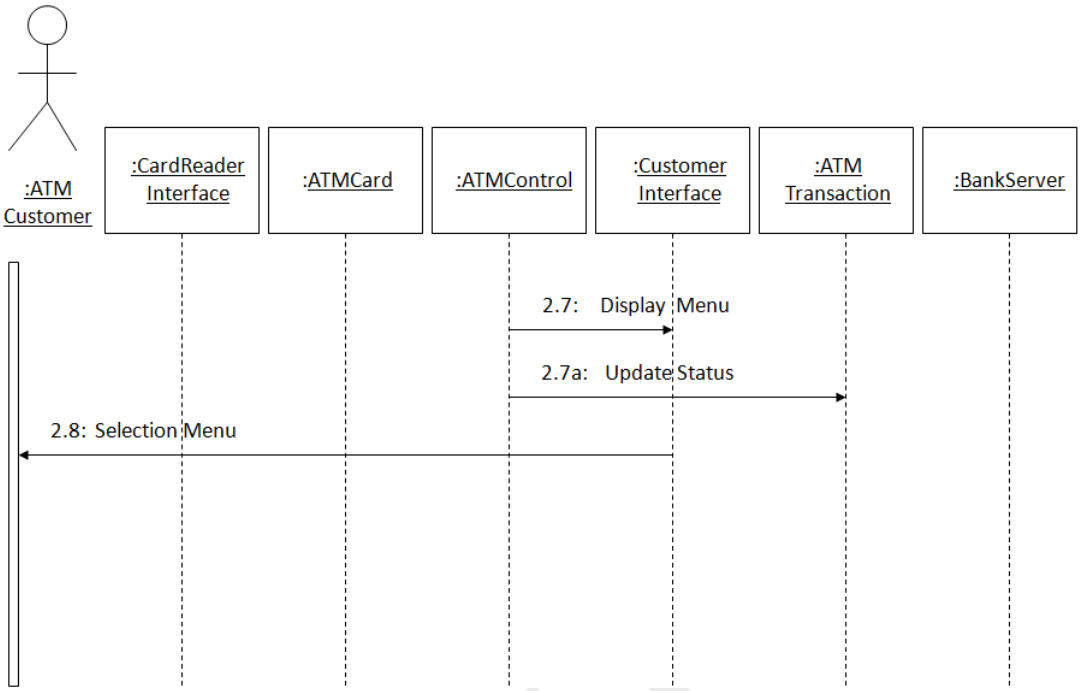
Sequence Diagram: ATM Client Validate PIN Use Case – 1



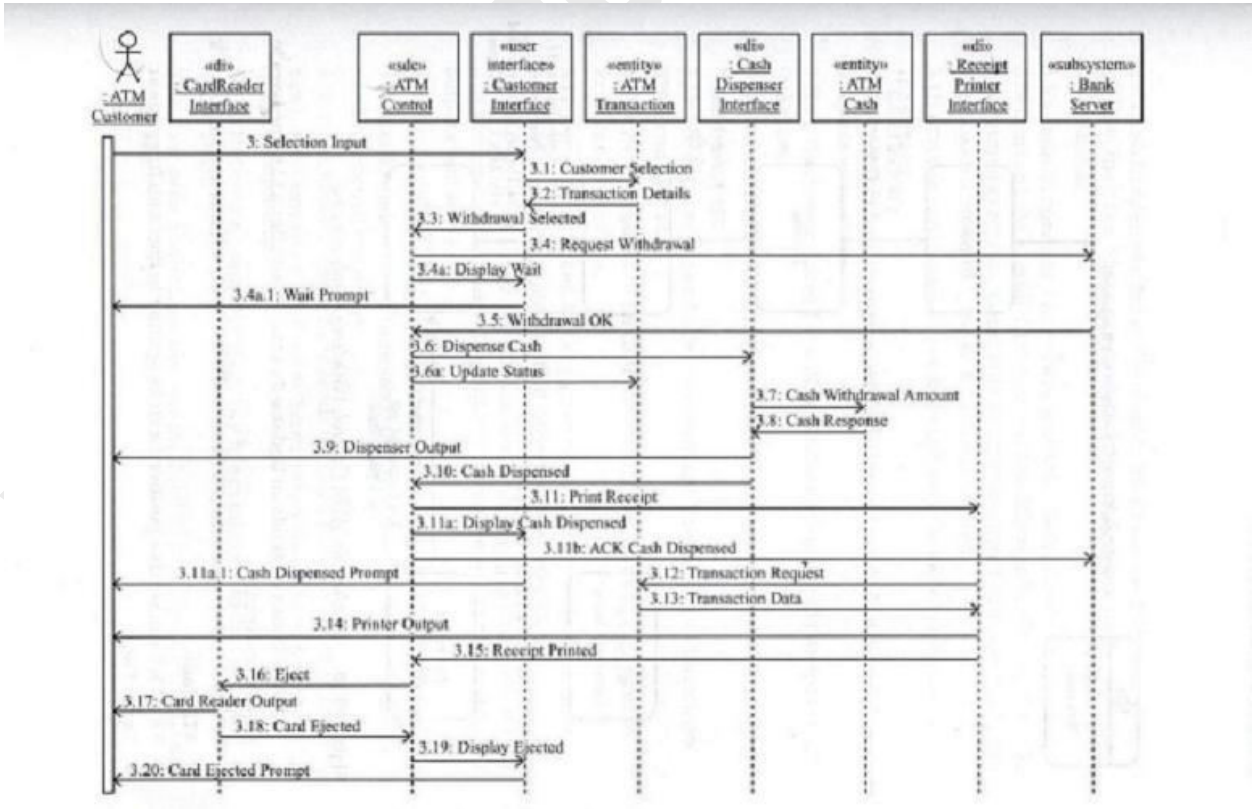
Sequence Diagram: ATM Client Validate PIN Use Case – 2



Sequence Diagram: ATM Client Validate PIN Use Case – 3



Sequence Diagram: ATM Client Withdraw Funds Use Case



Requirements Engineering for Agile Methods

Introduction

- Agile methods are a family of software development processes
- Their aim is to deliver products faster, with high quality, and satisfy customer needs through the application of the principles of lean production to software development
- Agile methods have become popular during the last few years
- Lean production has been conceived during the ’50s in Toyota. It involves several practices that are now part of most manufacturing processes, such as just-in-time development, total quality management, and continuous process improvement
- The principle of lean production is the constant identification and removal of waste, that is, anything that does not add value for the customer to the final product
- Agile methods focus on
    - ✓ Delivering value for the customer
    - ✓ Ensuring that the customer understand such value and be satisfied by the project
- Agile methods pose a lot of emphasis in producing and delivering to the customer only those features that are useful
- Producing anything that is not required is considered a mistake
- Adding a feature that is not needed not only consumes effort without adding customer value but also creates extra code, which may contain errors and make code longer and more complex to maintain, to correct and improve
- The waste includes general architectures that are used only partially or reusable components with functionalities that are likely to be never usedTo achieve such elimination of waste, agile methods claim to be
    - ✓ Adaptive rather than predictive
    - ✓ People-oriented rather than process-oriented
- A close collaboration between the development team and the customer is a must, so that
    - ✓ Requirements are fully identified and correctly understood
    - ✓ Final products reflects what the customer needs, no more no less

Agile Methods

- eXtreme Programming
- Scrum
- Dynamic Systems Development Method
- Adaptive Software Development
- The Crystal Family

Agile Manifesto

- Individuals and Interactions over Process and Tools
- Customer Collaboration over Contracts
- Working Software over Documentation
- Responding to Change over Planning

Common Practices and Behaviors

- Adaptability
- Incremental Development
- Frequent Releases
- Requirements Prioritization before every iteration
- High Customer Involvement

RE in Agile Methods

- In agile methods, the whole development team is involved in requirements elicitation and management, while in traditional approaches often only a subset of the development team is involved
- This approach is feasible only if the size of the problem is limited, as only a small development team can interact directly with the customer
- If the problem is bigger, the team can use other techniques for eliciting and managing requirements
- The understanding of requirements variability has a strong impact on the ability of agile methods to be "lean'

Focus of Agile Methods in RE

- The Customer
- Waste in Requirements
- Requirements Evolution
- Non-functional Requirements

Customer in RE of Agile Methods

- Customers/stakeholders assume a paramount role in requirements engineering in agile methods
- The interaction between the development team and stakeholders is complex due to the different perceptions of the problem that the stakeholders have
- In agile methods, the problem of multiple stakeholders is solved reducing their number to one, who represent all stakeholders in the project
- The customer should be a domain expert and able to make important decisions such as accepting the product, prioritize requirements, etc.

Customer-on-Site Requirements

- Availability
  - ✓ The customer has to be always available to answer questions coming from the development team. Any delay in the answer delays the development of the product
- Complete knowledge
  - ✓ The customer is the representative for all the stakeholders. Therefore, he is able to answer all questions, since he is the domain expert and knows how the application should work and input/output data required. Again, this is possible if the size of the project is limited
- Decision power
  - ✓ The customer is able to make final decisions and commitments. Changes in requirements, acceptance of the features implemented, etc., can be decided directly by the customer, allowing a fast decision making process

Waste in Requirements

- Identification and reduction of waste from requirements assume a paramount role to avoid the creation of waste later in the development process
- In lean practices, the reduction of waste is extremely important because waste always generates more waste
- Waste in requirements includes both wrong and useless requirements
- All the waste generated is a cost for the customer both directly and indirectly
- Such costs are likely to generate further waste inside the customer organization due to the reduced amount of money available to its core business and reduced revenues

Impact of Requirements Waste on Development Process

- More source code to write and higher cost
- Increased complexity of the source code
- Delayed delivery of the final version of the application with all functionalities
- More complex and costly maintenance
- More resources required by the application, including: memory usage, processing power, network usage, etc
- Increased complexity of the application from the point of view of the customer (e.g., more complex user interface, more effort to learn how to use the application, etc)
- Savings produced by the application in the production process of the customer are delayed

Techniques to Focus on Interaction with Customer

- The whole development team collects requirements from the customer
  - ✓ Usage of documents to share knowledge is reduced to a minimum
- Requirements are collected using a common language
  - ✓ Requirements are collected using the language of the customer, not any formal language
- Direct interaction between the development team and the customer
  - ✓ Reduces both the number of documents required and the probability of misunderstanding due to unnecessary communication layers
- Requirements splitting
  - ✓ If the development team considers a requirement too complex, this technique helps the customer to split it in simpler ones

Waste Reduction Techniques

- In order to reduce the waste created by the over specification of requirements, following techniques are used
  - ✓ Requirements prioritization
  - ✓ Incremental releases

Prioritization Activities

- The development team estimates the time required to implement each functionality. If the effort required is too high, the requirement is split into simpler ones that can be implemented with less effort
- The customer specifies business priorities for each functionality
- According to the business priorities, the development team assign a risk factor to the functionalities
- The customer and the development team identify the functionalities to implement in the iteration

Requirements Evolution

- Agile methods assume that is very hard to elicit all the requirements from the user upfront, at the beginning of a development project
- They also assume that such requirements evolve in time as the customer may change its mind or the overall technical and socio-economical environment may evolve
- Agile companies are aware that changes are inevitable and they include the management of variability into the development process
- Requirements are not well known at the beginning of the project
- Requirements change
- Making changes is not expensive
- Agile methods assume that the cost of introducing changes in a product is nearly constant over the time, but this hypothesis is not true in every context
- Usually, the cost of implementing changes grows exponentially over the time
- On the other hand, if development phases are grouped together in very short iterations and binding decisions are taken as late as possible, the growing of the cost is limited
- In order to manage requirements evolution, agile methods use variable scope-variable price contract. This means that the features really implemented into the system and its cost evolve as well
- Requirements are negotiated throughout the project b/w the customer the development team

Managing Variability

- Decoupling requirements
- Requirements elicitation and prioritization
- Decoupling requirements
  - ✓ Requirements have to be as independent as possible in order to clearly identify what to implement and make the order of their implementation irrelevant
- Requirements elicitation and prioritization
  - ✓ At the beginning of every iteration, there is a requirements collection and prioritization activity. During that, new requirements are identified and prioritized. This approach helps to identify the most important features inside the ongoing project.
- Features are implemented mainly according to their prioritization, not their functional dependence

Non-Functional Requirements

- Agile methods do not provide any widely accepted technique for eliciting and managing non-functional requirements
- Such requirements are collected implicitly during the requirements collection activity. The need of specifying non-functional requirements is less important than in other context due to the continuous interaction with the customer

Roles and Responsibilities

- Customer
- Developers
- Managers

Tools for Requirements Management in Agile Methods

- Paper, pencil, and pin boards
- UML modeling tools
- Requirements negotiation tools
- Instant messaging tools
- Project management tools

Internet & Requirement Engineering

Internet

- Internet is a network of networks
- It is a huge network providing various kinds of information in various forms
- We use internet in our daily life to send an email, to get information about a particular topic and even to chat with our friends

Web-Based Information Systems (WBIS)

- What makes web systems so different from the traditional software systems that their planning and construction requires a unique development process?
- First, we need to understand different types of web-based information systems

Types of WBIS Models

- Those that deliver advertising and promotion
- Those that assist business workflow
- Those that facilitate inter-organizational interaction
- Those that support multi-participant trading

Differences in WBIS Models

- Each kind of WBIS model emphasizes distinct aspects of site design depending on its purpose
- Some focus on supporting business to business transactions, the construction of online metaphors for business activity, and customer assistance
- Others look at promoting organizational brand, building market trust and credibility
- While some simply accentuate web contents, layout, navigation and search for organizational information

E-What?

- E-commerce
- E-banking
- E-auctioning
- E-government
- E-voting
- E-healthcare
- E-insurance
- E-everything

WBIS and Requirements Engineering

- In the richness of web design issues, many directly concern WBIS customers and thus necessarily absorb requirements engineers

Stakeholders in WBIS

- Apart from the obvious project stakeholders, such as sponsors, customers, and users, the parties involved also include content developers and copyright consultants, marketing and public relation specialists, media planner and strategists, creative and art directors, graphic designers, multimedia and interaction developers, and many others, who are not often considered by requirements engineers as having input into the specification of a traditional software system

- It is a fact that stakeholders' opinion conflict with each other
- In WBIS, these conflicts are firmly embedded not only in the needs of the software systems to be developed but rather in the business processes and objectives of online buyers and sellers, and in the constraints imposed on the system by agencies regulating the financial transactions or determining compliance with the laws of the land and international treaties

Key facts

- The scope of concerns to be considered in the earliest stages of web site construction can be significantly enlarged, due to the marketing-driven development process
- The delivery cycle for web-enabled applications is commonly very short, i.e., 3 months, which leaves very little time for any formal requirements gathering and their consolidation
- Current requirements engineering practices for WBIS projects are inadequate, failing requirements engineers in identification and characterization of the potential system users, their needs and preferences, and the features required of the web systems under development
- There is need to emphasize on the stakeholder views and opinions on requirements evolution in web development
- A stakeholder in this context is considered to be any individual, group, or organization whose actions can influence or be influenced by the development and use of the system whether directly or indirectly

RE Approaches for Development of WBIS

- Web engineering
- Relationship management methodology
- Internet commerce development methodology
- Web information systems development methodology

Web Engineering

- Web development should be recognized as a process with all its structure and complexity
- Most of the difficulties with respect to development of large web sites can be attributed to a lack of suitable process models for the project teams to follow, suitable architecture, or a project model for the development of web-enabled application
- One of the most significant points, as a new and emerging trend associated with the development and evolution of web-enabled services, is the acknowledgement of the importance for project teams to improve by *learning through experience*

Relationship Management Methodology (RMM)

- RMM was introduced as a methodology for the development of hypermedia systems
- While acknowledging the importance of requirements analysis, RMM sheds little light on its mechanism

Internet Commerce Development Methodology (ICDM)

- ICDM combines the elements of business analysis as well as system development
- Internet commerce is one of those fields, that necessitate intense business activity as part of their systems development, and thus it requires a thorough analysis of its place in the overall business strategy
- Customers and suppliers (users of the systems) are encouraged to be involved at various stages of the e-Business operations, and participate in periodic reviews
- The two commonly used requirements gathering techniques in ICDM are:
  - ✓ Brainstorming
  - ✓ Group Requirements Sessions (GRS)
- Although clearly acknowledging the importance of stakeholder issues and "learning from experience" in requirements establishment for WBIS, ICDM is not prescriptive as to the use of any specific model or a process where these issues could be addressed

Web Information Systems Development Methodology (WISDM)

- WISDM is employed with the aim for evaluating the effectiveness of a pre-web methodology to the web-based methodology
- RAD and prototyping are effective approaches for WISDM project development

Non-Functional Requirements

- Special focus on non-functional requirements
  - ✓ Security
  - ✓ Efficiency
  - ✓ Scalability
  - ✓ User-interface design

Requirements Engineering for Product Lines

Introduction

- The interest in software product lines emerged from the field of software reuse when developers realized that they could obtain much greater reuse benefits by reusing software architectures instead of reusing individual software components

Software Reuse

- Software reuse has been a goal in software engineering since 1968
- Goal is to maximize productivity, minimize cost of developing software and to improve profits for the software vendors
- This has been an research area in software engineering

Software Reuse Manifestos

- Code reuse
  - ✓ Mathematical libraries
- Design reuse
- Requirements reuse
- Knowledge reuse
- Product line approach

Software Family of Systems

- Parnas referred to a collection of systems that share common characteristics as a family of systems. It is worth considering the development of a family of systems when there is more to be gained by analyzing the systems collectively rather than separately – that is, when the systems have more features in common than features that distinguish them

Software Product Lines

- A software product line consists of a family of software systems that have some common functionality and some variable functionality
- To take advantage of the common functionality, reusable assets are developed
- Reusable assets include: requirements, designs, components, and so on
- These reusable assets can be reused by different members of the family
- The software industry is increasingly recognizing the strategic importance of software product lines
- Product lines are not new and they have been used in ancient era also
- Egyptian pyramids are an example of product lines

Airline Product Lines

- A modern example of product lines comes from the airline industry, with the European Airbus A-318, A-319, A-320, A-321 airplanes, which share common product features, including jet engines, navigation equipment, and communication equipment

Software Product Lines

- The traditional mode of software development is to develop single systems - that is, to develop each system individually
- For software product lines, the development approach is broadened to consider a family of software systems
- A software product line is also referred to as a software product family, a family of systems, or an application domain
- The terms used most in the early days of this field were domain analysis and domain engineering
- The architectures developed for application domains were referred to as domain-specific software architectures

Software Development Approach for Product Lines

- This approach involves analyzing what features (functional requirements) of the software family are common, what features are optional, and what features are optional, and what features are alternatives
- After the feature analysis, the goal is to design a software architecture for the product line, which has common components (required by all members of the family), optional components (required by only some members of the family), and variant components (different versions of which are required by different members of the family)
- To model and design families of systems, the analysis and design families of systems, the analysis and design concepts for single-product systems need to be extended to support software product lines

Clements and Northrop

- "A set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"

Activities in Software Product Lines

- Modeling commonality and variability is an important activity in developing software product line
- There have been a number of product line techniques

Early Product Line Approaches

- Family-Oriented Abstraction, Specification, and Translation
- Work at Software Engineering Institute
- Work at Software Productivity Consortium
- Evolutionary Development Life Cycle

Variability in Software Product Lines

- Variability in a software product line can be modeled at the software requirements level and at the software design level
- The most widely used approach for modeling variability in requirements is through feature modeling

The Concept of a Feature

- Features are an important concept in software product lines because they represent reusable requirements or characteristics of a product line
- The concept of a feature is quite intuitive and applies to all product lines, not just software product lines

Example of a Product Line

- A vehicle is a product line
- Common features
- Alternative features
- Optional features

Feature Modeling

- The FODA (feature-oriented domain analysis method uses features, which are organized into a feature tree. Features may be mandatory, optional, or mutually exclusive
- Other product line methods also used features

Common Features

- Common features are those features that must be provided by every member of the software product line
- They are also referred to as mandatory, necessary, or kernel features

Optional Features

- Optional features are those features that need to be provided by only some members of the product line
- Optional features can assume that common features are provided and so can depend on the presence of the common features

Alternative Features

- Two or more features may be alternatives to each other, where only one of them can be provided in any given member of the software product line
- Thus the features are mutually exclusive

Feature Groups

- Related features can be grouped into feature groups, which place a constraint on how the features are used by a given member of the software product line

Mutually Exclusive Features

- A feature can be an alternative – that is, one of a group of mutually exclusive features. In some product families, one of the alternatives must always be chosen for a given product family member
- In other cases, however, selecting one of the alternatives is optional for a given family member
- A zero-or-one-of feature group handles the case in which selecting one of the alternatives is optional

Exactly-One-Of Features

- In an exactly-one-of- feature group, a feature must always be selected from the group. Unlike the mutually exclusive case, the option of not selecting a feature does not exist
- This group is also known as one-and-only-one-of feature group
- It is possible for such a feature group to have a default feature and to have a prerequisite
- The feature group must contain at least two features, one of which could be a default feature

At-Least-One-Of Features

- In an at-least-one-of group, one or more optional features can be selected from the group, but at least one feature must be selected
- It is also possible for this feature group to have a default feature and to have a prerequisite
- The feature group must contain at least two features, one of which could be a default feature

Zero-or-More-Of Features

- At first, a zero-or-more-of feature group might seem unnecessary, since zero or more of means that all features in the group are optional and there are no constraints on feature selection from within the group
- However, sometimes it is useful to group optional features together because of dependency
- For example, all features in the group depend on another feature or feature group
- There is usually no default feature in this case

Requirements Prioritization

The Decision Making Process

- In everyday life, we make decisions, e.g., when buying a DVD-player, food, a telephone, etc
- Usually, we do not have more than a couple of choices to consider. Even with just a couple of choices, decisions can be difficult to make

Need for Prioritization

- When having tens, hundreds or even thousands of alternatives, decision-making becomes much more difficult
- One of the keys to making the right decision is to prioritize between different alternatives. It is often not obvious which choice is better, because several aspects must be taken into consideration

Example

- Buying a car
- One aspect or several aspects

Requirements Prioritization

- When developing software systems, similar trade-offs must be made
- The functionality that is most important for the customers might not be as important when other aspects (e.g. price) are factored in
- We need to develop the functionality that is most desired by the customers, as well as least risky, least costly, and so forth
- The quality of software products is often determined by the ability to satisfy the needs of the customers and users
- So, it is very important to include those requirements in the product, which are really needed by the customers
- Most software projects have more candidate requirements than can be realized within the time and cost constraints
- Prioritization helps to identify the most valuable requirements from this set by distinguishing the critical few from the trivial many
- Act of giving precedence or priority to one item over another item
- Requirements prioritization means giving precedence to some requirements over other requirements based on feedback from system stakeholders

Benefits of Requirements Prioritization

- Stakeholders can decide on the core requirements for the system
- Planning and selection of ordered, optimal set of software requirements for implementation in successive releases
- Helps in trade-offs of conflicting constraints such as schedule, budget, resources, time to market, and quality
- Balances the business benefit of each requirement against its cost

- Balances the implications of requirements on the software architecture and future evolution of the product and its associated cost
- Selects only a subset of the requirements and still produce a system that will satisfy the customers
- Get technical advantage and optimize market opportunity
- Minimize rework and schedule slippage (plan stability)
- Handle contradictory requirements, focus the negotiation process, and resolve disagreements between stakeholders
- Establish relative importance of each requirement to provide the greatest value at the lowest cost

Requirements Prioritization (Continued)**:**

- Prioritization can be done arbitrarily, as we discussed in the requirements negotiations
- Prioritization should be based on certain criteria like objectives, risks, quality factors, or viewpoints of stakeholders
- "The challenge is to select the 'right' requirements out of a given superset of candidate requirements so that all the different key interests, technical constraints and preferences of the critical stakeholders are fulfilled and the overall business value of the product is maximized"
  - ✓ Ruhe et. al.

Prioritization Process

- Prioritization is an iterative process and might be performed at different abstraction levels and with different information in different phases during the software lifecycle
- Prioritization techniques can roughly be divided into two categories:
  - ✓ Methods
  - ✓ Negotiation approaches

Prioritization Methods

- The methods are based on quantitatively assigning values to different aspects of requirements
- Quantitative methods make it easier to aggregate different decision variables into an overall assessment and lead to faster decisions

Negotiation Approaches

- Negotiation approaches focus on giving priorities to requirements by reaching agreement between different stakeholders
- They are based on subjective measures and are commonly used when analyses are contextual and when decision variables are strongly interrelated

Important Considerations

- In addition, one must be mindful of the social nature of prioritization. There is more to requirements prioritization than simply asking stakeholders about priorities

- Stakeholders play roles and should act according to the goals of that roles, but they are also individuals with personalities and personal agendas, and there are organizational issues also
- These issues must be addressed while prioritizing requirements

Aspects of Prioritization

- Requirements can be prioritized taking many different aspects into account
- An aspect is a property or attribute of a project and its requirements that can be used to prioritize requirements
- Often aspects interact and changes in one aspect could result in an impact on another aspects
- Importance
  - ✓ The stakeholders should prioritize which requirements are most important for the system. Importance is multifaceted, and could be urgency of implementation, importance for product architecture, strategic importance
- Penalty
  - ✓ It is possible to evaluate the penalty that is introduced if a requirement is not fulfilled. Penalty is not just the opposite of importance
- Cost
  - ✓ The implementation cost is usually estimated by the developing organization. Measures that influence cost include: complexity of the requirement, the ability to reuse existing code, the amount of testing and documentation needed
  - ✓ Cost is often expressed in terms of staff hours
- Time
  - ✓ Time is influenced by many other factors such as degree of parallelism in development, training needs, need to develop support infrastructure, complete industry standards
- Risk
  - ✓ Every project carries some amount of risk
- Volatility
  - ✓ Volatility of requirements is considered a risk factor and is sometimes handled as part of the risk aspect. A point of view is to handle them separately
- Other aspects
  - ✓ Financial benefit, strategic benefit, competitors, competence/resources, release theme, ability to sell
- It is important for stakeholders to develop a list of aspects to help in decision-making process
- Combining different aspects

Prioritization Techniques

- The purpose of any prioritization is to assign values to distinct prioritization objects that allow establishment of a relative order between the objects in the set. In our case, the objects are the requirements to prioritize
- The prioritization can be done with various measurement scales and types

- The least powerful prioritization scale is the ordinal scale, where the requirements are ordered so that it is possible to see which requirements are more important than others, but not how much more important
- The ratio scale is more powerful since it is possible to quantify how much more important one requirement is than another
- An even more powerful scale is the absolute scale, which can be used in situations where an absolute number can be assigned

Analytical Hierarchy Process (AHP)

- AHP is a systematic decision-making method that has been adapted for prioritization of software requirements
- It involves comparing all possible pairs of hierarchically classified requirements, in order to determine which has higher priority, and to what extent

- The total number of comparisons to perform with AHP are n * (n-1)/2; where n is the number of requirements; at each hierarchy level, which results in a dramatic increase in the number of requirements
- Studies have shown that AHP is not suitable for large number of requirements

Cumulative Voting, the 100-Dollar Test

- The 100-dollar test is a very straightforward prioritization technique where the stakeholders are given 100 imaginary units (money, hours, etc.) to distribute between the requirements
- The result of the prioritization is presented on a ratio scale
- One should only perform the prioritization once on the same set of requirements, since the stakeholders might bias their evaluation the second time around if they do not get one of their favorite requirements as a top priority

Numerical Assignment (Grouping)

- It is the most common prioritization technique, and is based on grouping requirements into different priority groups
- The number of groups can vary, but in practice, three groups are very common
- When using numerical assignment, it is important that each group represents something that the stakeholders can relate to (e.g. critical, standard, optional), for a reliable classification
- Using relative terms such as high, medium, and low will confuse the stakeholders
- Everything can be marked critical

Ranking

- As in numerical assignment, ranking is based on an ordinal scale but the requirements are ranked without ties in rank
- This means that the most important requirement is ranked 1 and the least important is ranked n (for n requirements)

Top-Ten Requirements

- In this approach, the stakeholders pick their top-ten requirements (from a larger set) without assigning an internal order between the requirements
- This makes the approach especially suitable for multiple stakeholders of equal importance
- The reason to not prioritize further is that it might create unnecessary conflict when some stakeholders get support for their top priority and others only for their third priority
- It is not advisable to take average across all stakeholders since it might lead to some stakeholders not getting any of their top requirements
- The main challenge in this technique is to balance issues related to the fact that top priority requirements of all stakeholders are included in the next development activity

Summary of Presented Techniques

	Scale	Granularity	Sophistication
AHP	Ratio	Fine	Very Complex
100 Dollar Test	Ratio	Fine	Complex
Ranking	Ordinal	Medium	Easy
Numerical Assignment	Ordinal	Coarse	Very Easy
Top-Ten	---	Extremely Coarse	Extremely Easy

Hints and Tips on Requirements Prioritization

- A general advice is to use the simplest appropriate prioritization technique and use more sophisticated ones when a more sensitive analysis is needed for resolving disagreements or to support the most critical decisions
- More sophisticated techniques, generally, are more time consuming, the simplest possible technique ensures cost effective decisions
- The trade-off is to decide exactly how “quick and dirty” the approach can be without letting the quality of the decisions suffer
- Commercial tools are available that facilitate the use of sophisticated techniques
- Different techniques can be combined also

Involvement of Stakeholders

- One Customer
- Several Known Customers
- Perspectives of customers, developers, and financial representatives
- Mass Market
- Concept of personas

Requirements Prioritization Issues

- Abstraction level
- Reprioritization
- Non-functional requirements
- Introducing prioritization into an organization
- Evaluating prioritization
- Using the results of requirements prioritization