- 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

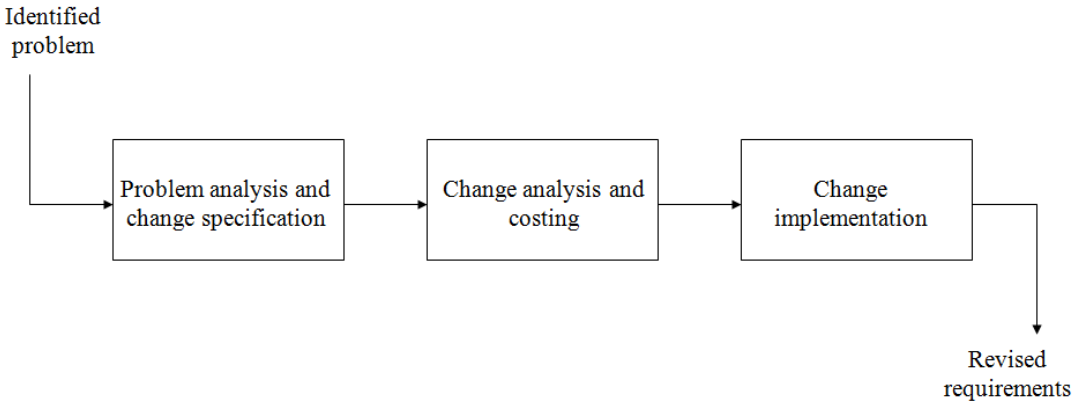
Change Management

- 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

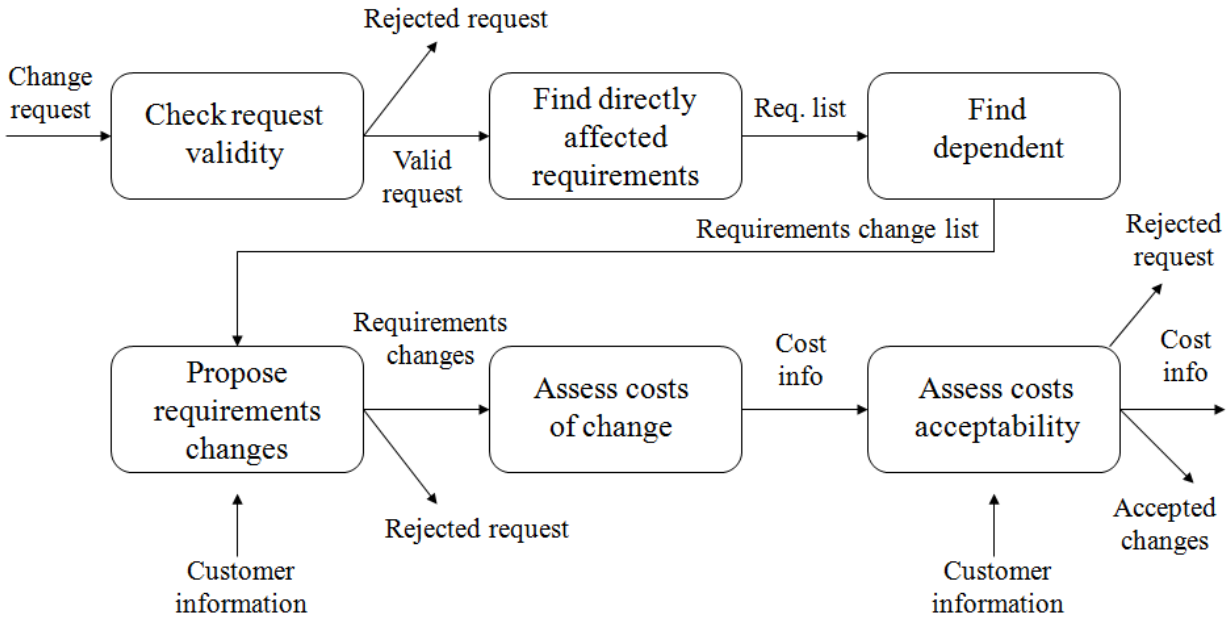
Change Analysis and Costing

- 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

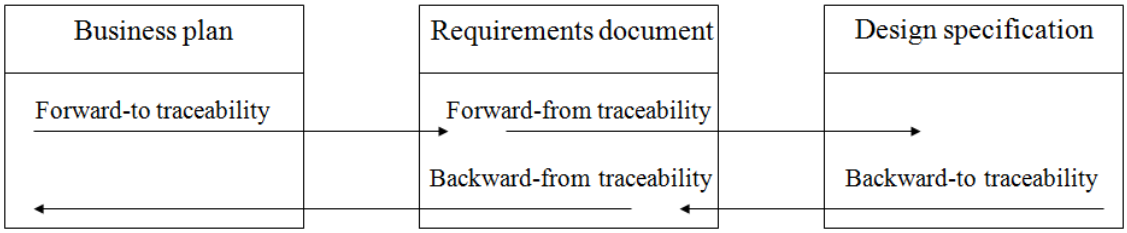
- It is important to trace requirements both ways
  - ✓ Origin of a requirement
  - ✓ How is it implemented
- This is a continuous process

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

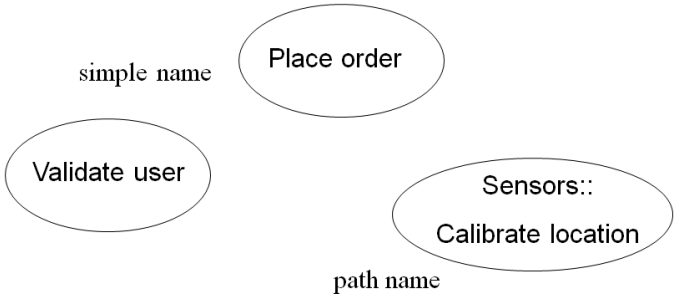
Use Cases

- 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

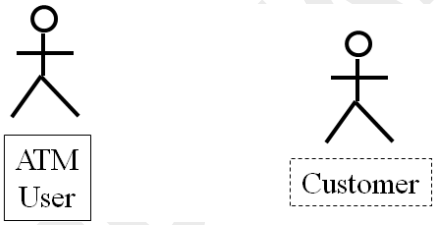
- Every use case must have a name that distinguishes it from other use cases
- A name is a textual string, consisting of any number of letters, numbers, and most punctuation marks
- In practice, use case names are short active verb phrases naming some behavior found in the vocabulary of the system being 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

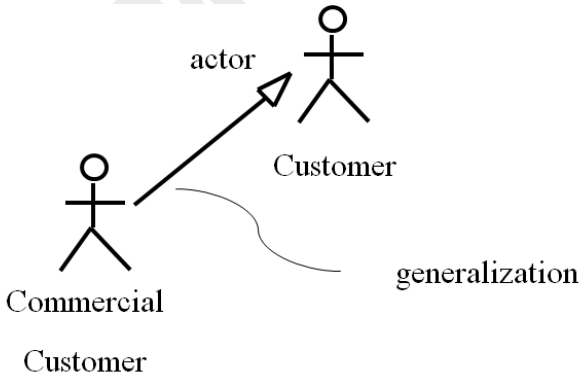


Actors

- 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



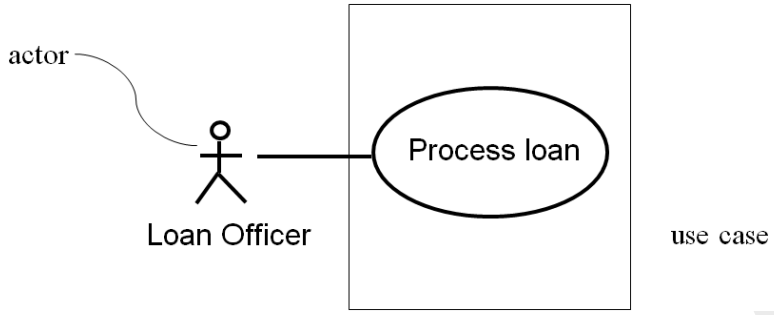
Specialized Actors



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

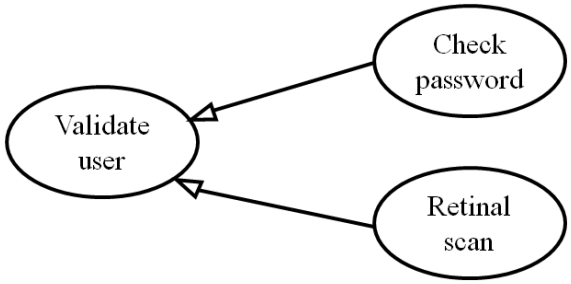
- A use case describes what the system (or subsystem, class, or interface) does but it does not specify how it does it
- It is important that you keep clear the separation of concerns between this outside and inside view
- The behavior of a use case can be specified by describing a flow of events in 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

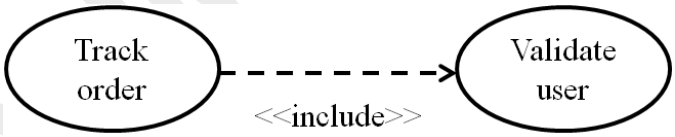
- Use cases can be organized by grouping them in packages
- You can also organize use cases by specifying generalization, include, and extend relationships among them
- You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and 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)

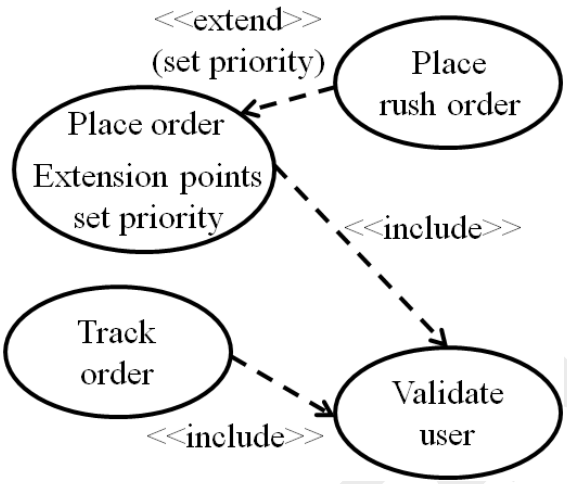


- An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base
- The included use case never stands alone, but is only instantiated as part of some larger base that includes it
- It is like the base 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

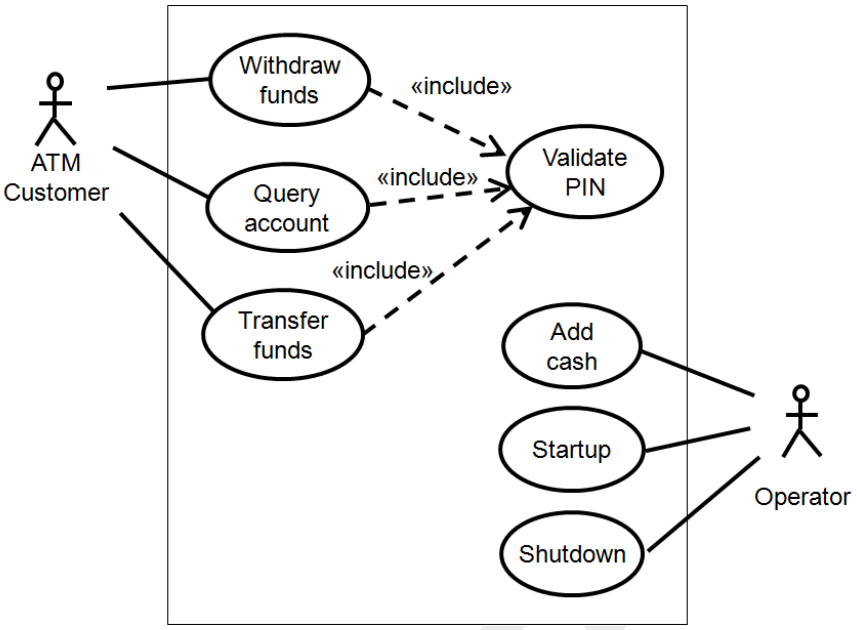


- In this example, set point is an extension point
- A use case may have more than one extension points
- Apply use cases to model the behavior of an element (system, subsystem, class)
- Outside view by domain experts
- Developers can approach the element
- Basis for testing

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

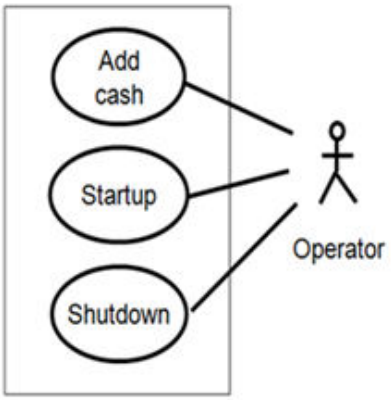
- Withdraws funds from the checking or savings account
- Query the balance of an account
- Transfer funds from one account to another
- The ATM customer interacts with the system via the ATM card reader, keyboard/display, cash dispenser, and receipt printer

ATM Operator

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

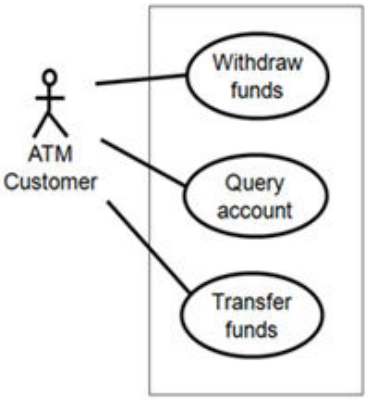
Use Cases for ATM Operator

- Add cash
- Startup
- Shutdown



User Case for ATM Customer

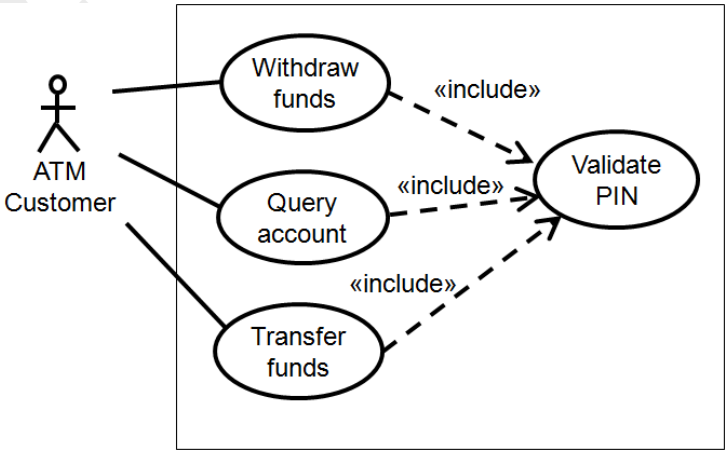
- Withdraw funds
- Query account
- Transfer funds



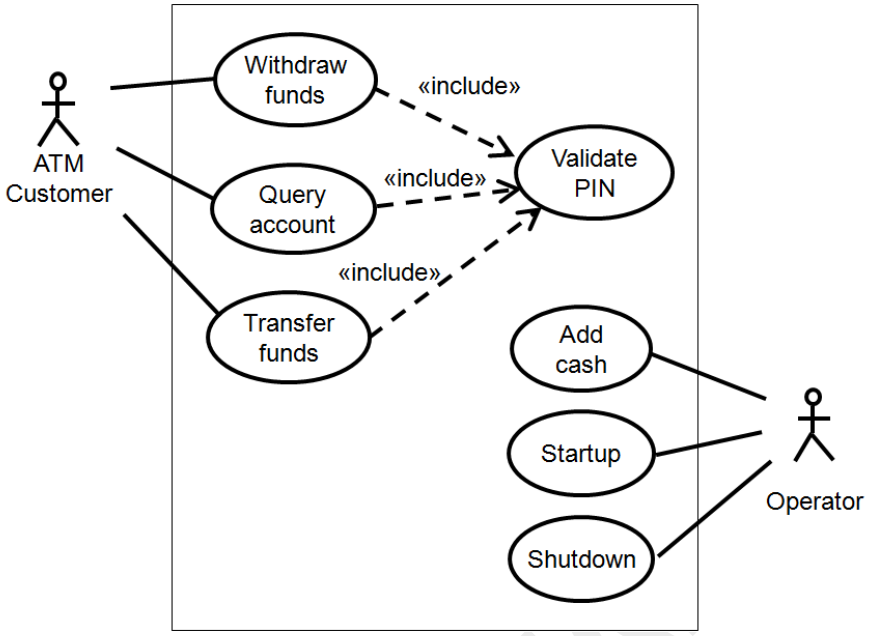
Organizing Use Cases for ATM Customer

- 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)

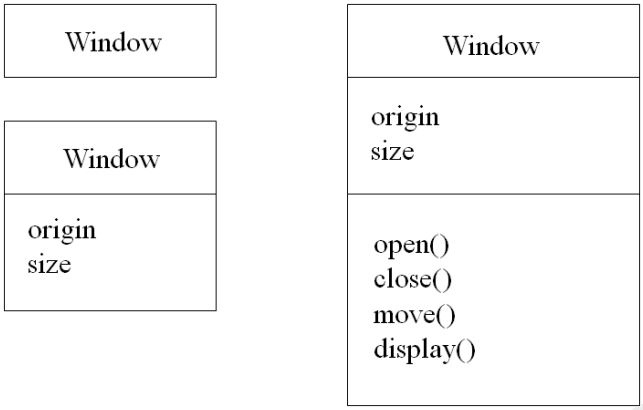
Unified Modeling Language

- 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

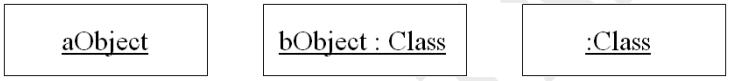
UML Notation for Classes



Objects and Classes

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

UML Notation for Objects

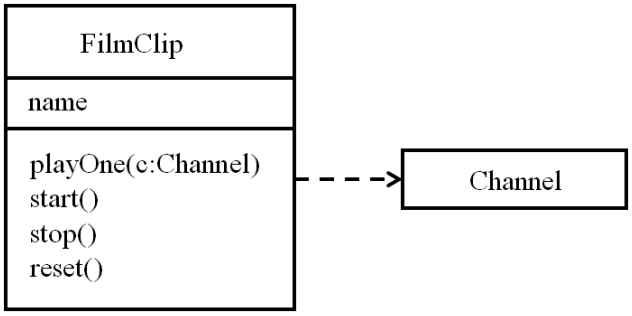


Relationships between Objects

- 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

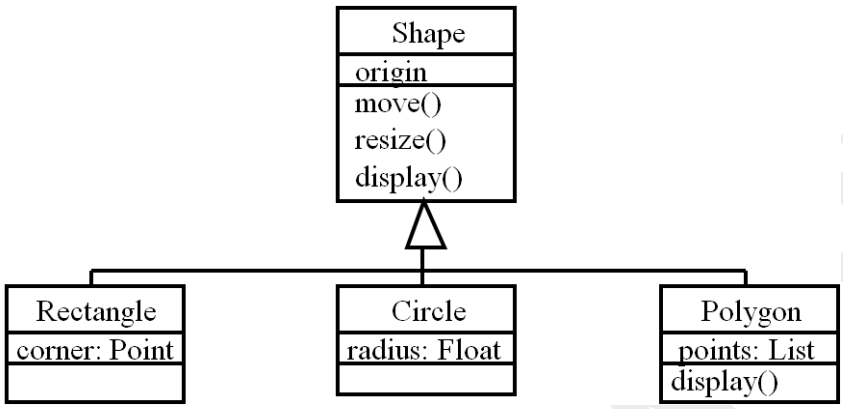
Dependency Relationship

- 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



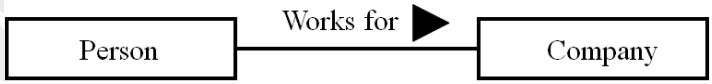
Generalization Relationship

- 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

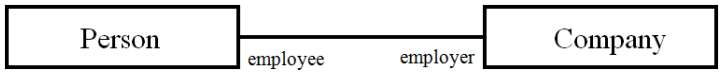


Association Relationship

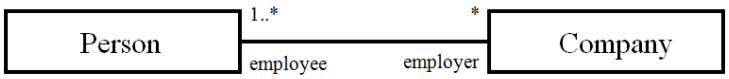
- An association is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa
- Graphically, an association is rendered 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:



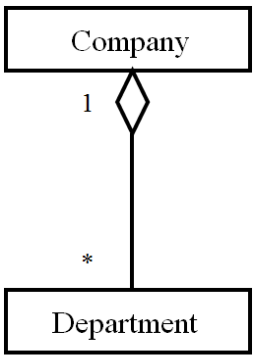
Association Names



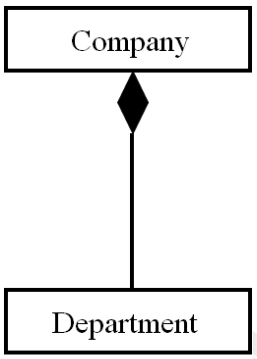
Roles



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

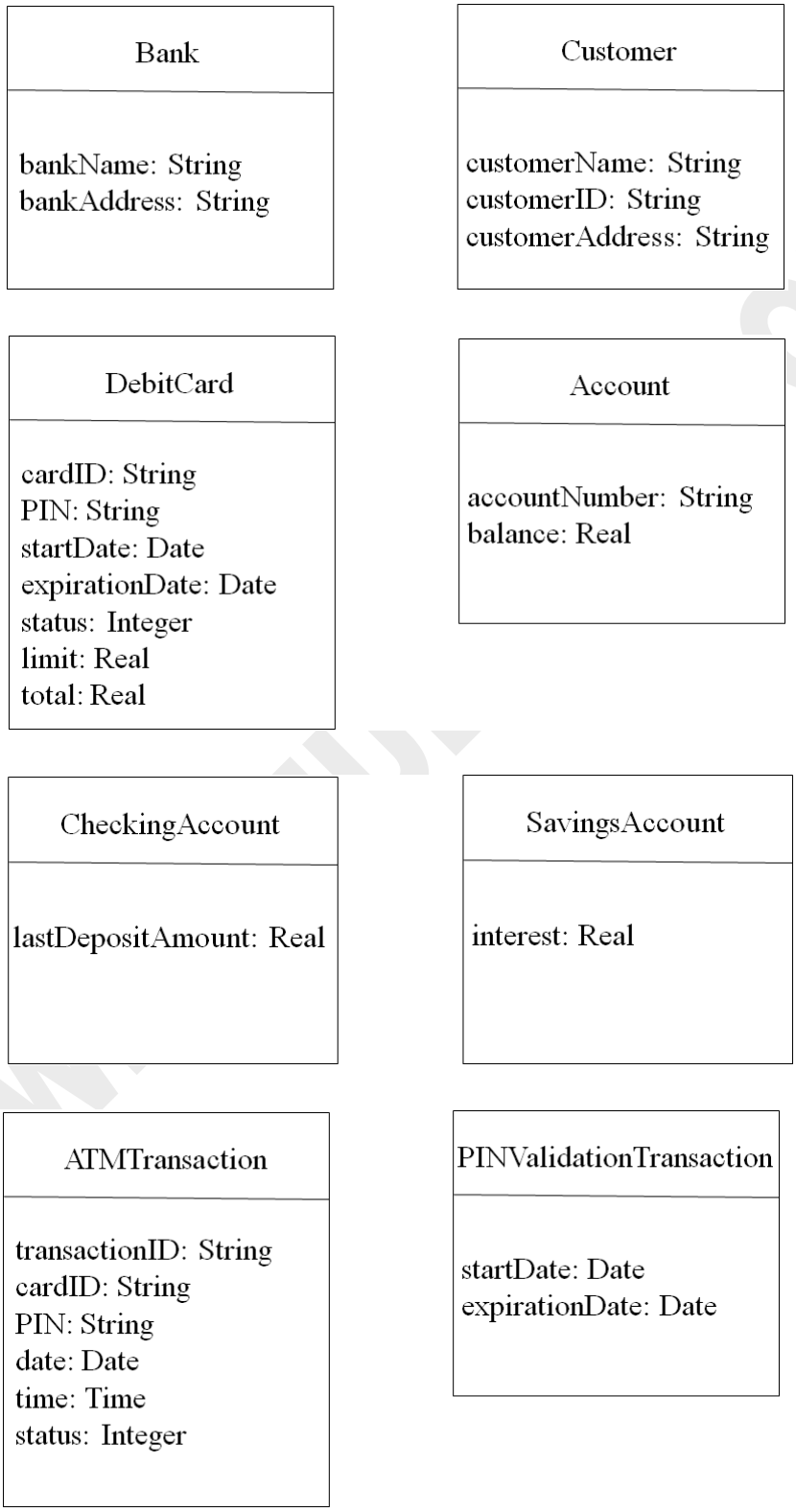
Observations

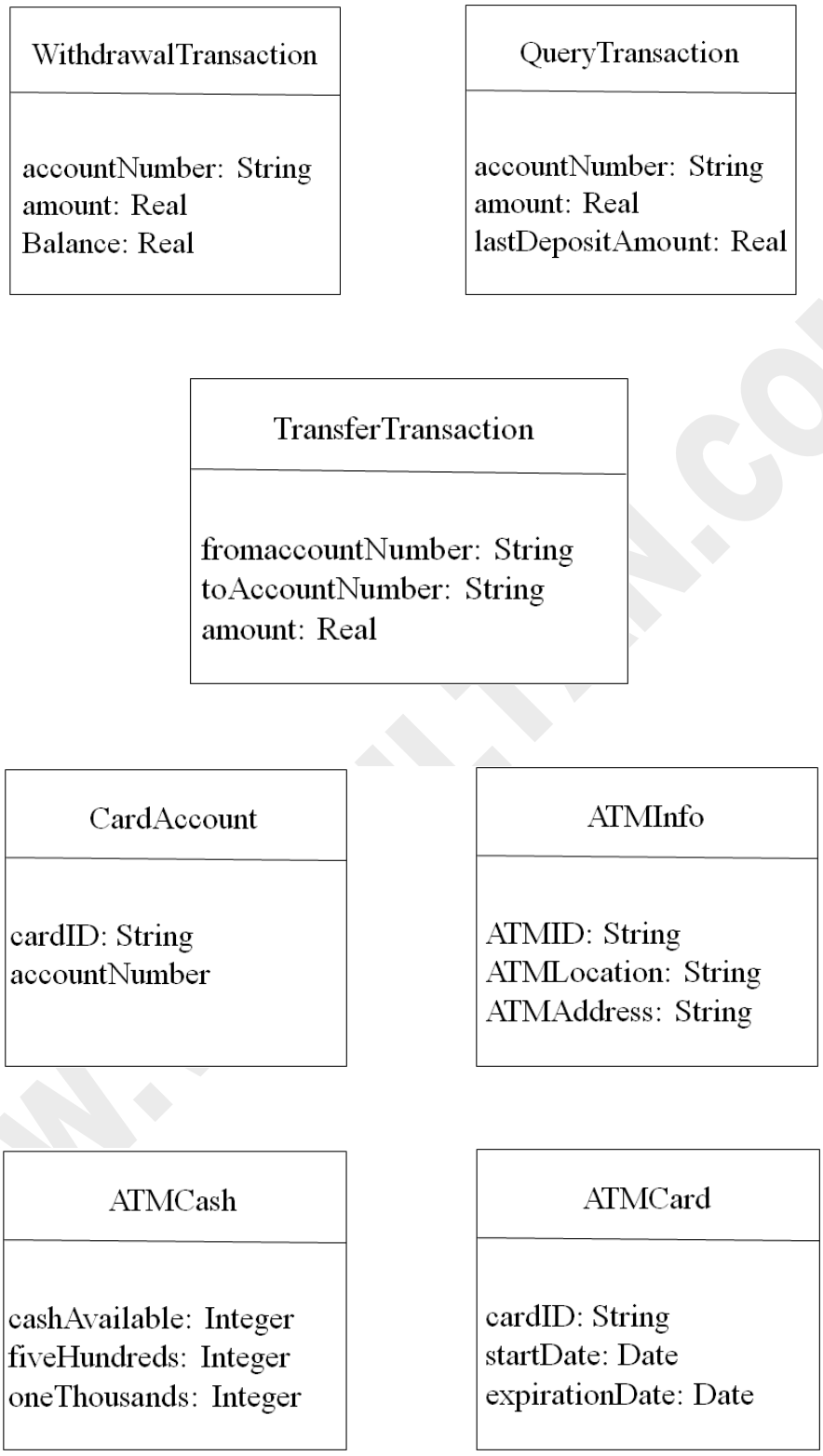
- A bank has several ATMs
- Each ATM has a card reader, a cash dispenser, a receipt printer, and a user who interacts with the ATM through a keyboard/display unit
- The card reader reads an ATM card - physical thing
- Dispensed cash and receipt are also physical entities
- ATM operator maintains an ATM

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

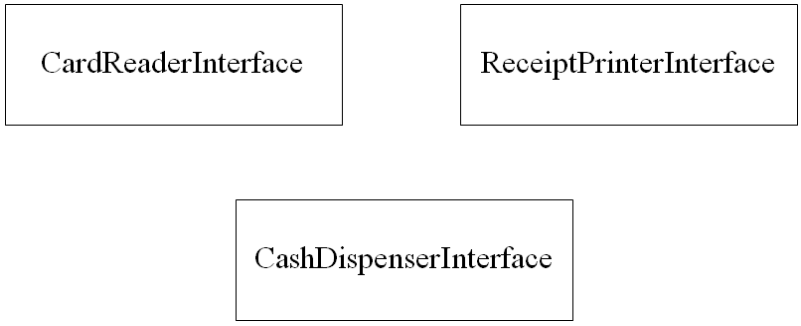
Entity Classes in Banking System



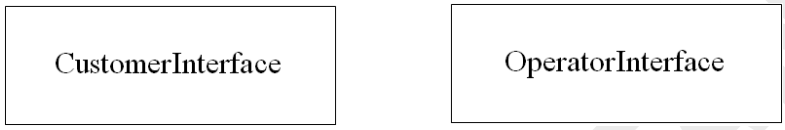

WithdrawalTransaction
accountNumber: String
amount: Real
Balance: Real
QueryTransaction
accountNumber: String
amount: Real
lastDepositAmount: Real
TransferTransaction
fromaccountNumber: String
toAccountNumber: String
amount: Real
CardAccount
cardID: String
accountNumber
ATMInfo
ATMID: String
ATMLocation: 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

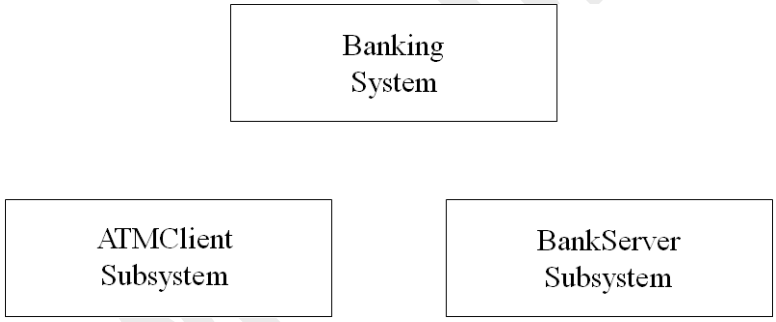


- User Interface Classes in the Banking System



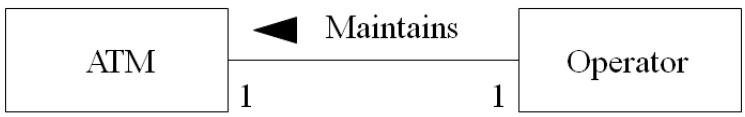
System and Subsystem Classes

- System and Subsystem Classes in the Banking System

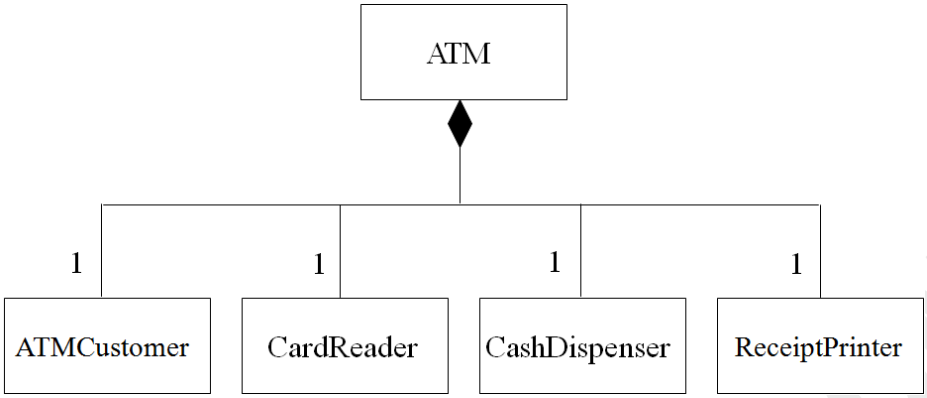


- A Bank Has Many ATMs

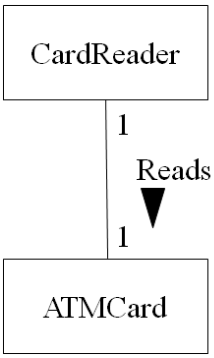
- An Operator Maintains an ATM



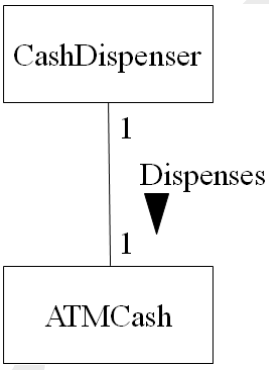
An ATM Has Other Objects



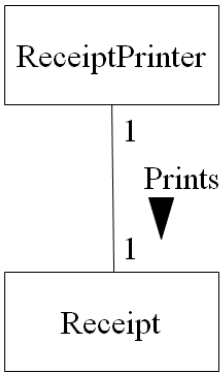
Relation Between CardReader and ATMCard



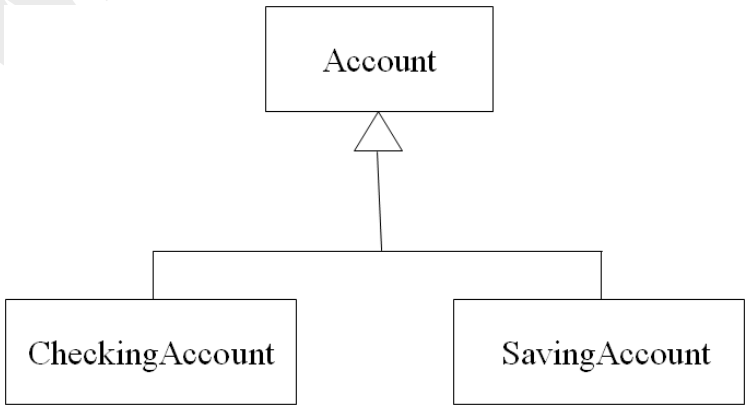
Relation Between CashDispenser and ATMCash



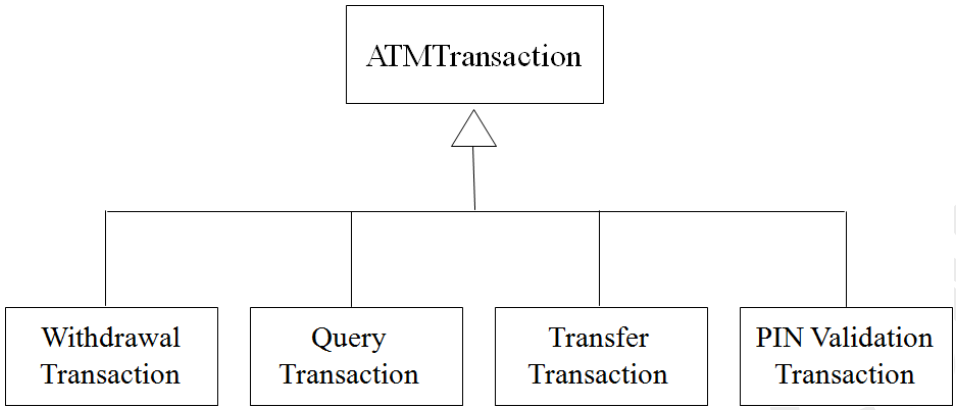
Relation Between ReceiptPrinter and Receipt



Relationship Between Account and CheckingAccount & SavingsAccount

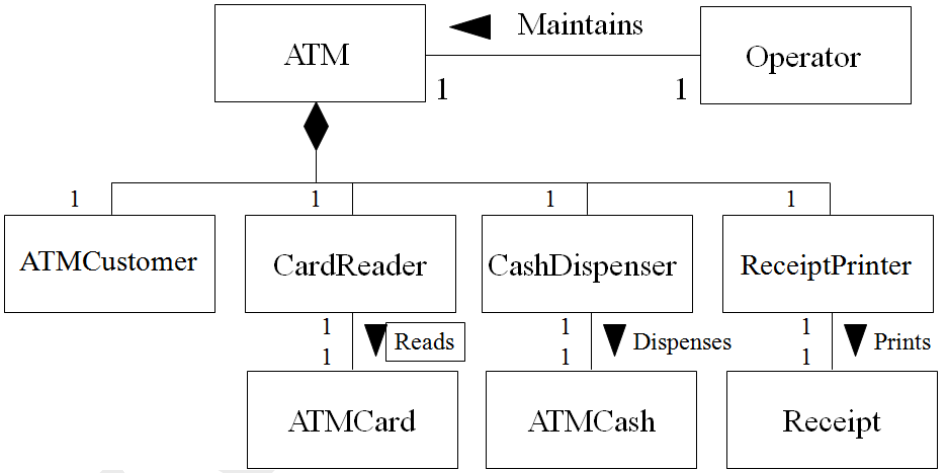


ATMTransaction and its Subclasses



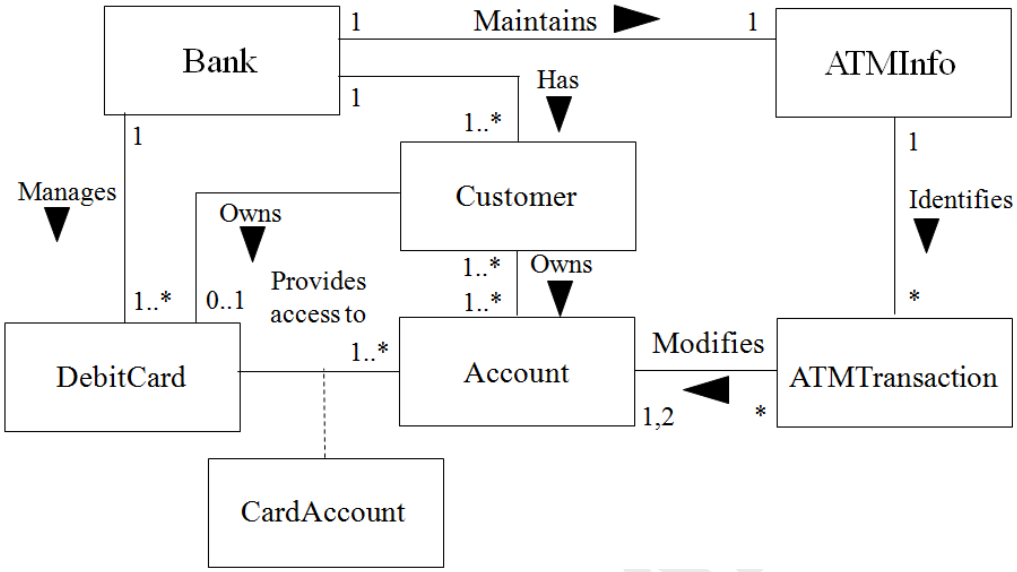
Putting the Classes Together

Conceptual Static Model for Problem Domain: Physical Classes



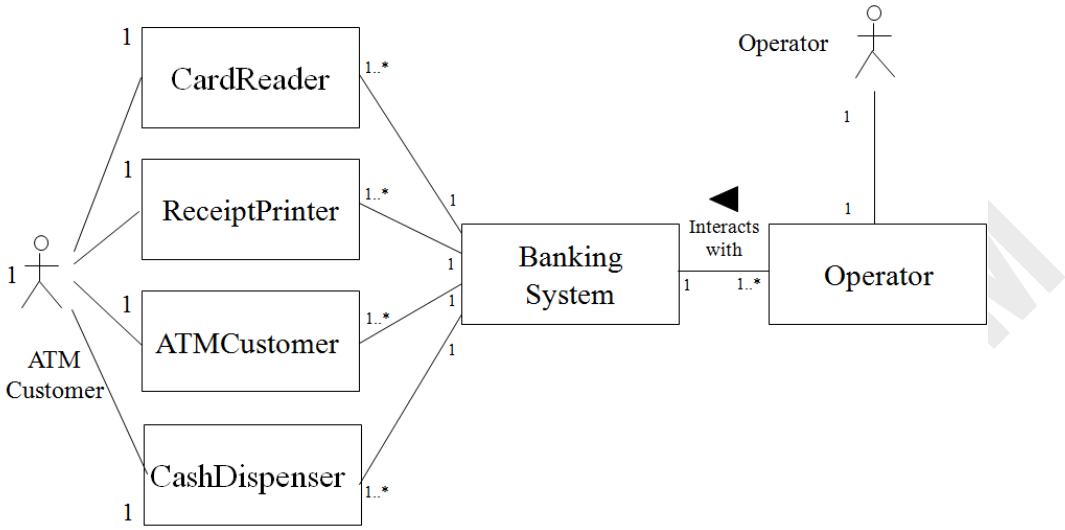
- A bank has several ATMs.
- Each ATM is a composite class consisting of a Card Reader, a Cash Dispenser, a Receipt Printer, and a user, the ATM Customer, who interacts with the system by using a keyboard/display
- The Card Reader reads an ATM Card, which is a plastic card and hence a physical entity
- 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

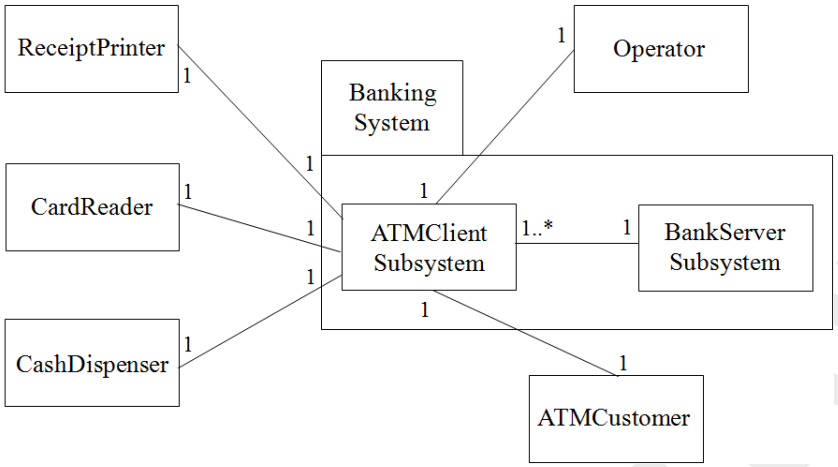


- The system context diagram is developed to show the external classes to which the Banking system has to interface
- We develop the context class diagram by considering the physical classes
- The external classes correspond to the users and I/O devices depicted to the classes in the application domain
- 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

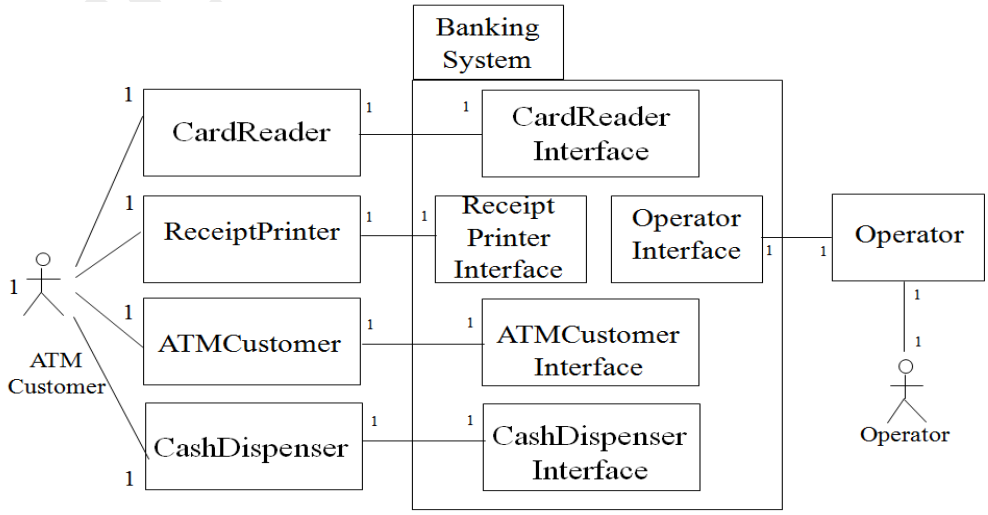


- The ATM Client Subsystem is a composite class, and one instance of this class is located at each ATM machine
- All the external classes interface to this aggregate class
- The Bank Server Subsystem has one instance
- This aggregate class has a one-to-many association with the ATM Client Subsystem
-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

ATM Client Object Structuring

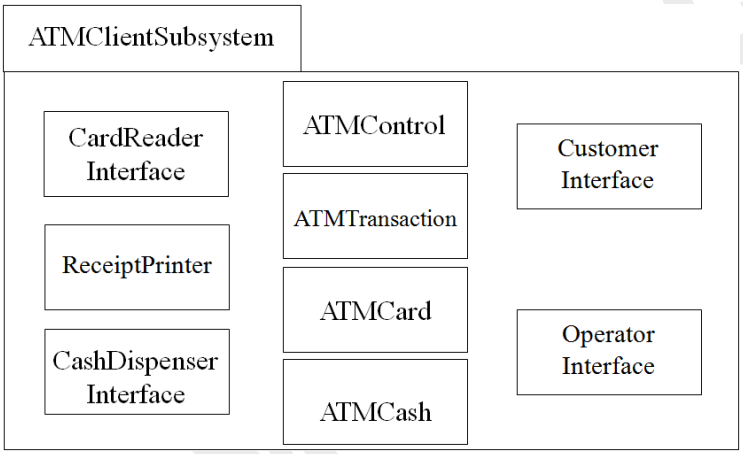
- Consider the interface classes for the ATM Client Subsystem. These classes are determined from the system context diagram
- We design one interface class for each external class, which can be a device interface class or user interface class

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

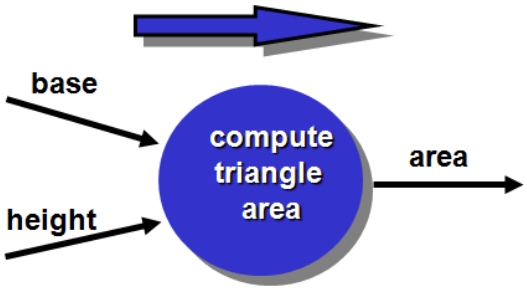
- Data flow diagrams are composed of data on the move, transformations of data into other data, sources and destinations of data, and data in static storage
- Data flow diagrams show the flow of data through a system

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

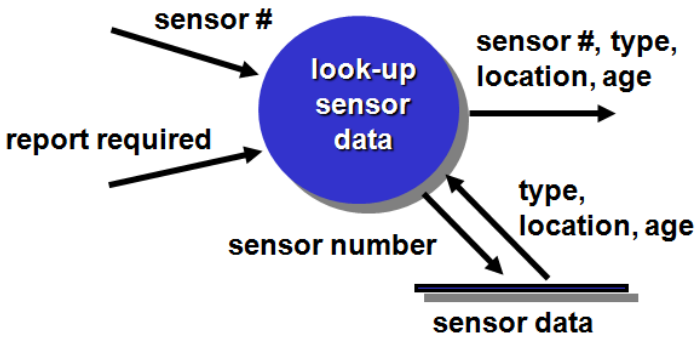
Data Flow

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



Data Stores

- Data is often stored for later use.

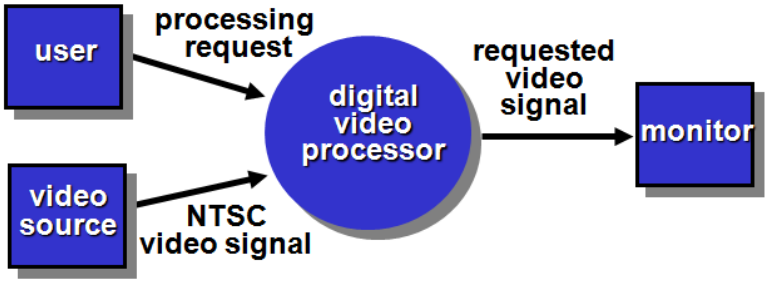


Data Flow Diagramming: Guidelines

- 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

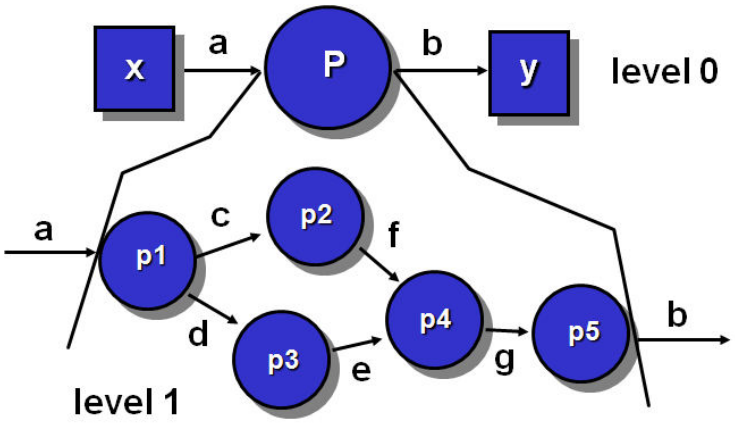
Constructing a DFD

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



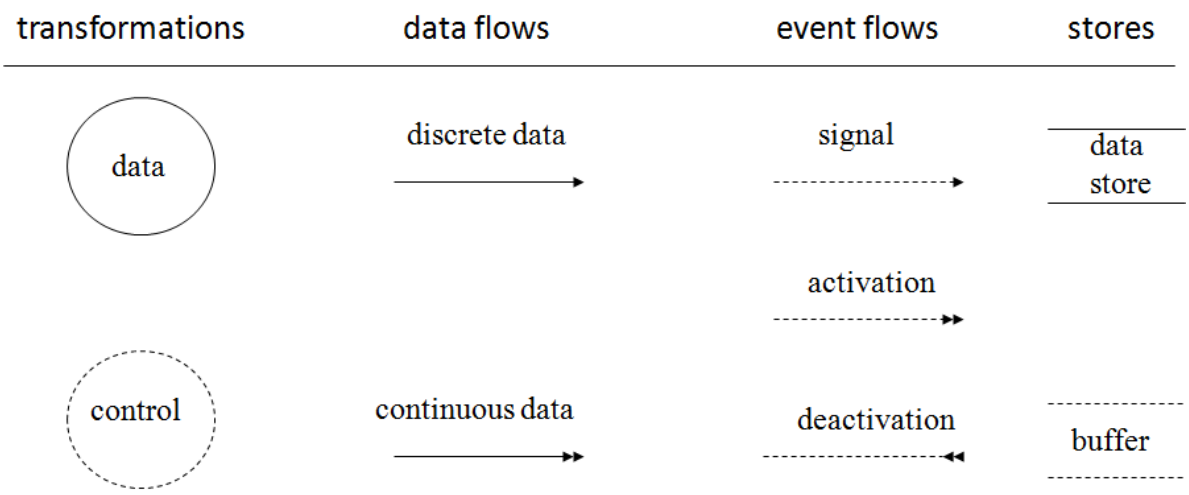
- Write a narrative describing the transform
- Parse to determine next level transforms
- "Balance" the flow to maintain data flow continuity
- Develop a level 1 DFD
- Use a 1:5 (approx.) expansion ratio

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

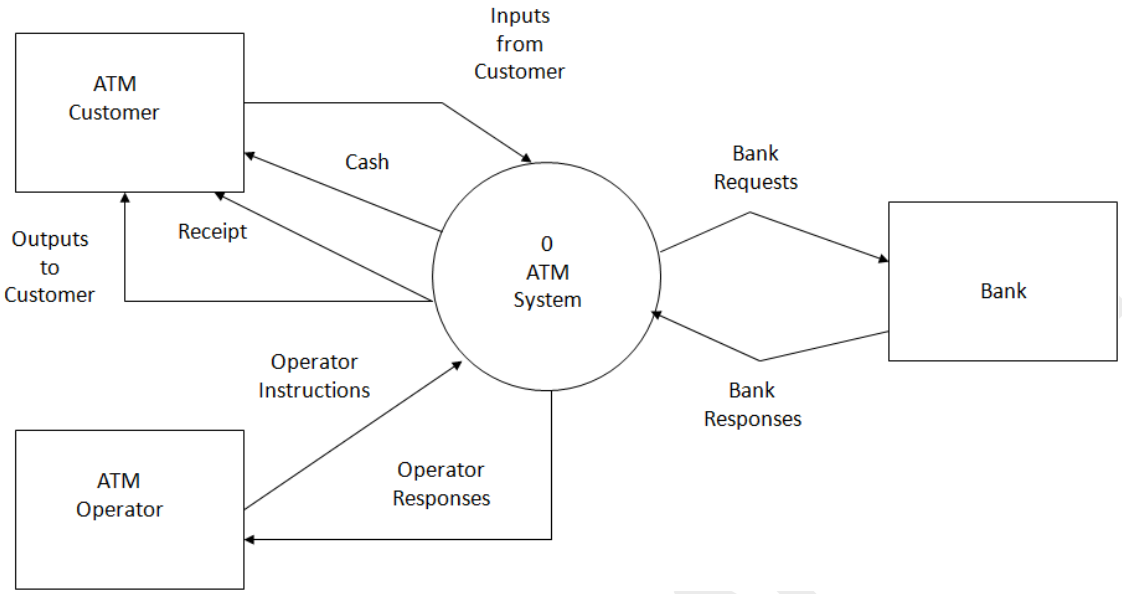
Function-oriented Modeling Techniques

- 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

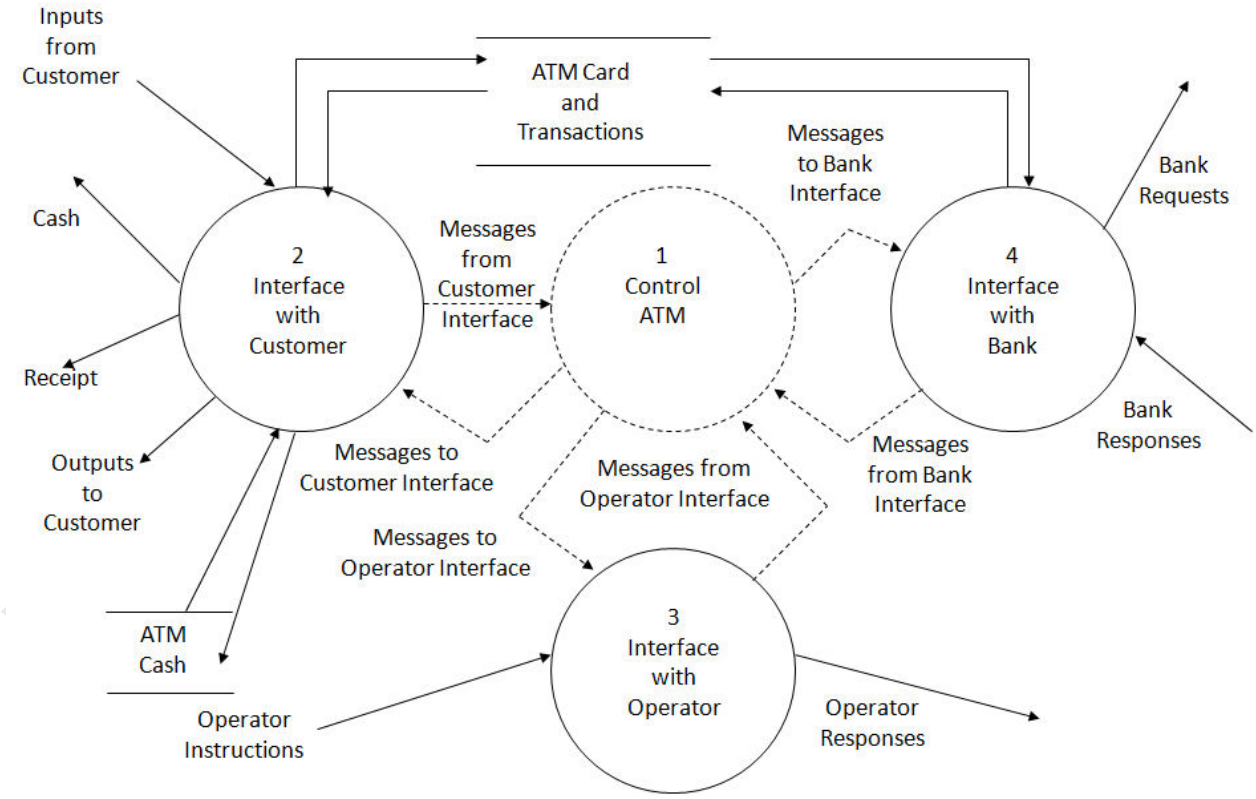
Real-Time Structured Analysis and Structured Design (RSTAD)

- 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

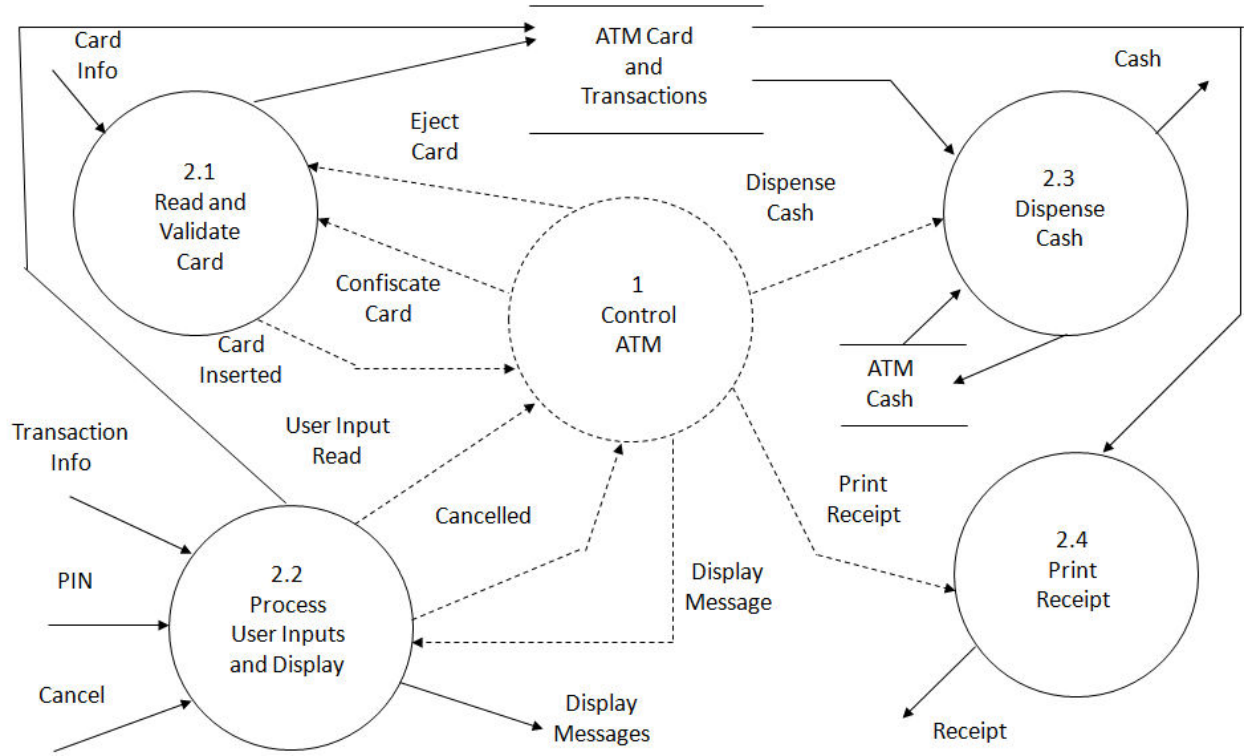
System Context Diagram



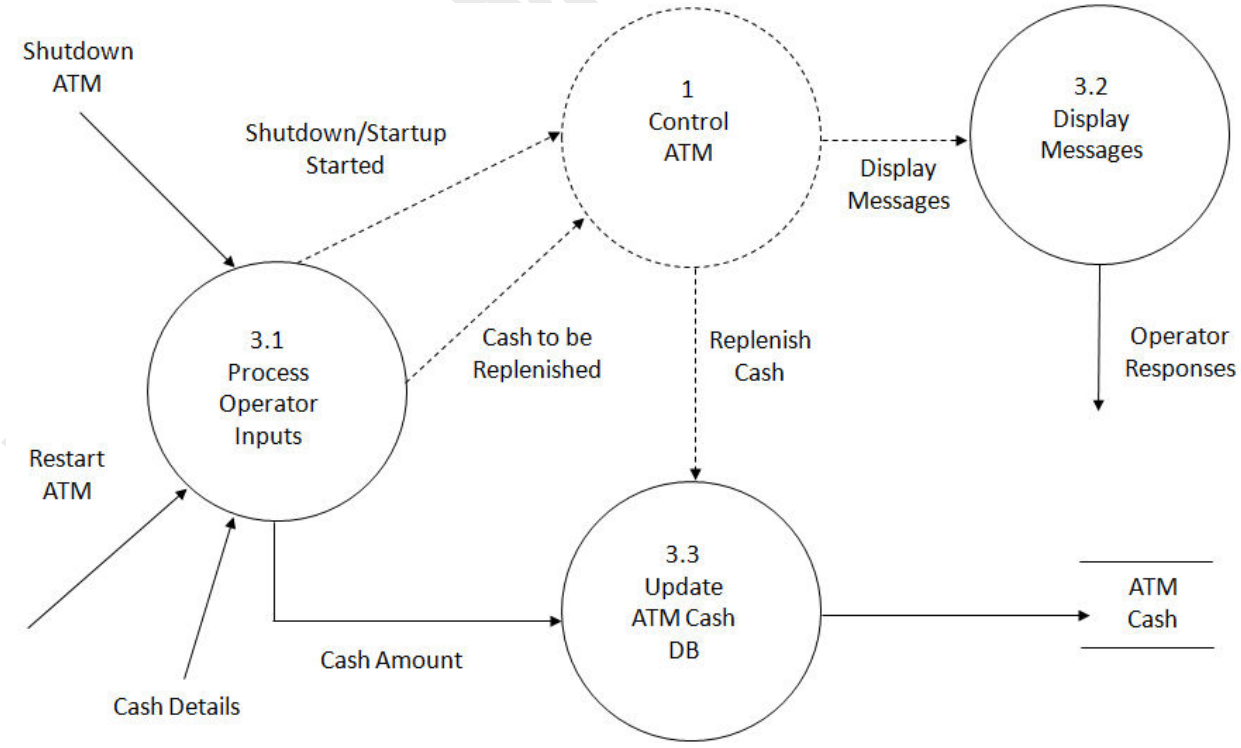
Data Flow Diagram – Level 1



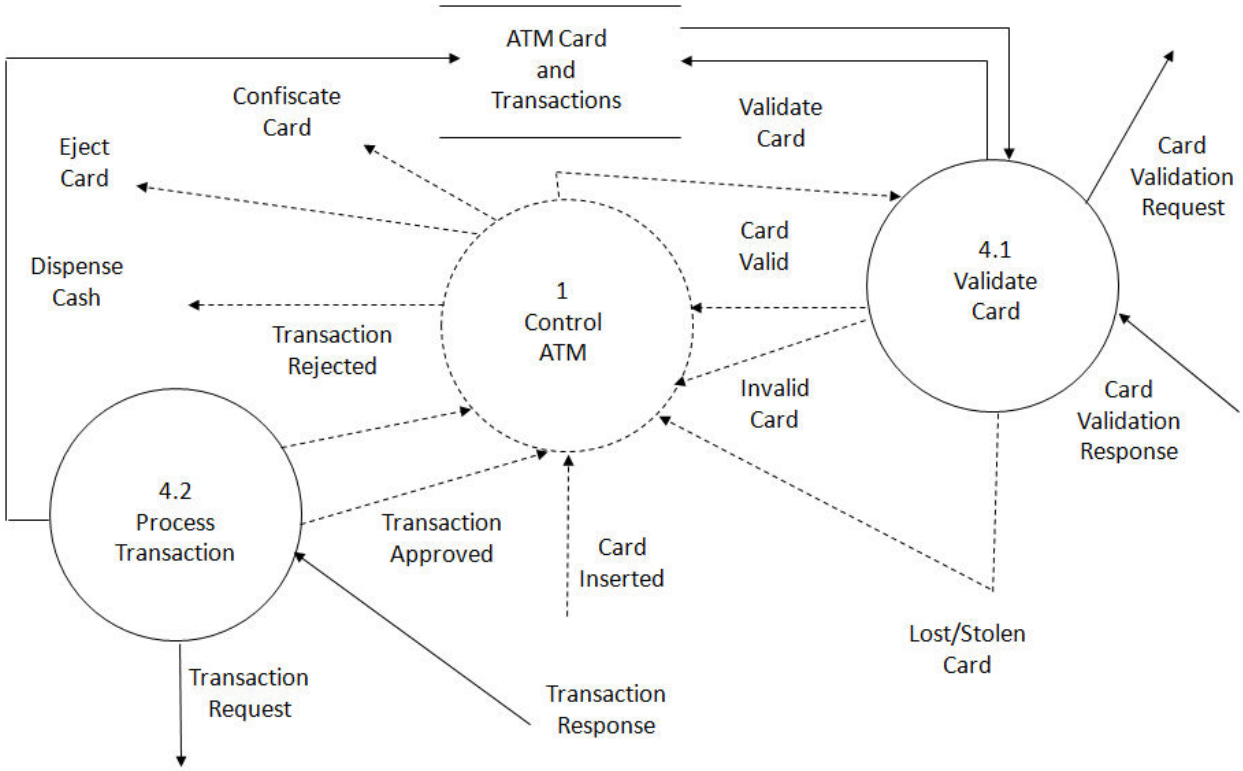
Level 2 DFD: Interface with Customer



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

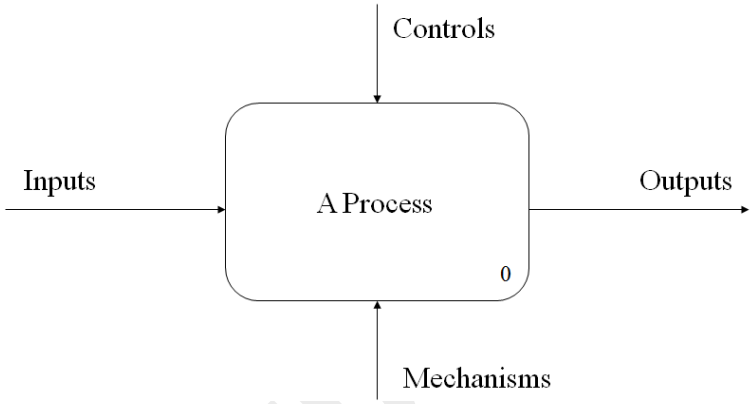
- Name of the data item
- Aliases
- Description/purpose
- Related data items
- Range of values
- Data flows
- Data structure definition/form

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

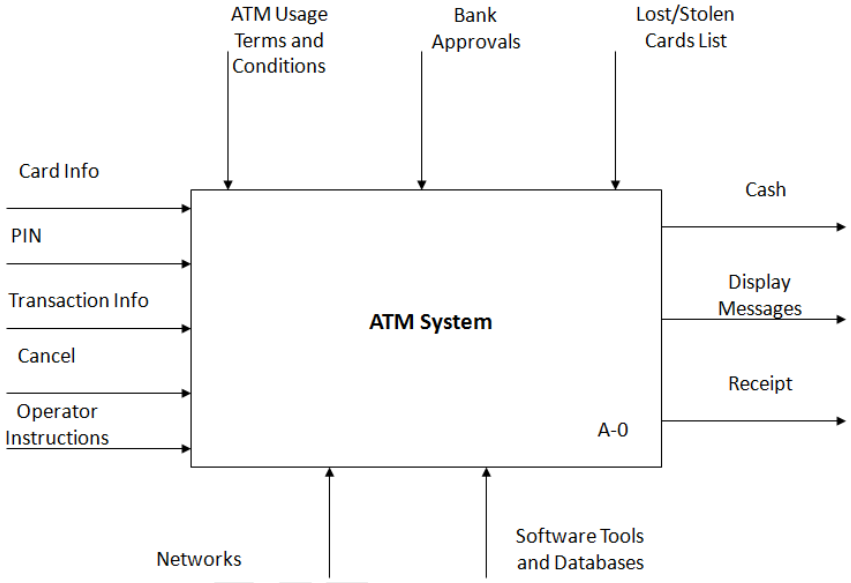
- Diagrams based on simple box and arrow graphics, arrows convey data or objects
- Text labels to describe boxes and arrows and glossary and text to define the precise meanings of diagram elements
- Box name shall be a verb or verb phrase, such as "Perform Inspection", arrows are nouns
- Gradual exposition 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

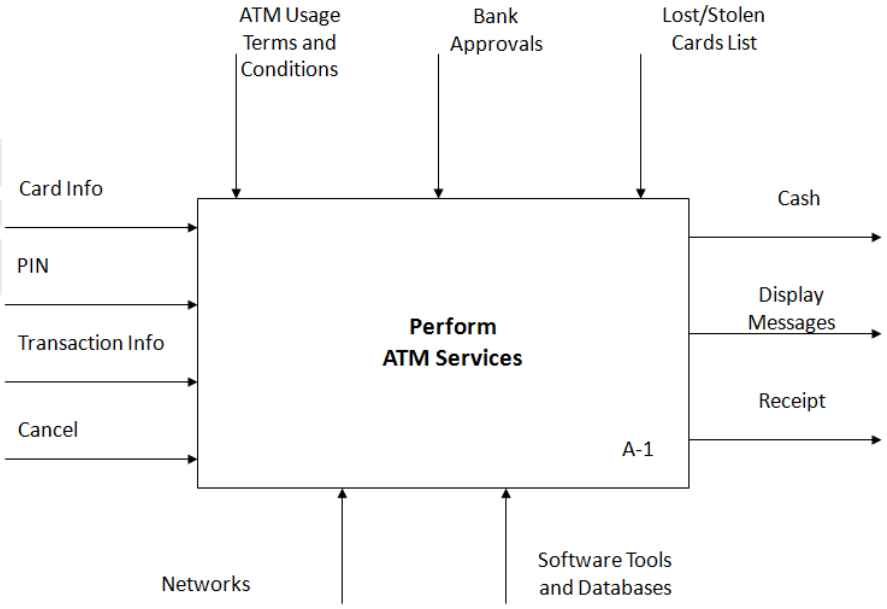
- There is always an A-0 context diagram, and its box number is always 0. This is a strict part of the standard that helps identify the overall description of the system
- A non-context diagram has from 3 to 6 boxes. This helps us manage detail
- Each box is numbered in its lower right corner, generally 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

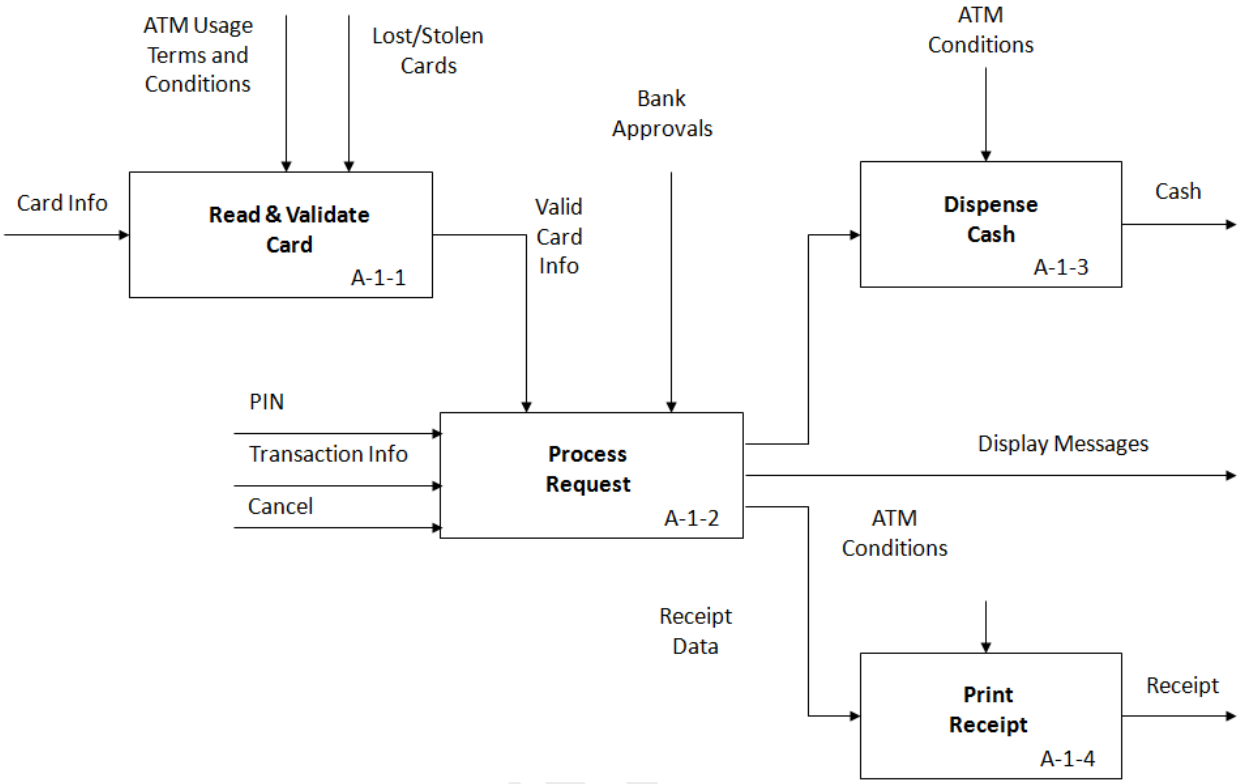
Banking System Context Diagram



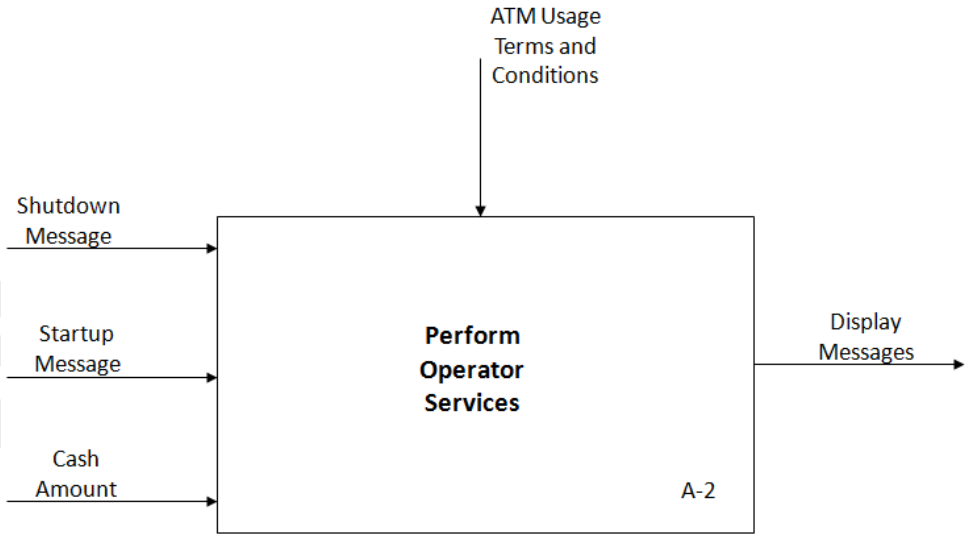
SADT Level 1 Diagram



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

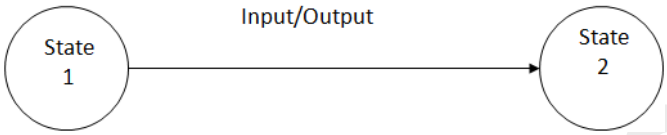
1. Finite state machines (FSM)
2. Statecharts
3. Petri nets
4. Decision tables and decision trees
5. Collaboration diagrams
6. Sequence diagrams

1 – Finite State Machines

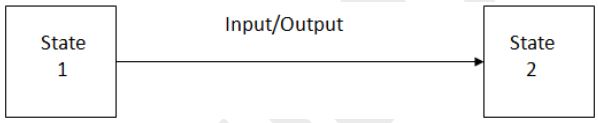
- A finite state machine (FSM) is a hypothetical machine that can be in only one of a given number of states at any specific time
- In response to an input, the machine generates an output and changes state
- Both the output (O) and the next state (S_N) are purely functions of the current state (S_C) 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

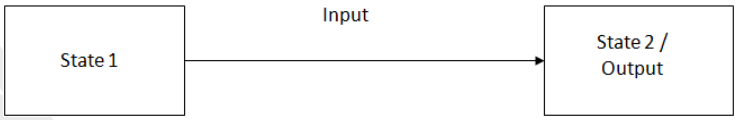
- A circle denotes a state
- A directed arc connecting two states denotes the potential to transition between the two indicated states
- A label on the arc, which has two parts separated by a slash, means the input that triggers the transition and the output with which the system responds
- Mealy model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

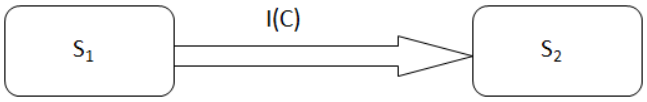


- Another model exists for state transition diagrams, Moore model, in which system responses are associated with the state rather than the transition between states
- On Moore STDs, arcs are labeled with only the stimulus name, and circles are labeled with the state name and the system response

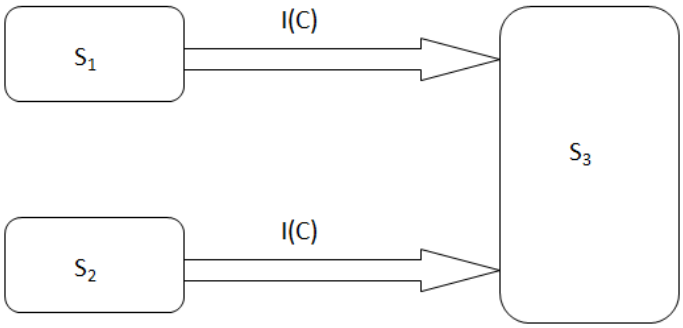


2 – Statecharts

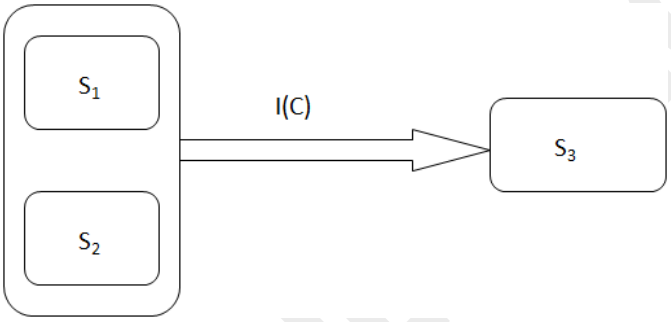
- Statecharts are an extension to finite state machines, proposed by Harel
- They provide a notation and a set of conventions that facilitate the hierarchical decomposition of finite state machines and a mechanism for communication between concurrent finite state machine
- Statecharts allow a transition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



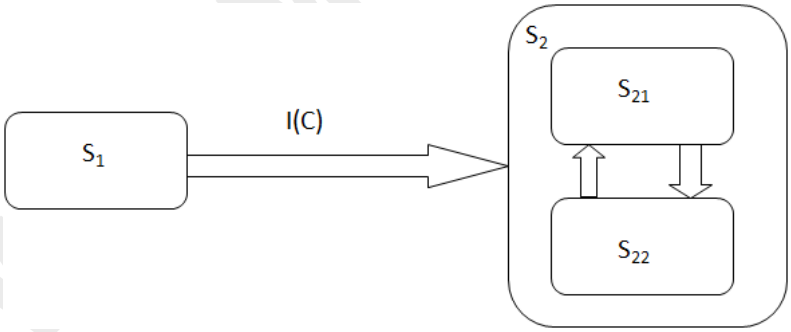
- The concept of superstate, which can be used to aggregate sets of states with common transitions

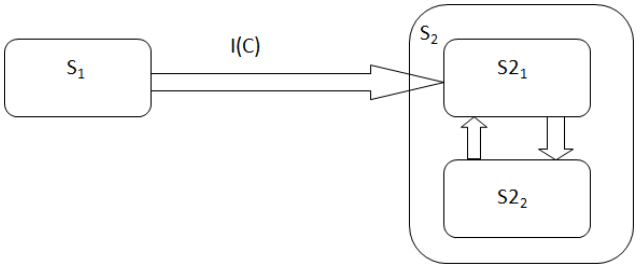


- A super-state can be used an abstract state, in which all states transition to a particular state

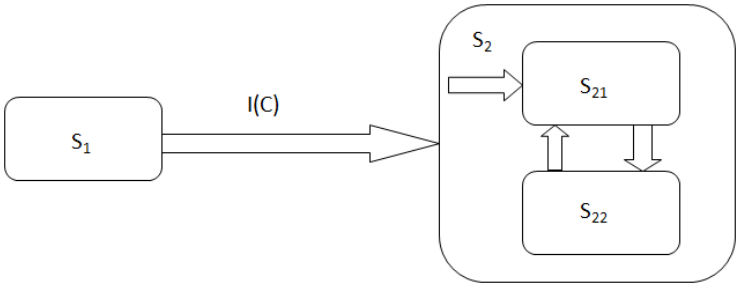


- A super-state can be used to transition into from more than one states

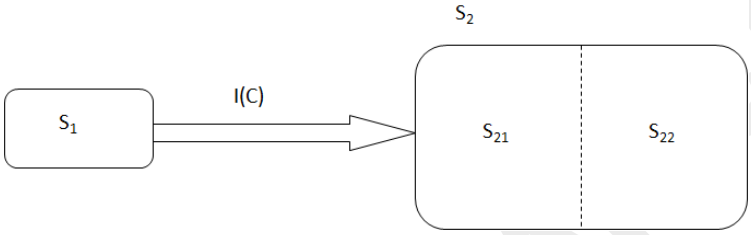




- Default entry state/ 'or' decomposition

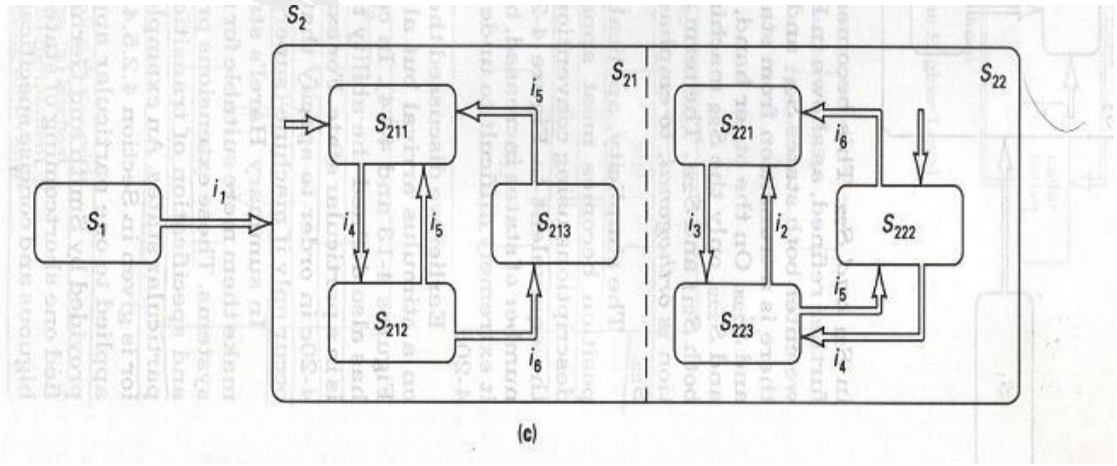


- Statecharts describes the 'and' function of state refinement

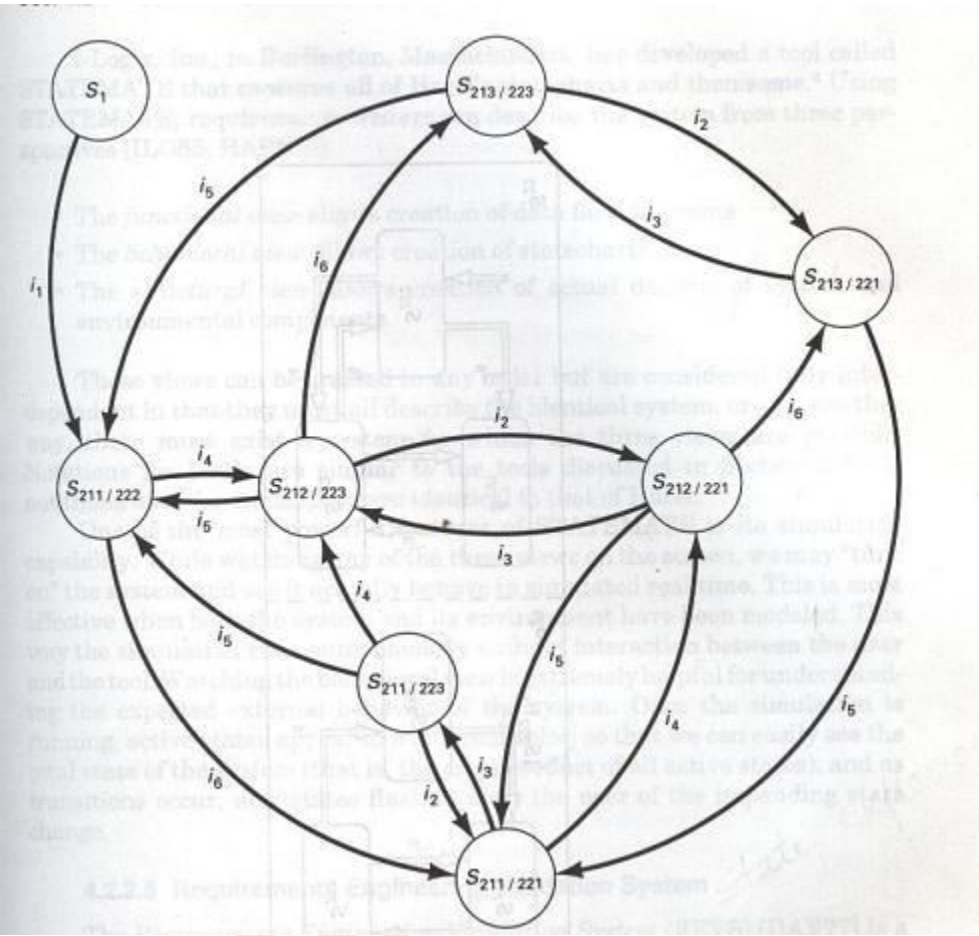


- The simplicity, applicability, and elegance of Harel's orthogonal decomposition becomes most apparent when we compare equivalent behavioral descriptions using conventional state transition diagrams
- Statecharts provide natural extensions to FSMs to make them more suitable for specifying external behavior 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



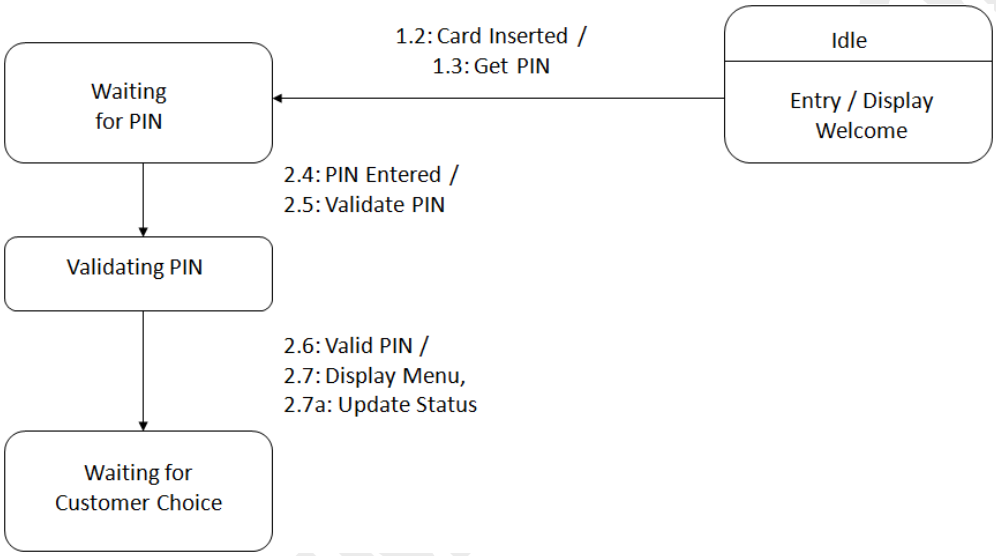
Dynamic Modeling (Continued)

- 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

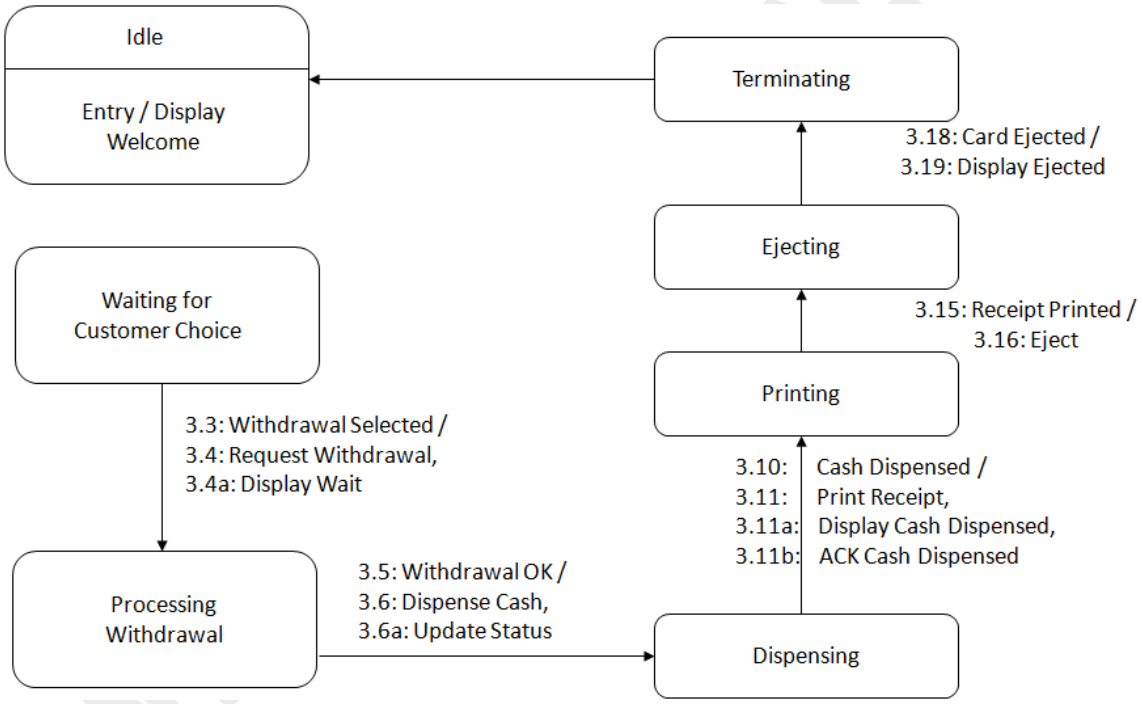


Statechart for ATM Control: Validate PIN Use Case

- **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

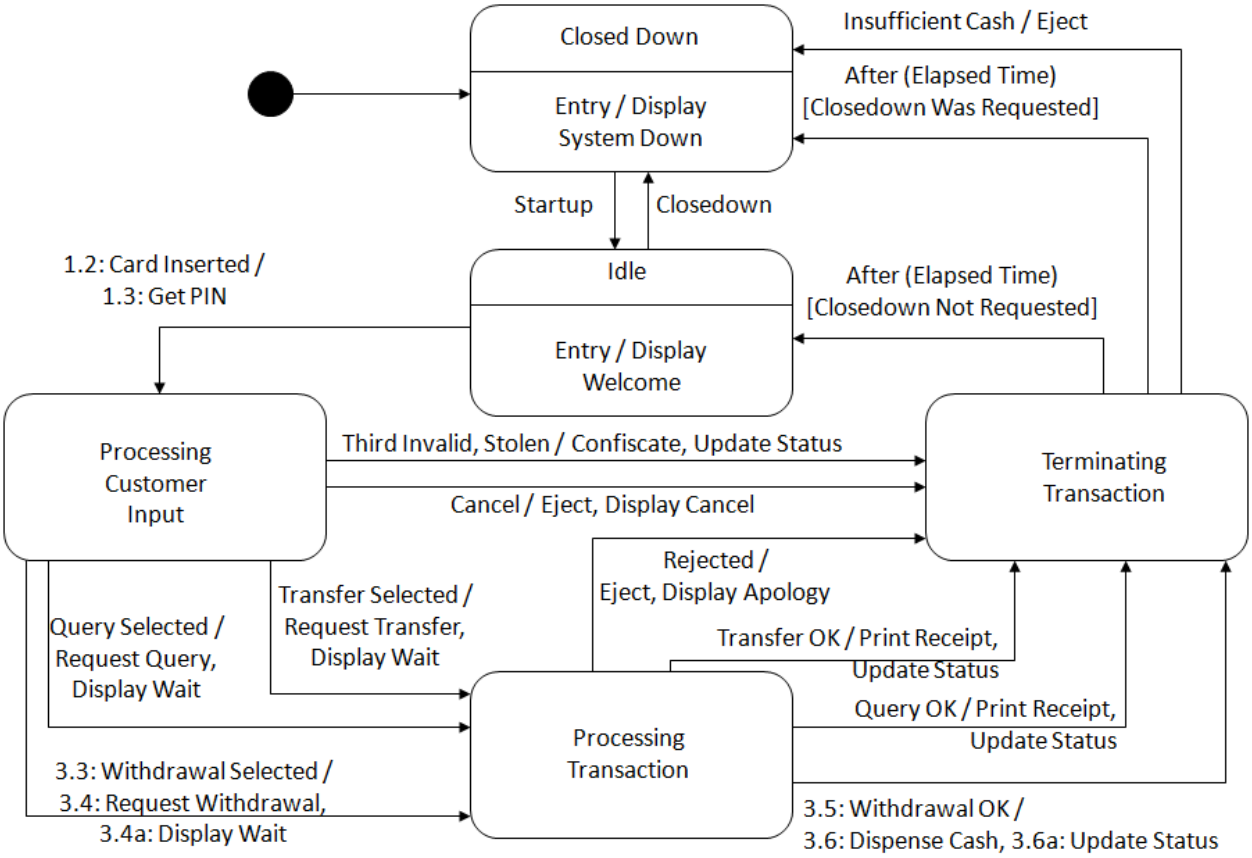
- **3.15:** Receipt Printer Interface sends the Receipt Printed event to ATM Control. As a result, ATM Control transitions to Ejecting state. The output event is Eject
- **3.16:** ATM Control sends the Eject event to Card Reader Interface
- **3.17:** Card Reader Interface sends the Card Reader Output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

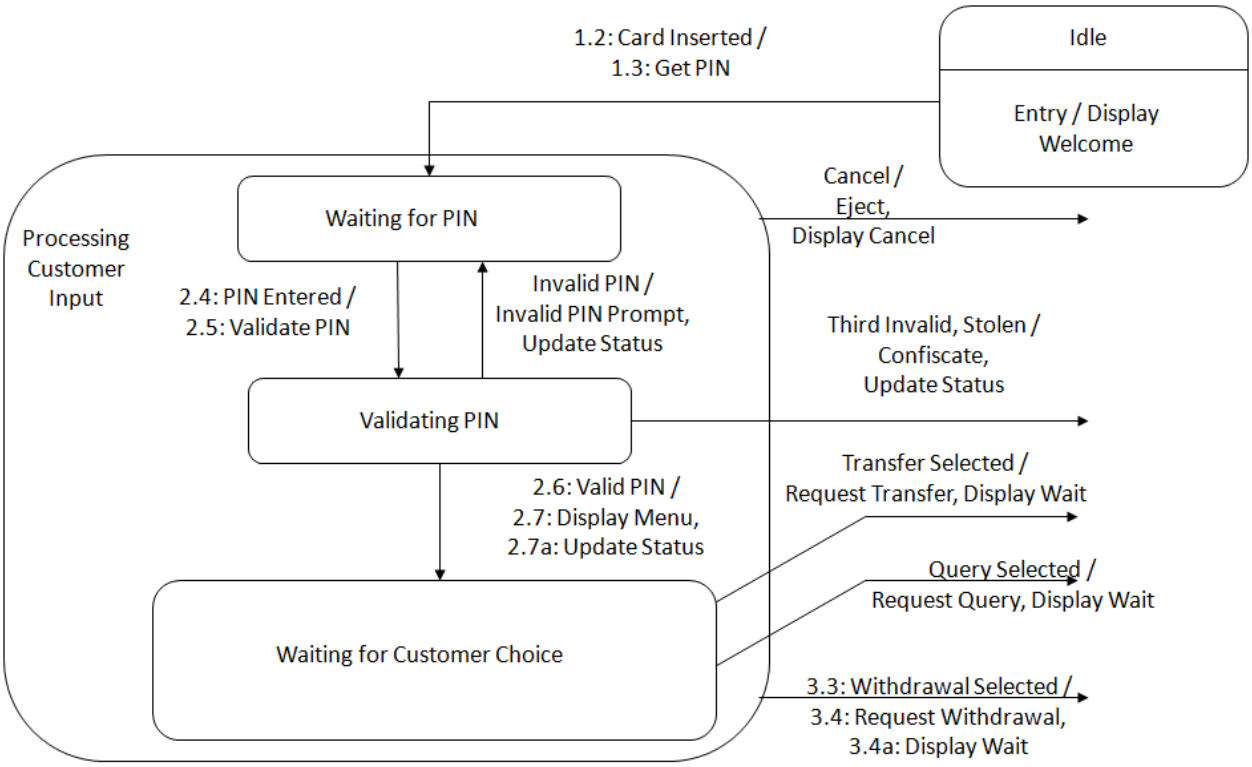
Top-Level ATM Control Statechart



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

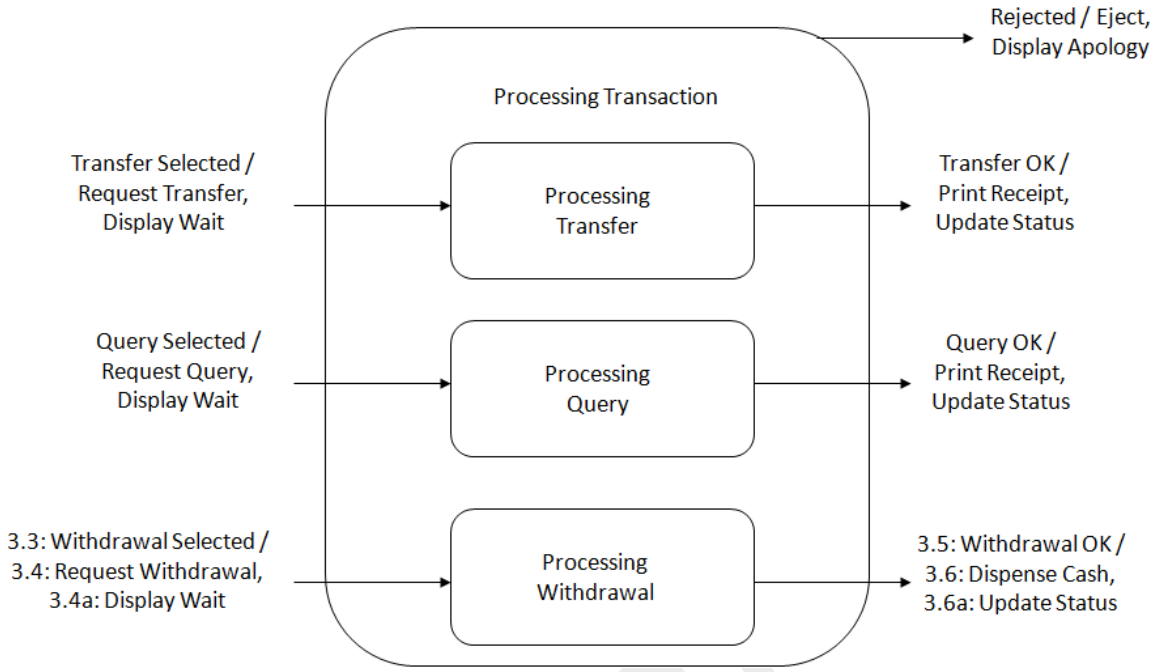
- At system initialization time, given by the event Startup, the ATM transitions from the initial Closed Down state to Idle state. The event Display Welcome message is triggered on entry into idle state. In Idle state, the ATM is for a customer-initiated event

ATM Control Statechart: Processing Customer Input Superstate

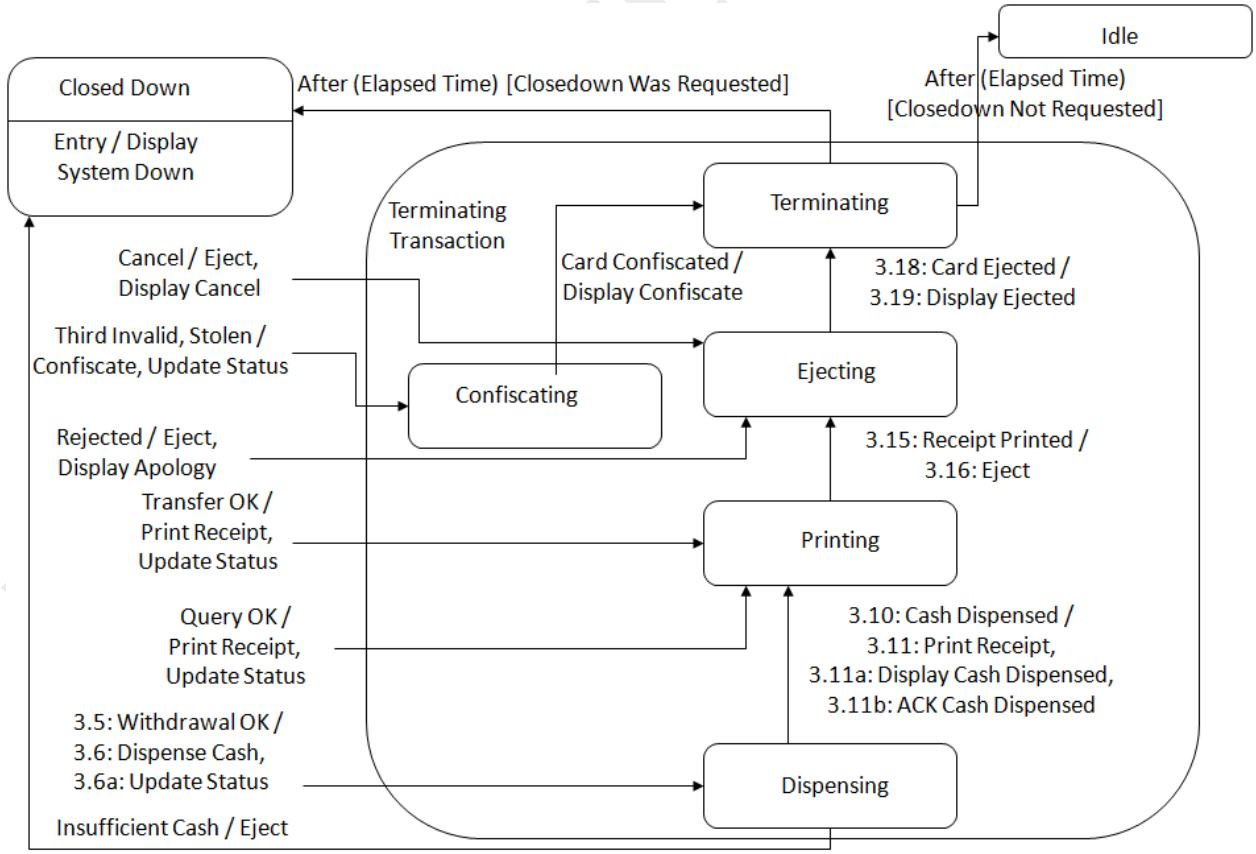


- Processing Customer Input Superstate
  - ✓ The Processing Customer Input superstate is decomposed into three substates
  - ✓ Waiting for PIN
  - ✓ Validating PIN
  - ✓ Waiting for Customer Choice
- Waiting for PIN Substate
  - ✓ This substate is entered from Idle state when the customer inserts the 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

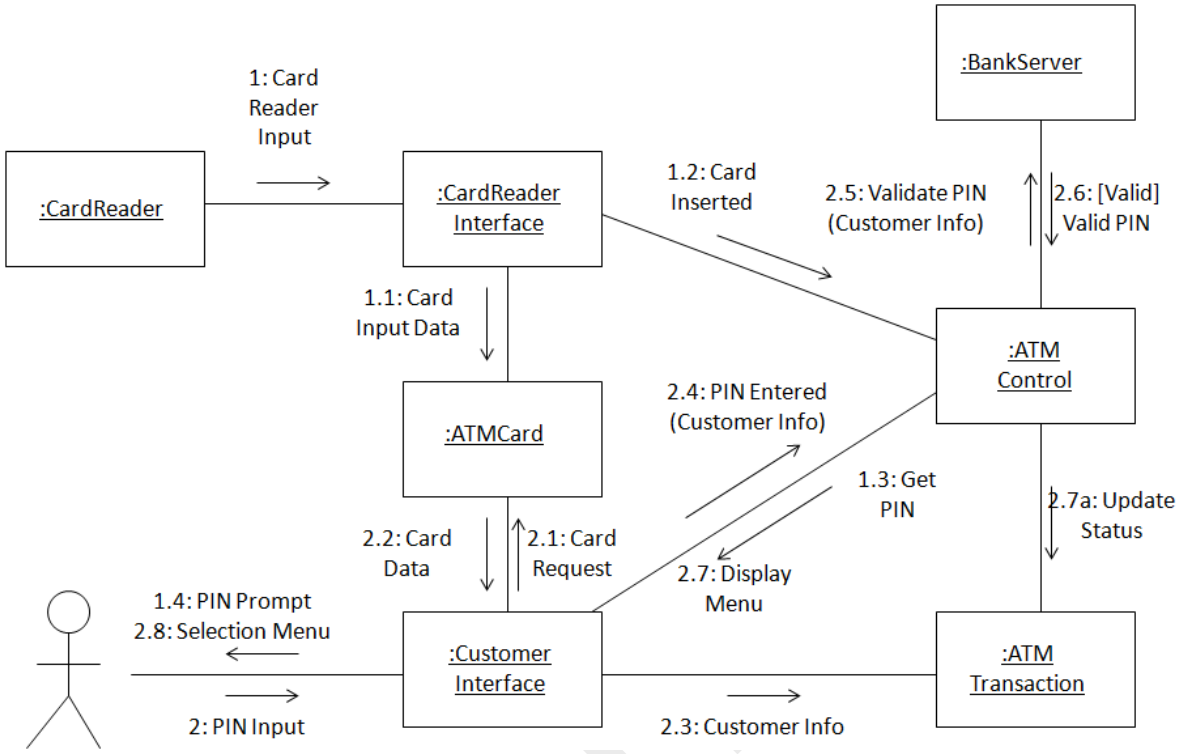
Interaction Diagrams

- 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

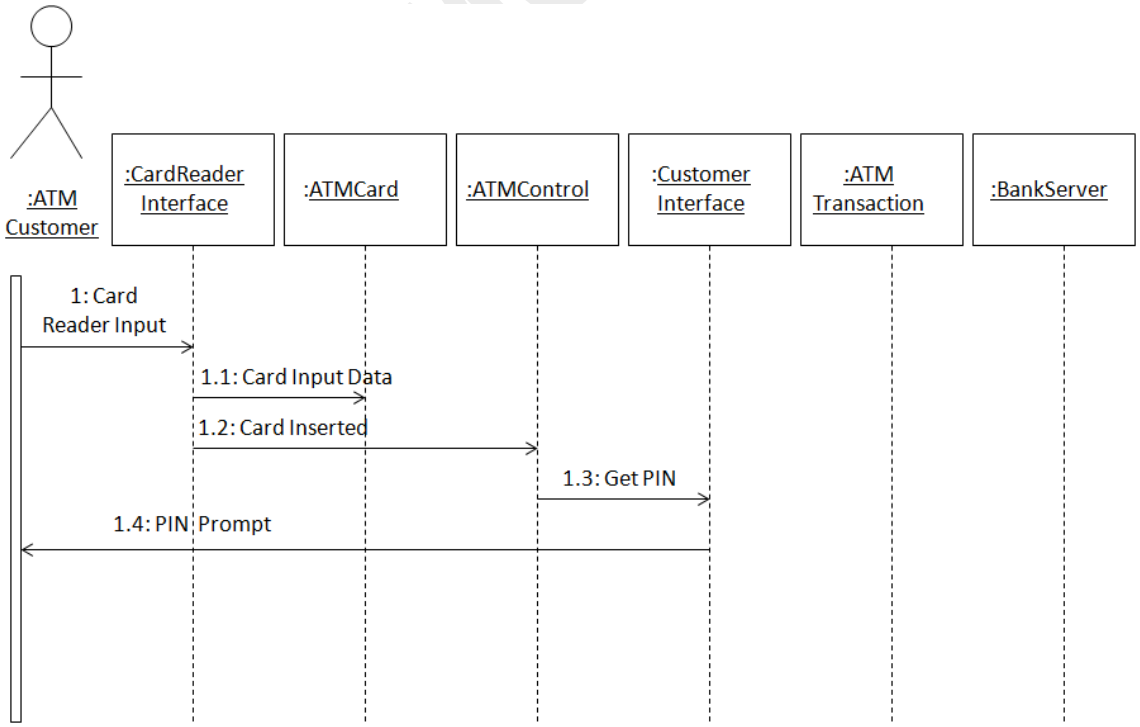
Hints and Tips on Interaction Diagrams

- 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

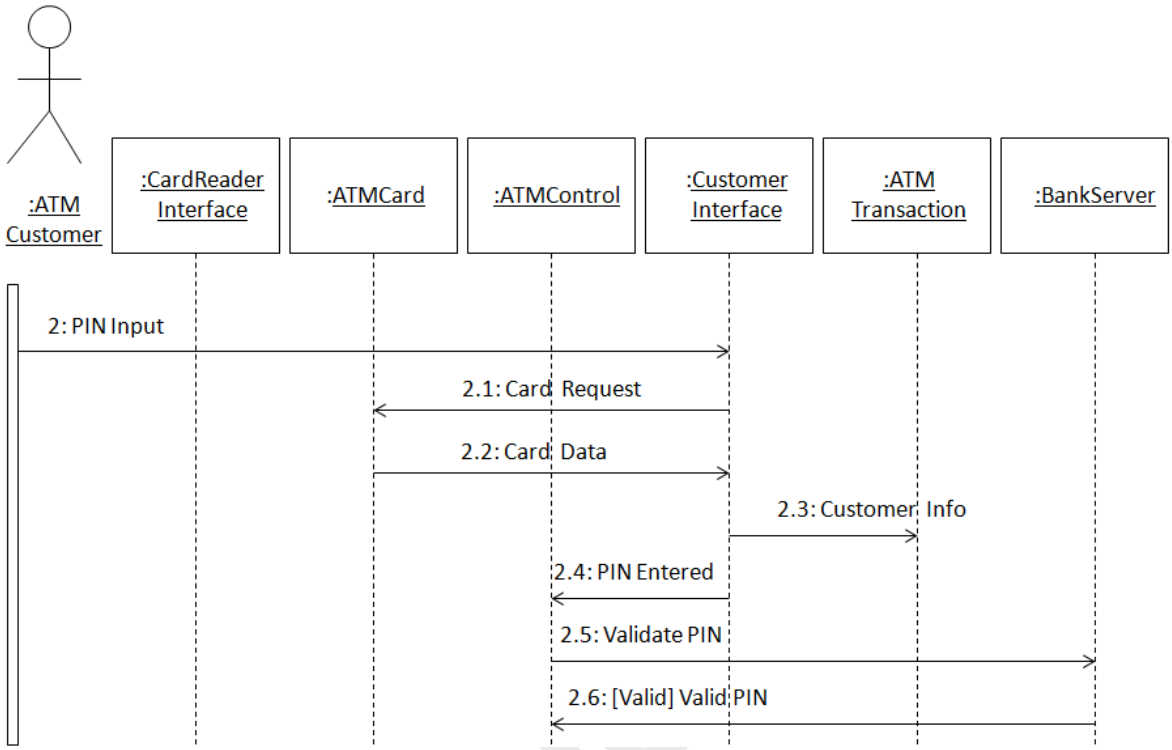
Collaboration Diagram: ATM Client Validate PIN Use Case



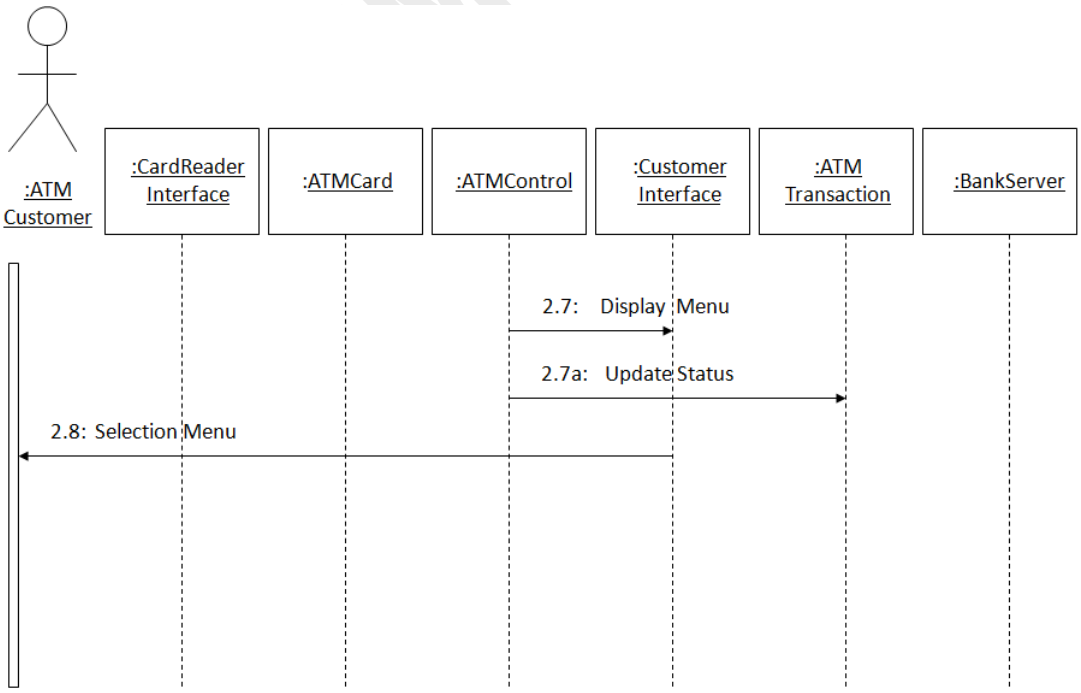
Sequence Diagram: ATM Client Validate PIN Use Case – 1



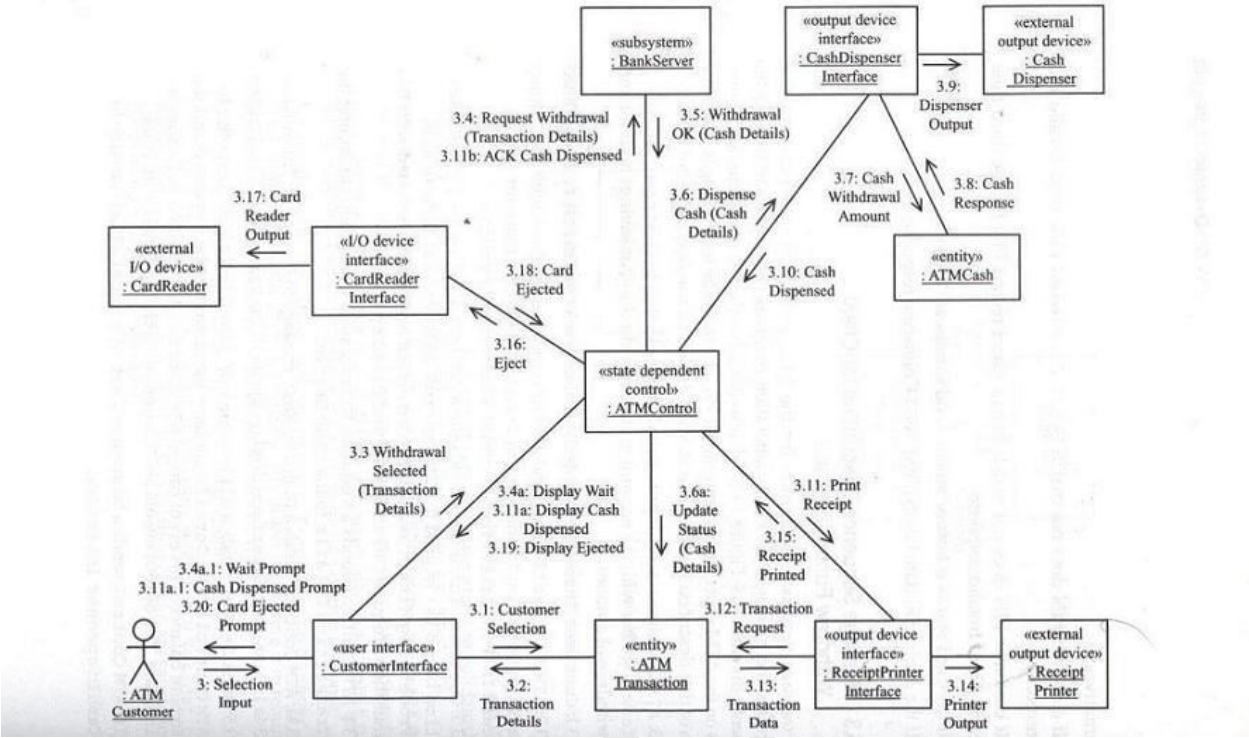
Sequence Diagram: ATM Client Validate PIN Use Case – 2



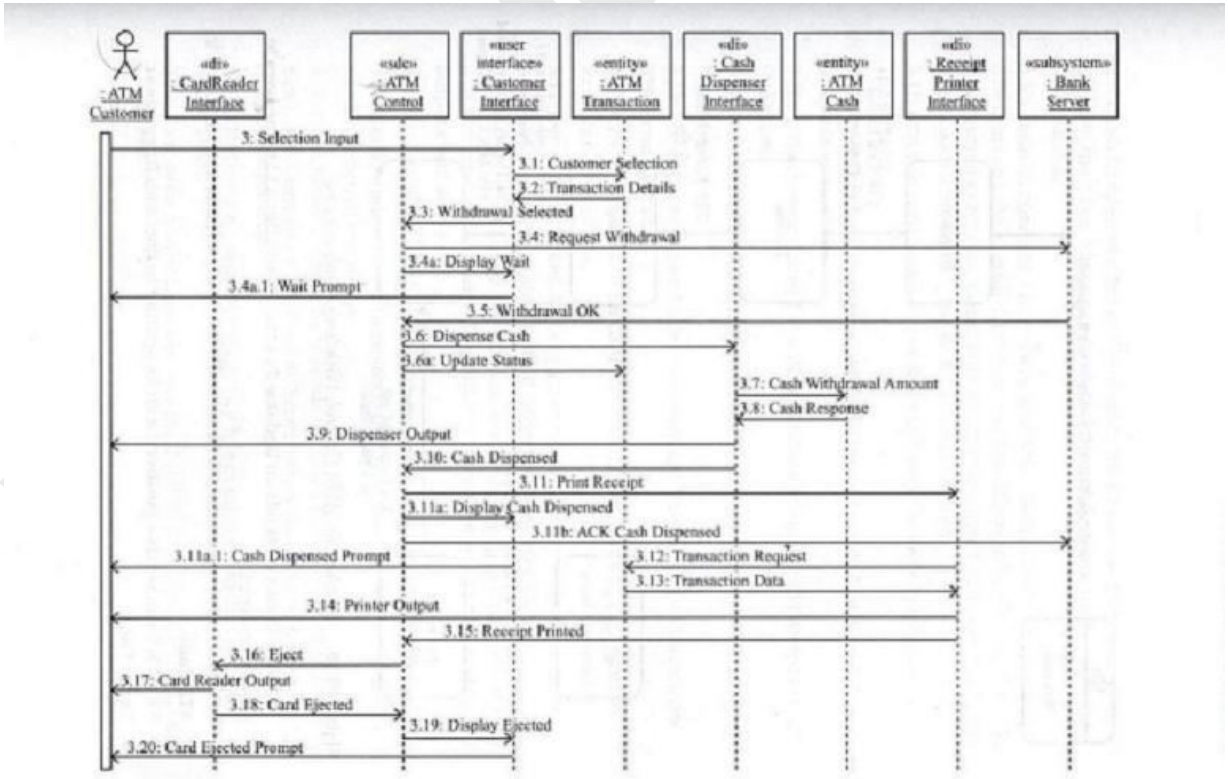
Sequence Diagram: ATM Client Validate PIN Use Case – 3



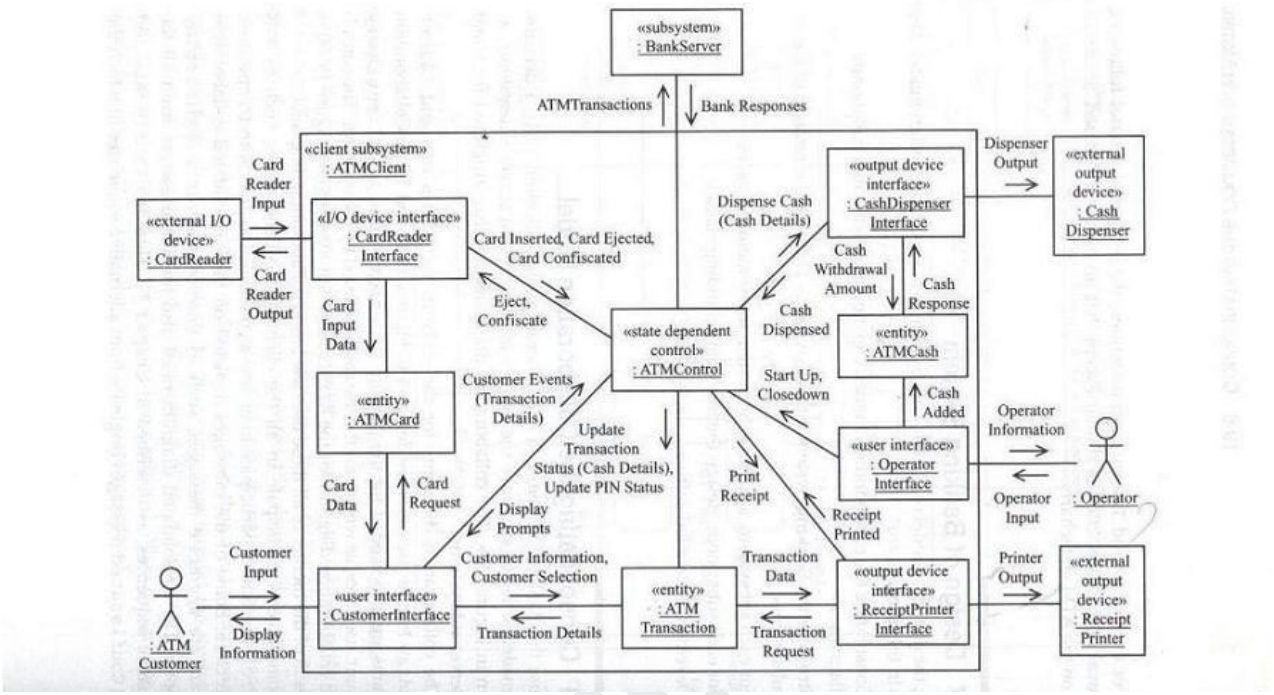
Collaboration Diagram: ATM Client Withdraw Funds Use Case



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.

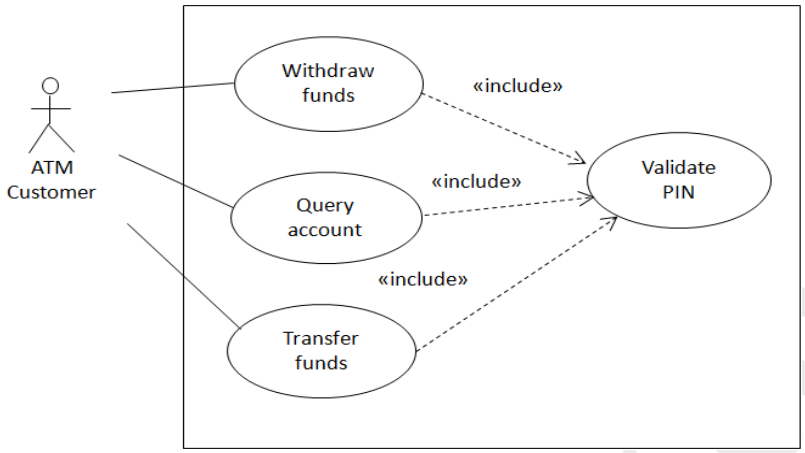
10. If a customer selects query transaction, the system shall prompt the customer to enter account number.
11. If a query transaction is approved, the system shall print a receipt and eject the card. The information contained on the receipt includes transaction number, transaction type, and account balance.
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

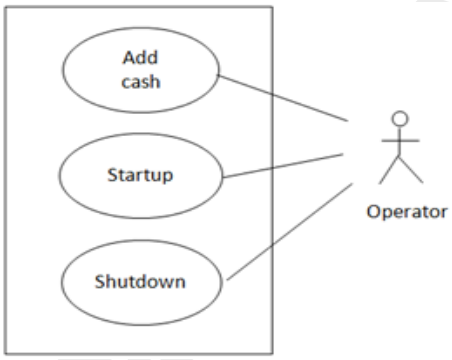
5.1 Use-case model
5.2 Object model
5.3 Functional model
    5.3.1 Data-flow model
    5.3.2 SADT model
5.4 Dynamic model
    5.4.1 Statecharts
    5.4.2 Interaction diagrams

Use Case Model

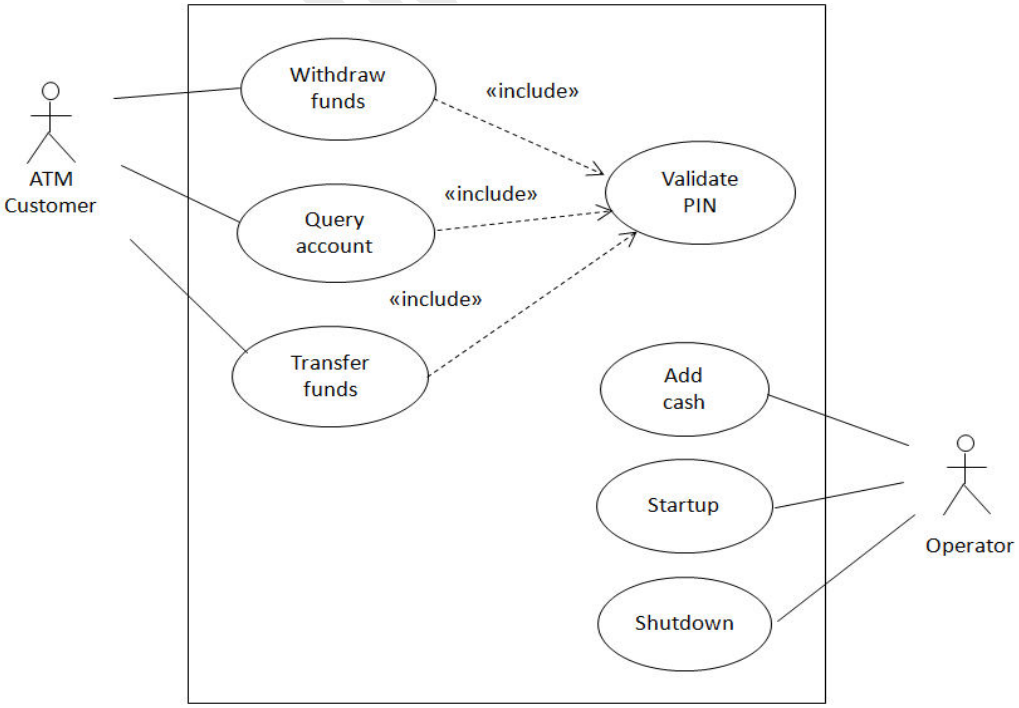
Uses Case Diagram for ATM Customer



Use Case Diagram for ATM Operator

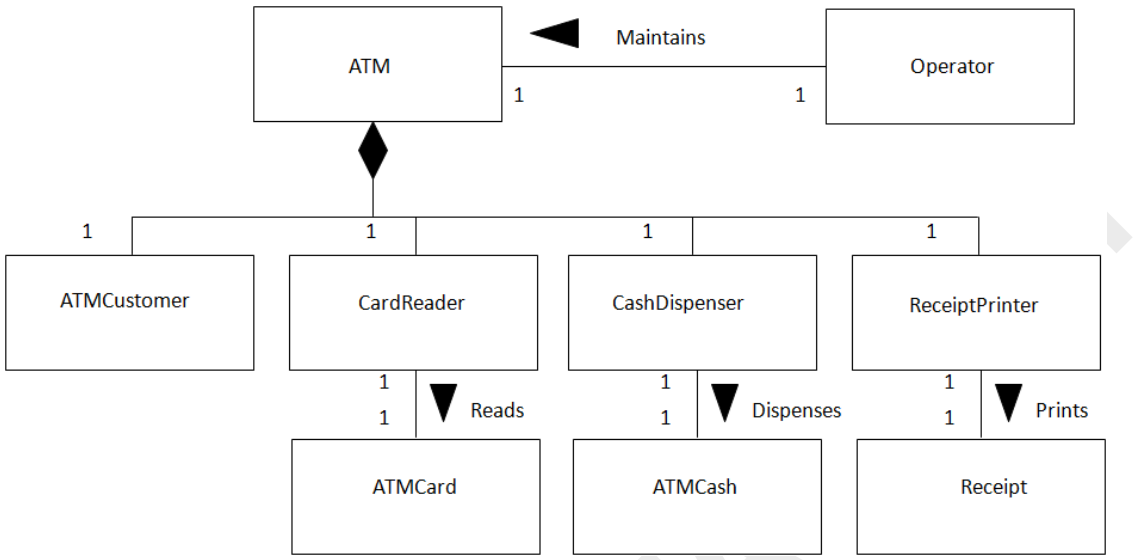


Use Case Diagram for ATM

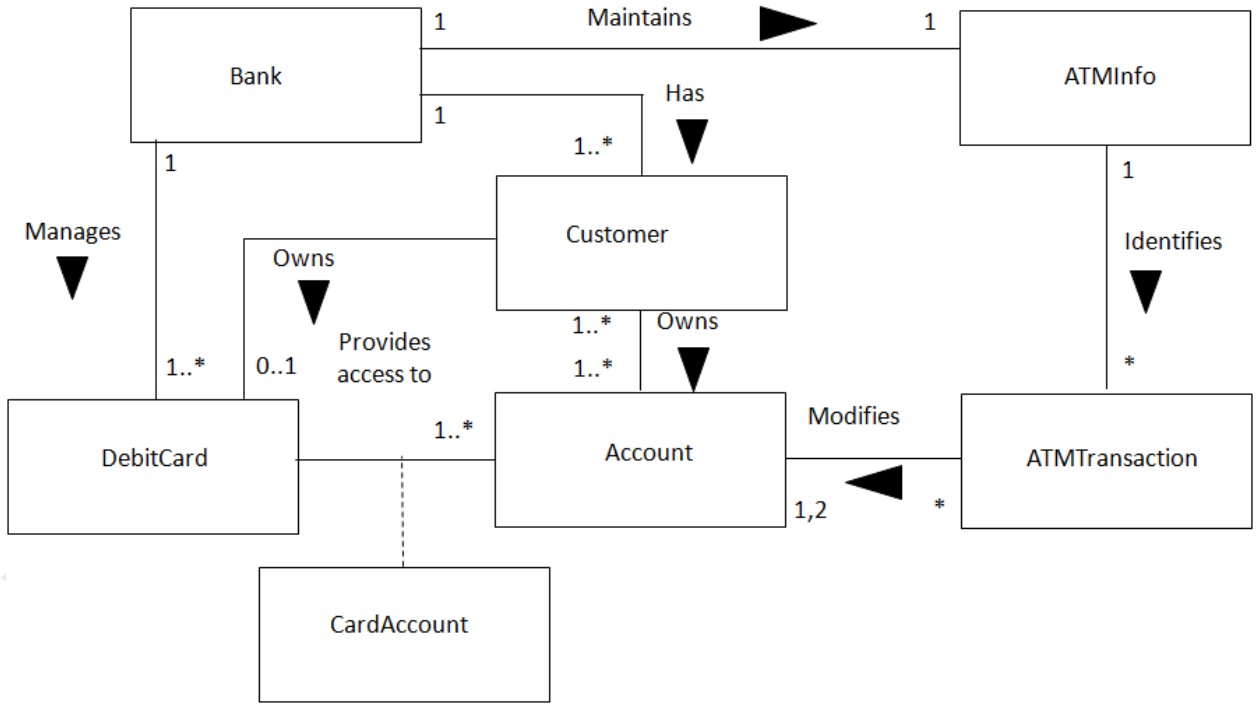


Object Model

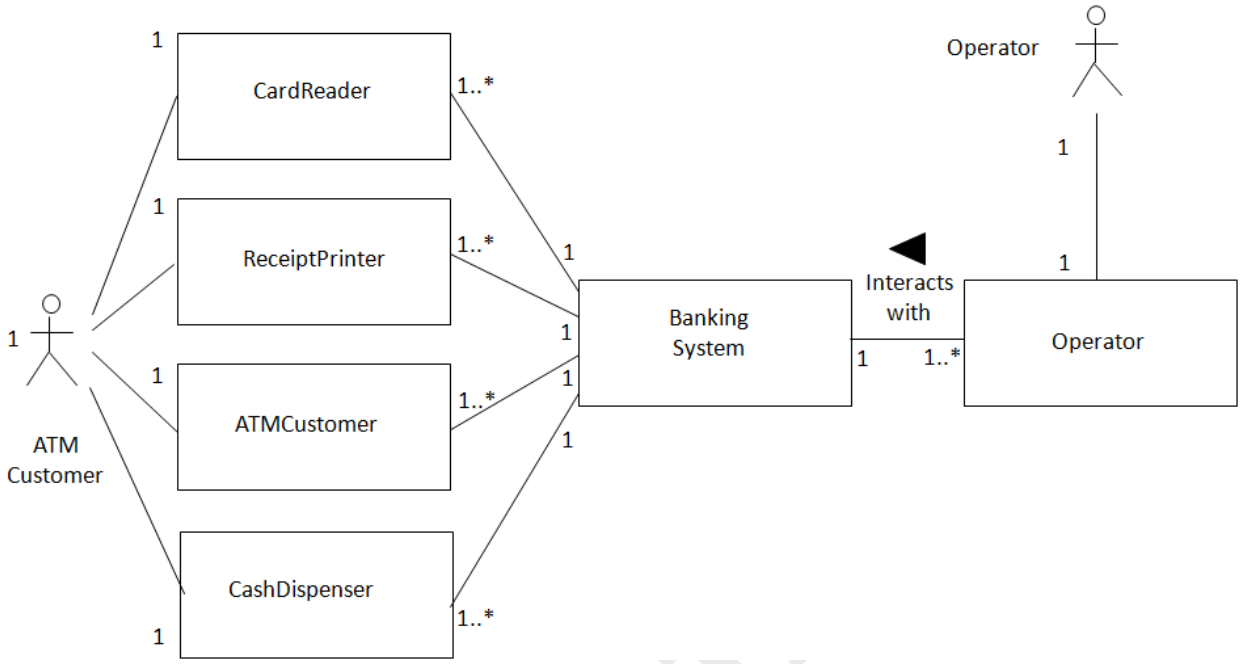
Conceptual Static Model for Problem Domain: Physical Classes



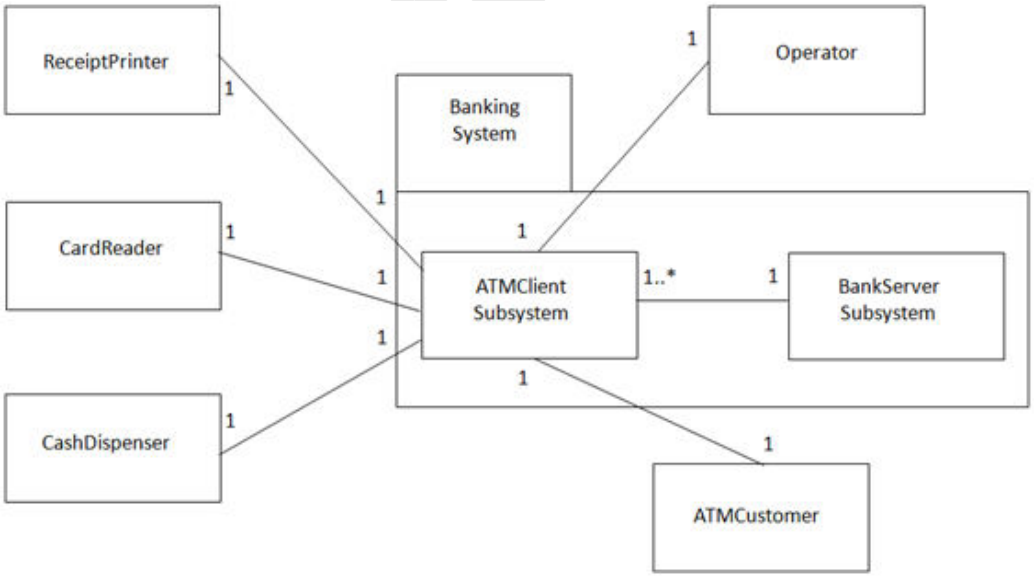
Conceptual Static Model for Problem Domain: Entity Classes



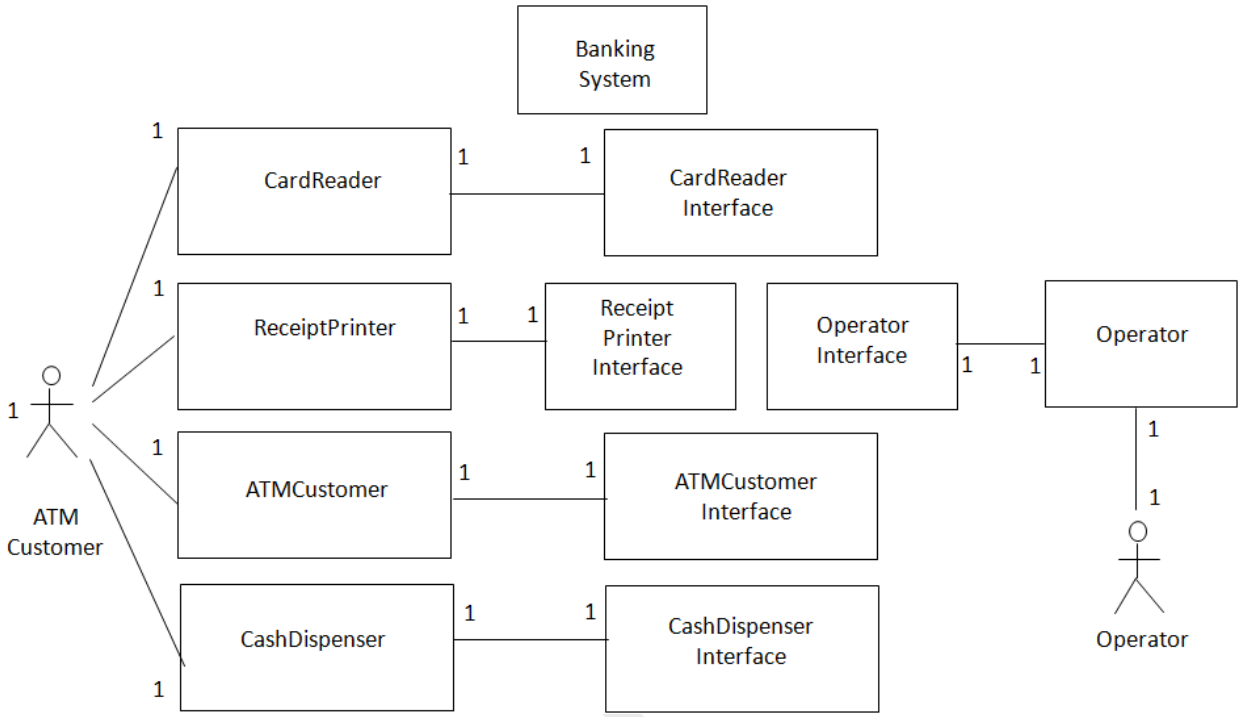
Banking System Context Class Diagram



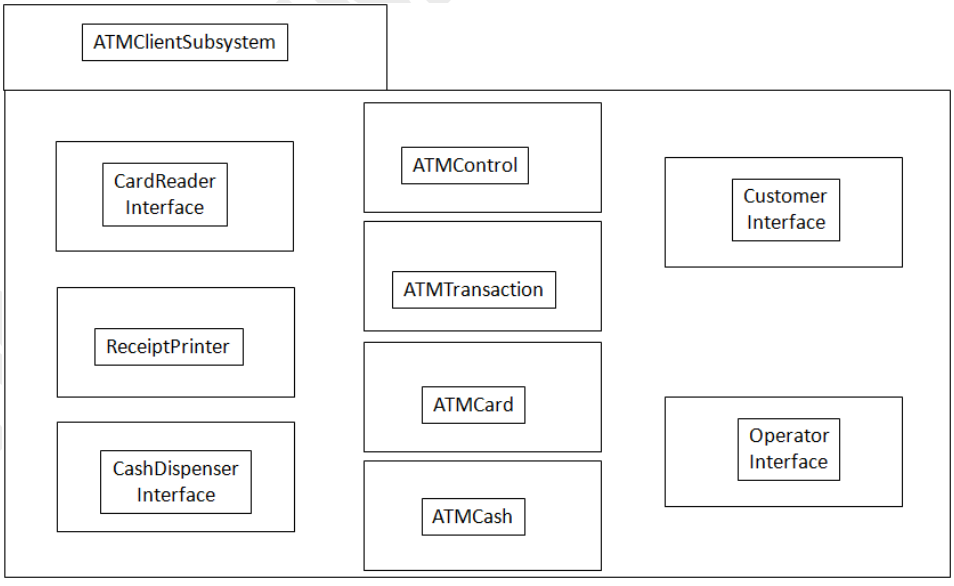
Banking System: Major Subsystems



Banking System External Classes and Interfaces Classes

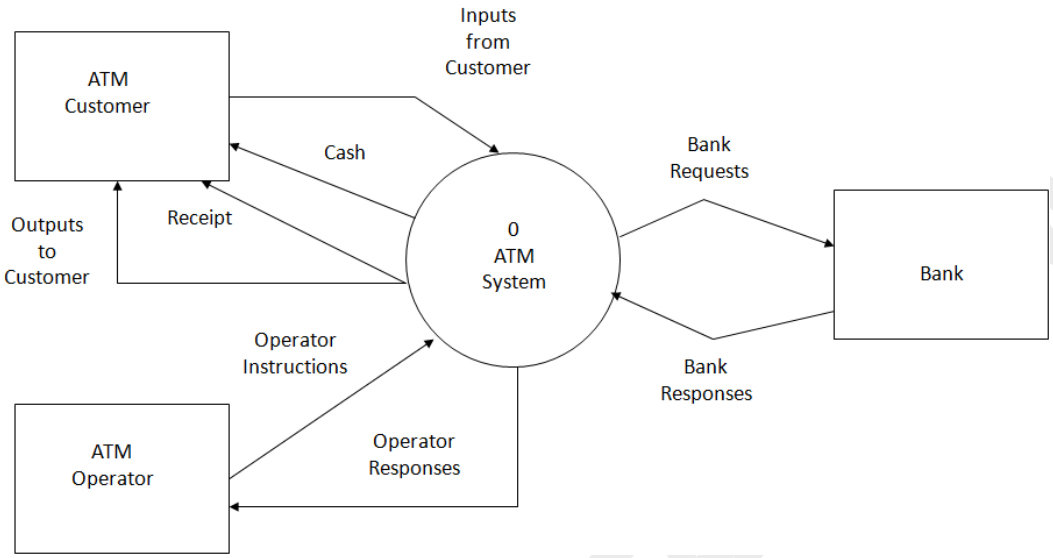


ATM Client Subsystem Classes

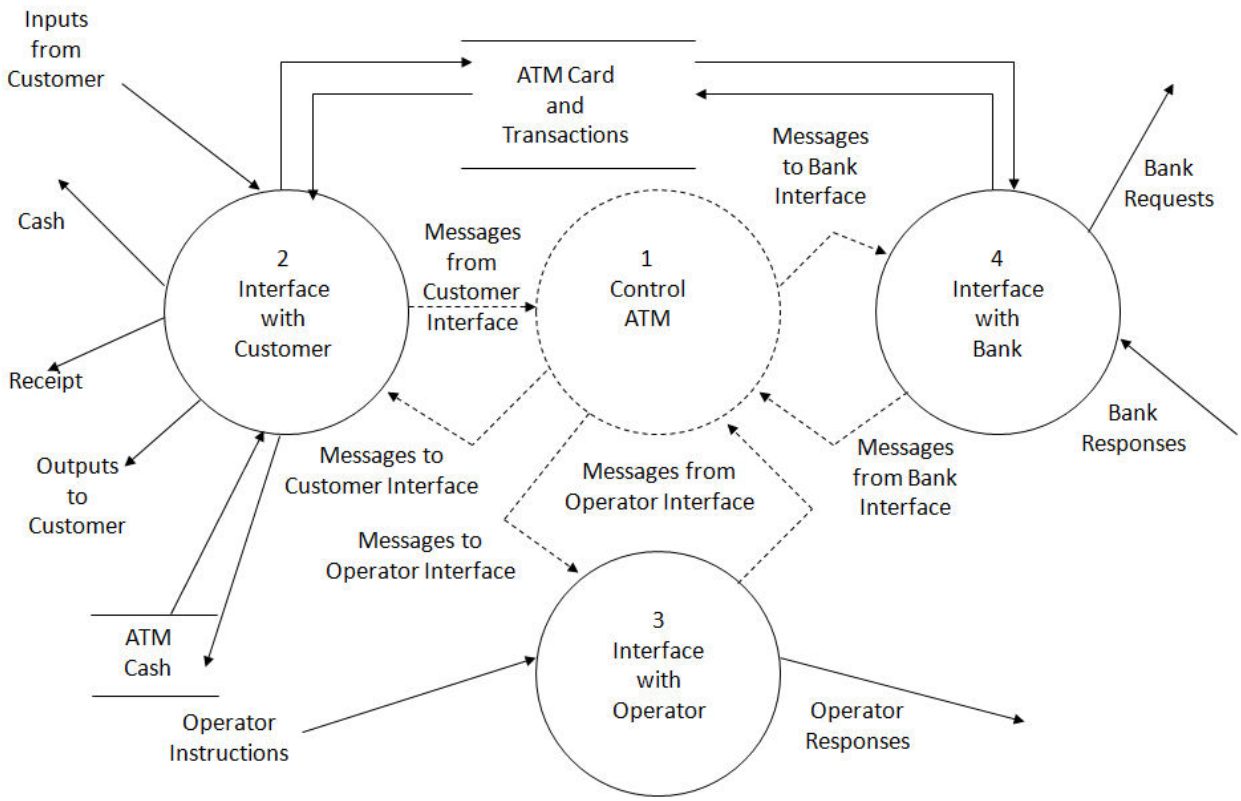


Data Flow Diagrams

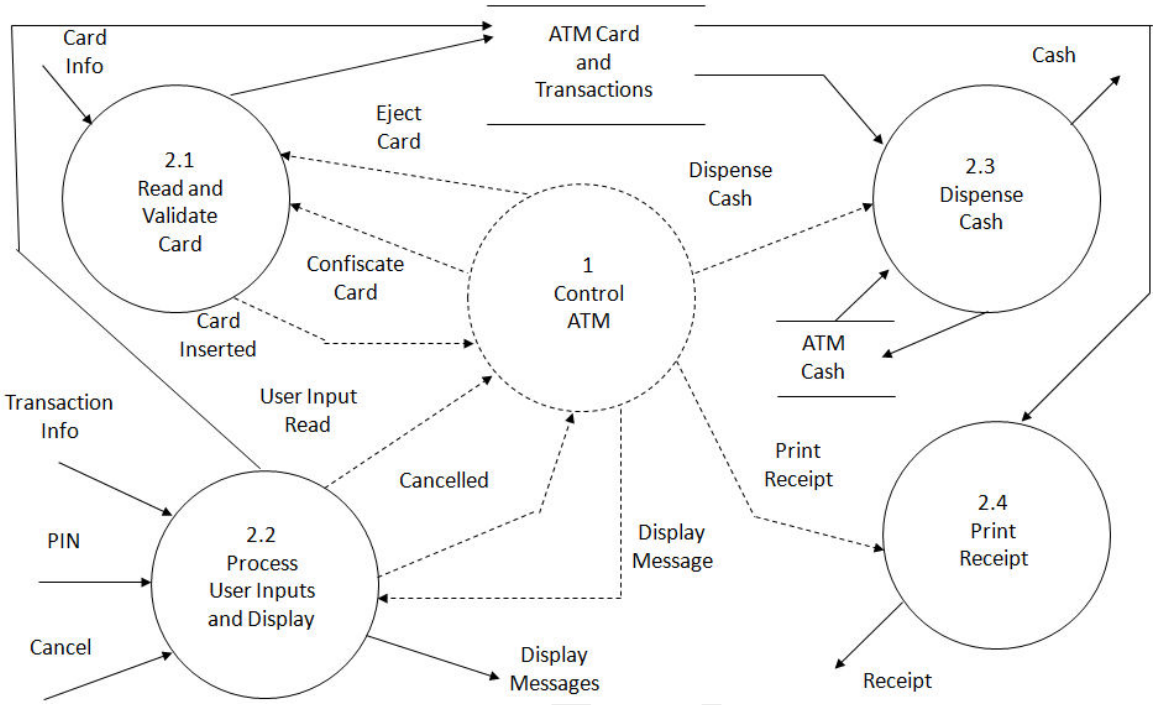
System Context Diagram



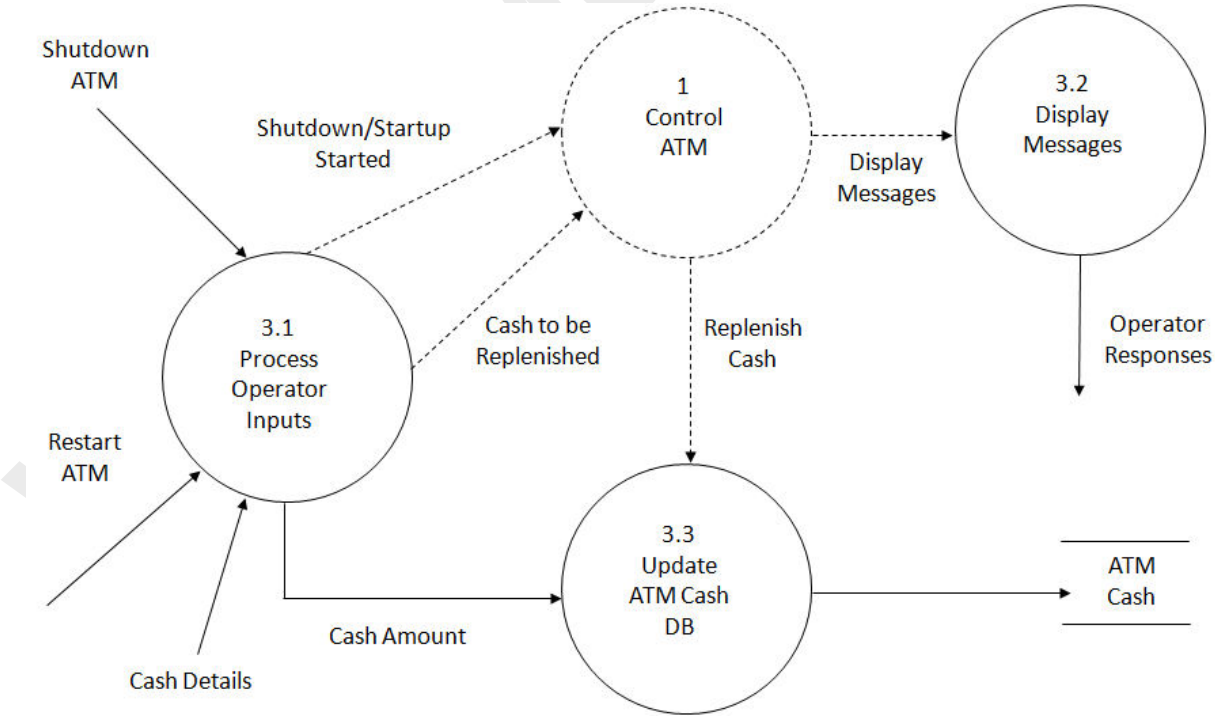
Data Flow Diagram – Level 1



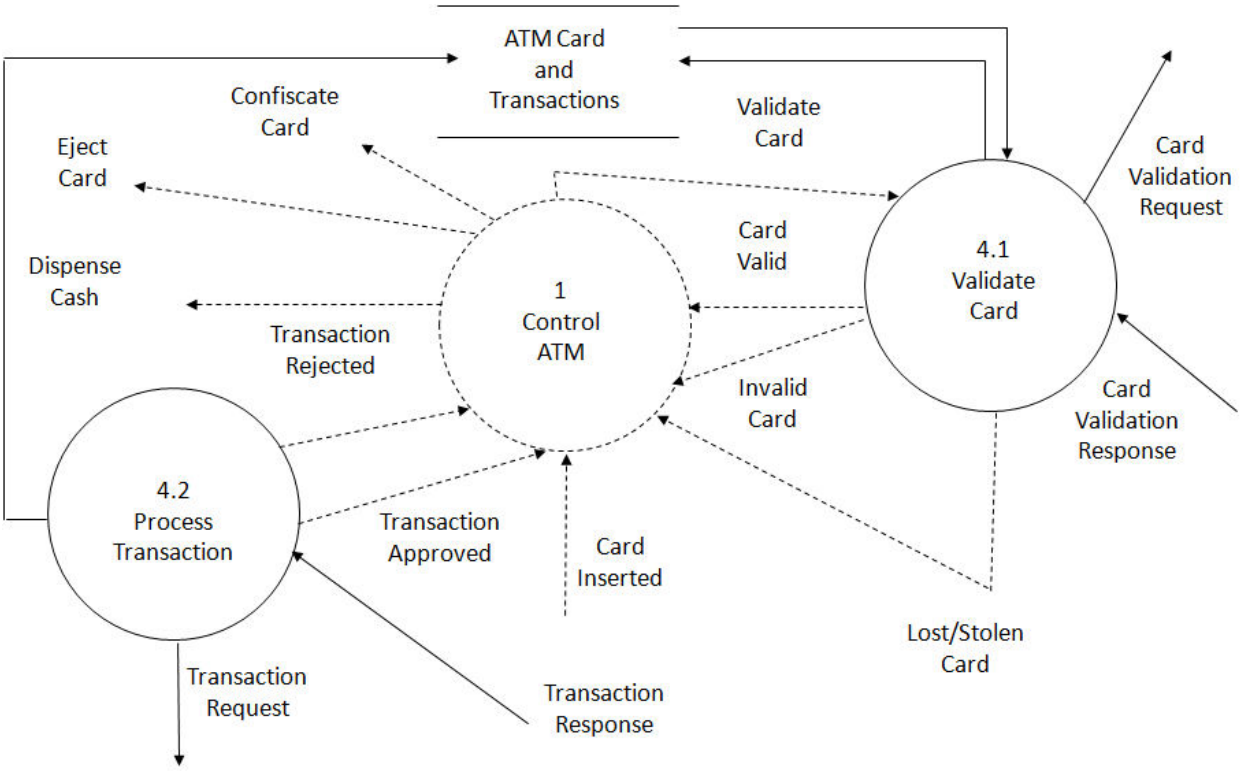
Level 2 DFD: Interface with Customer



Level 2 DFD: Interface with Operator

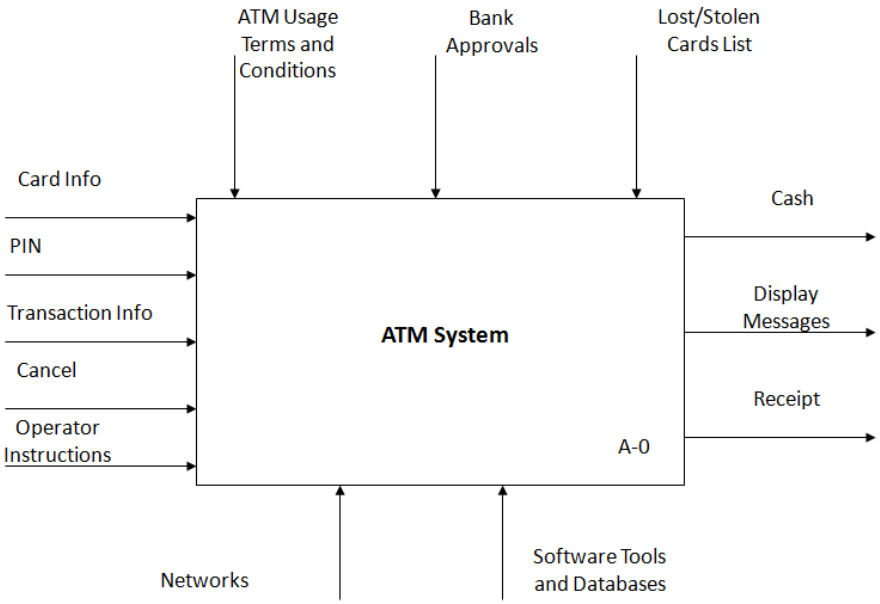


Level 2 DFD: Interface with Bank

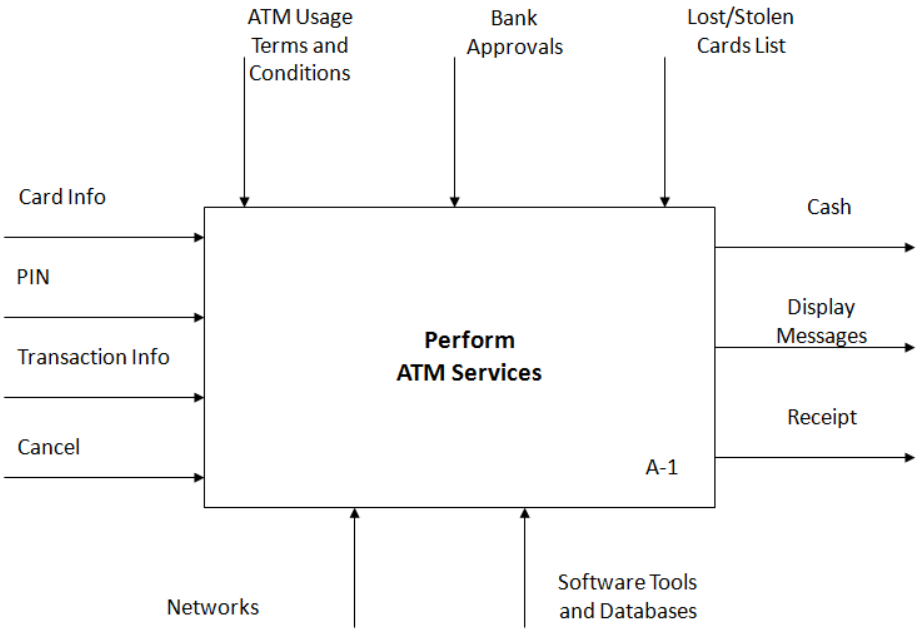


SADT Diagrams

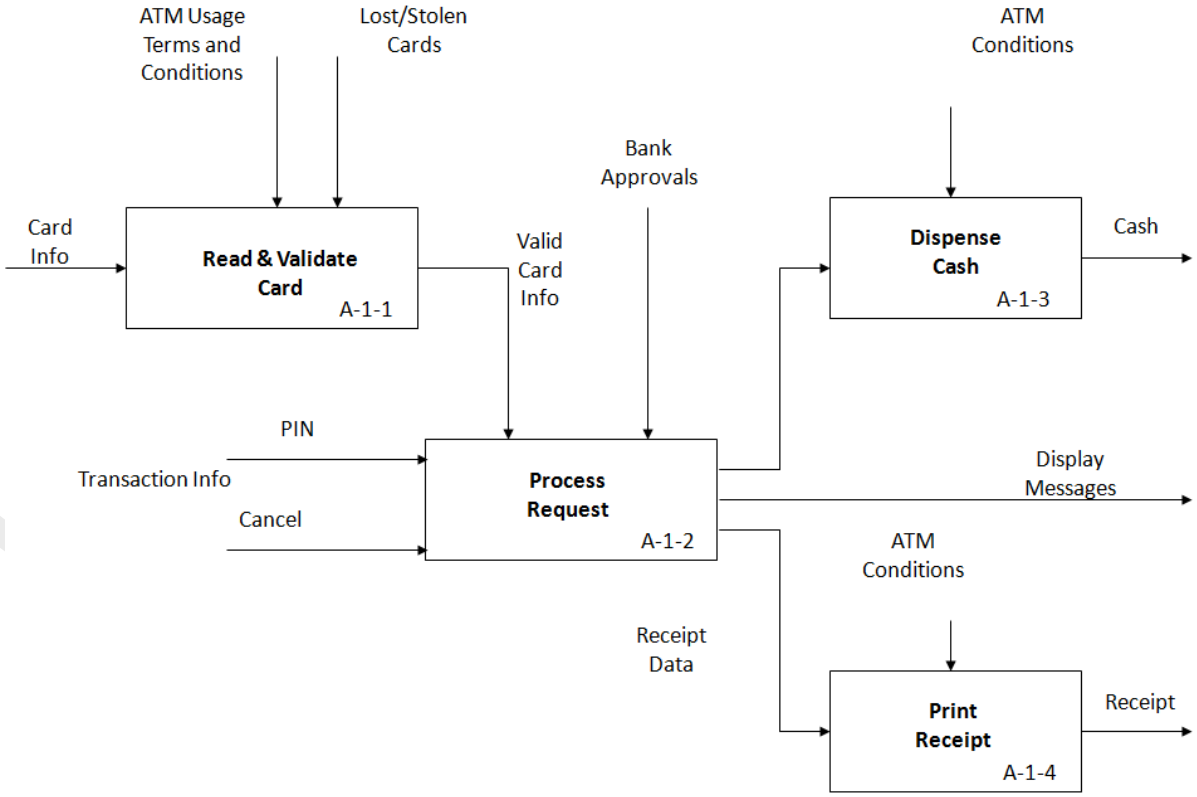
Banking System Context Diagram



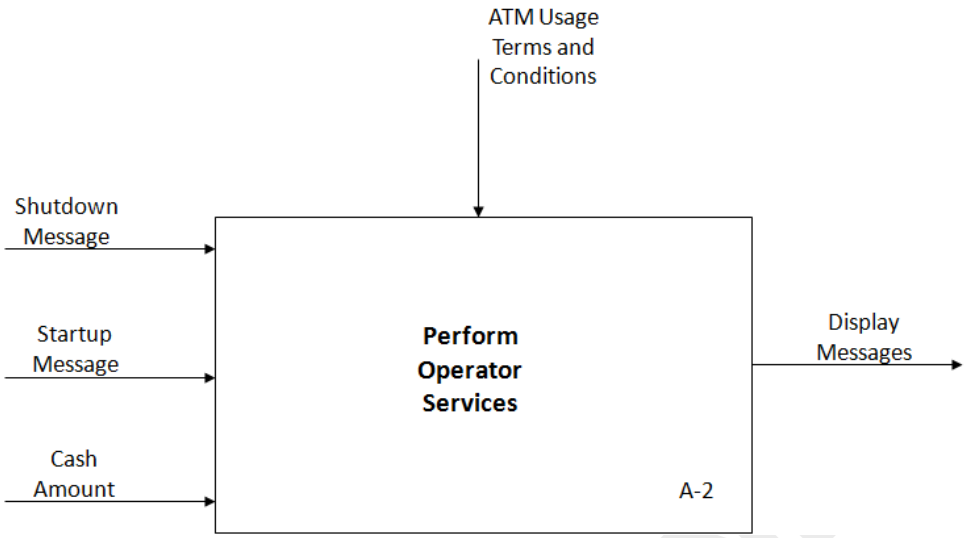
SADT Level 1 Diagram



SADT Level 2

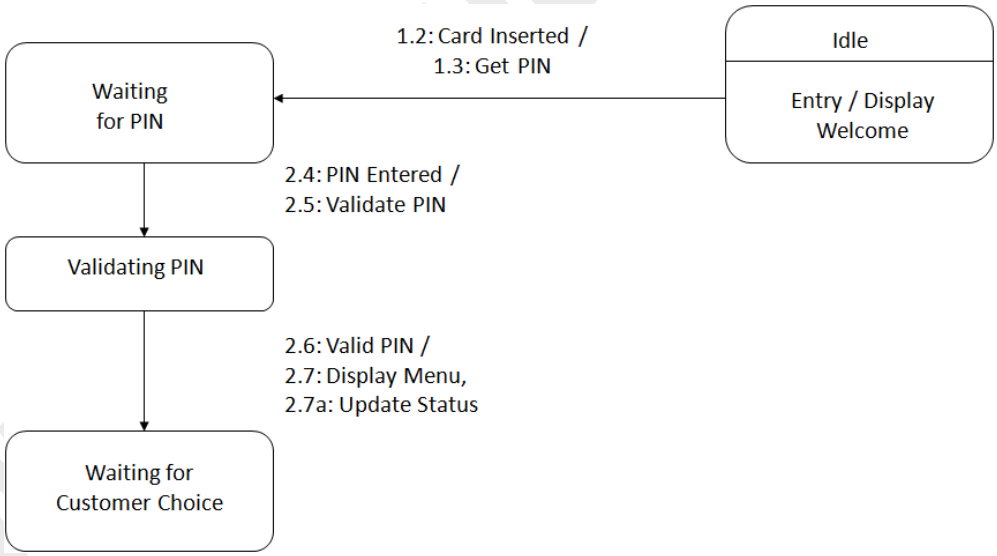


SADT Level 1 Diagram

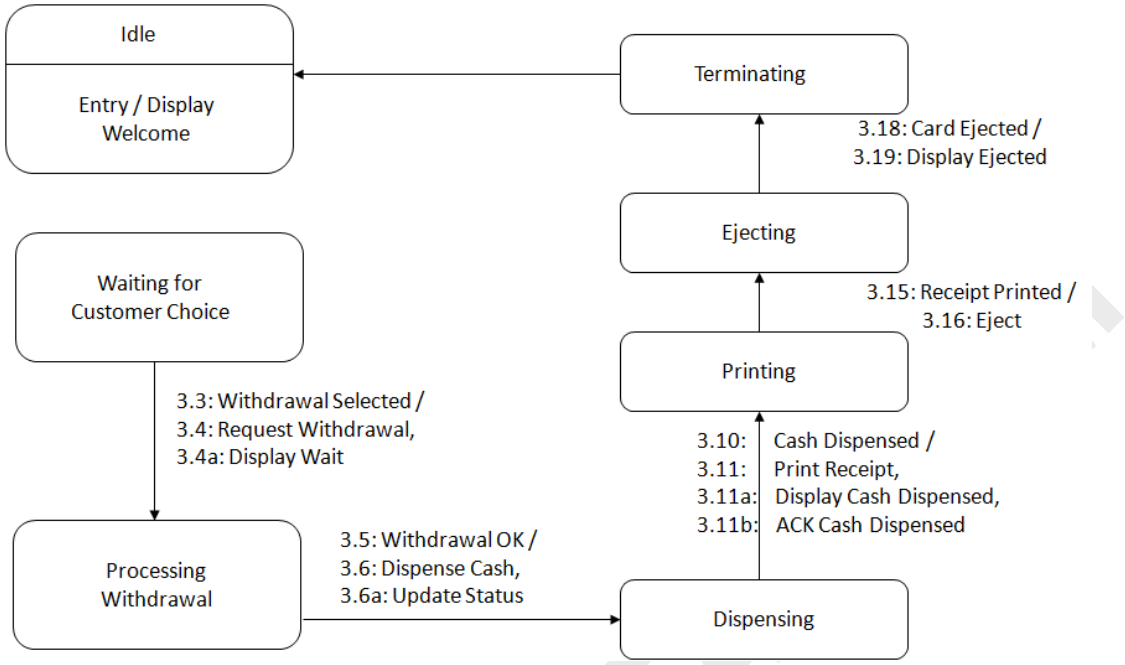


Statecharts Diagrams

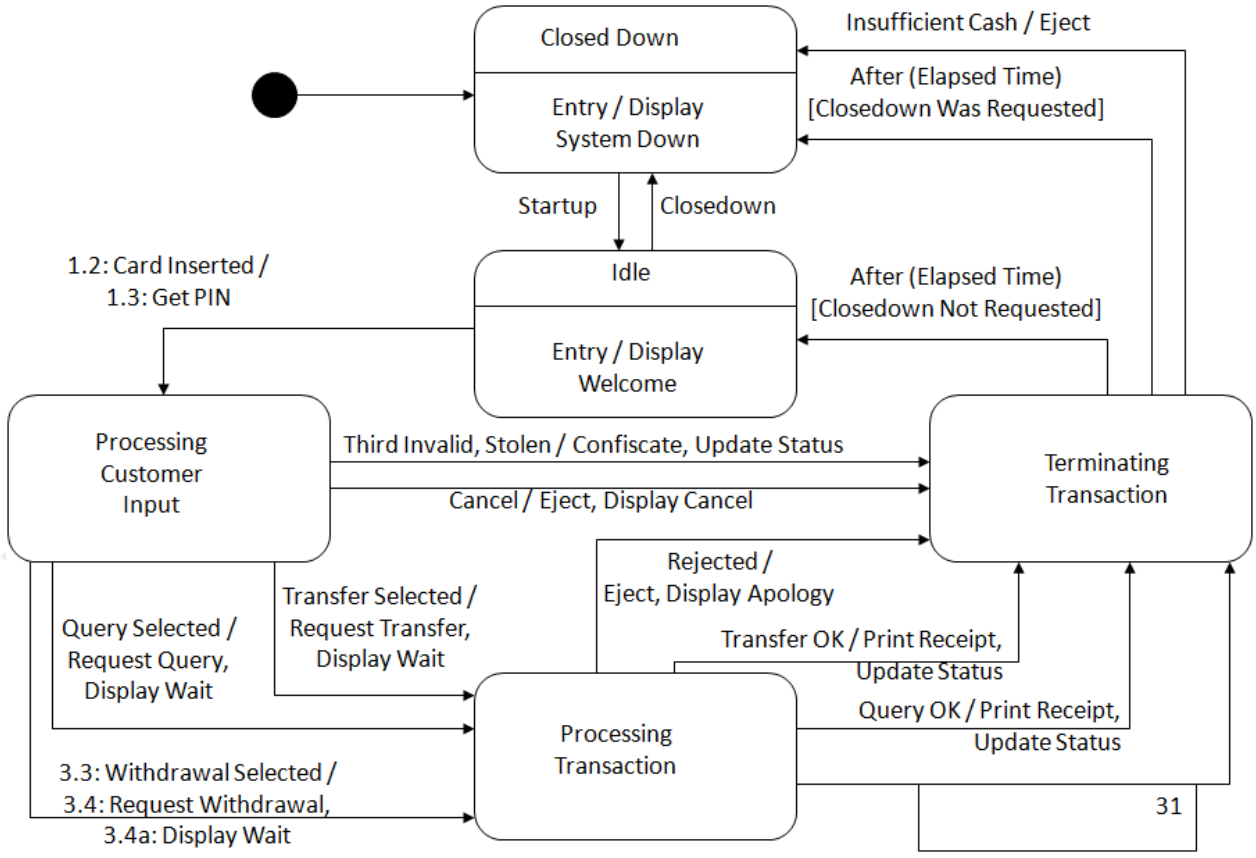
Statechart for ATM Control: Validate PIN Use Case



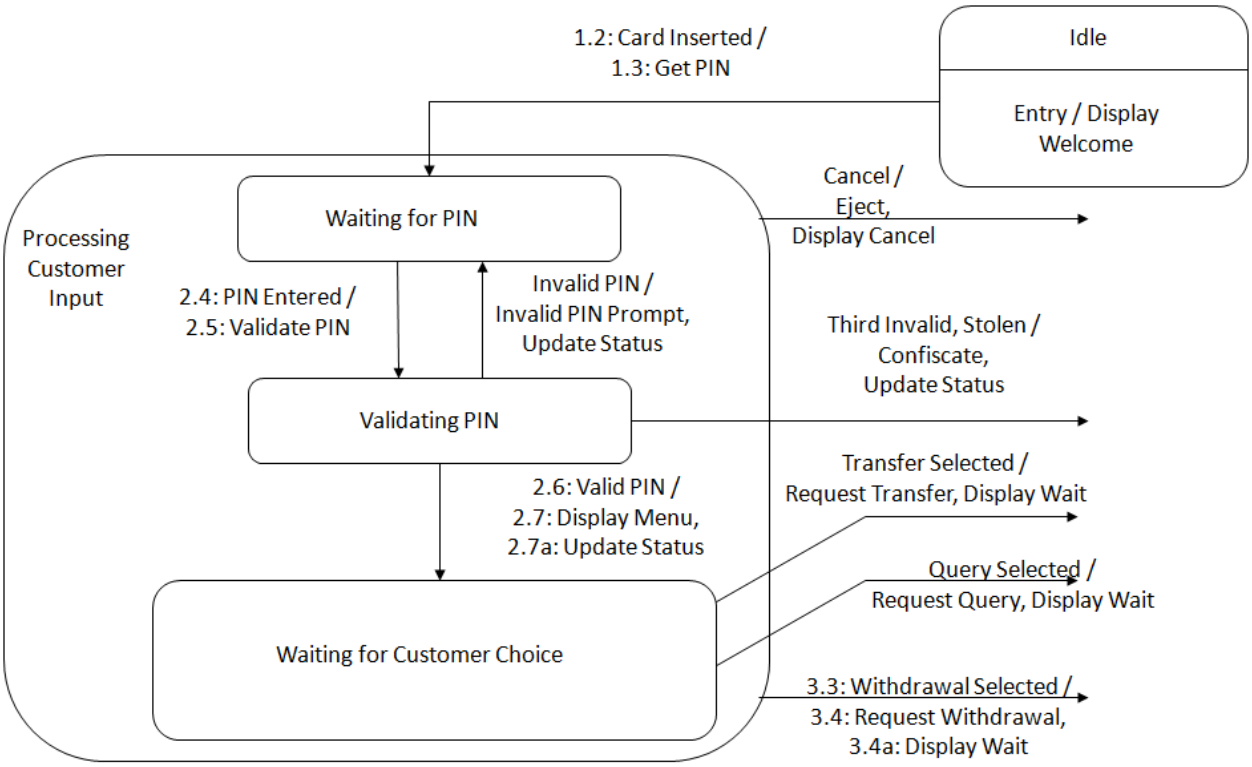
Statechart for ATM Control: Withdraw Funds Use Case



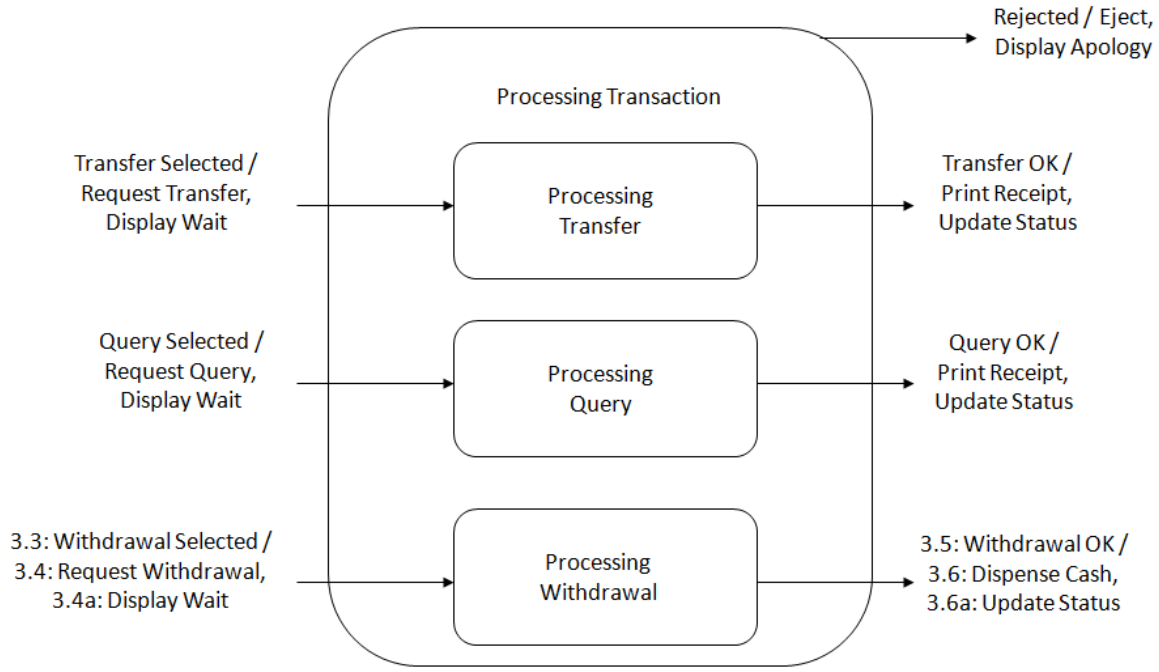
Top-Level ATM Control Statechart



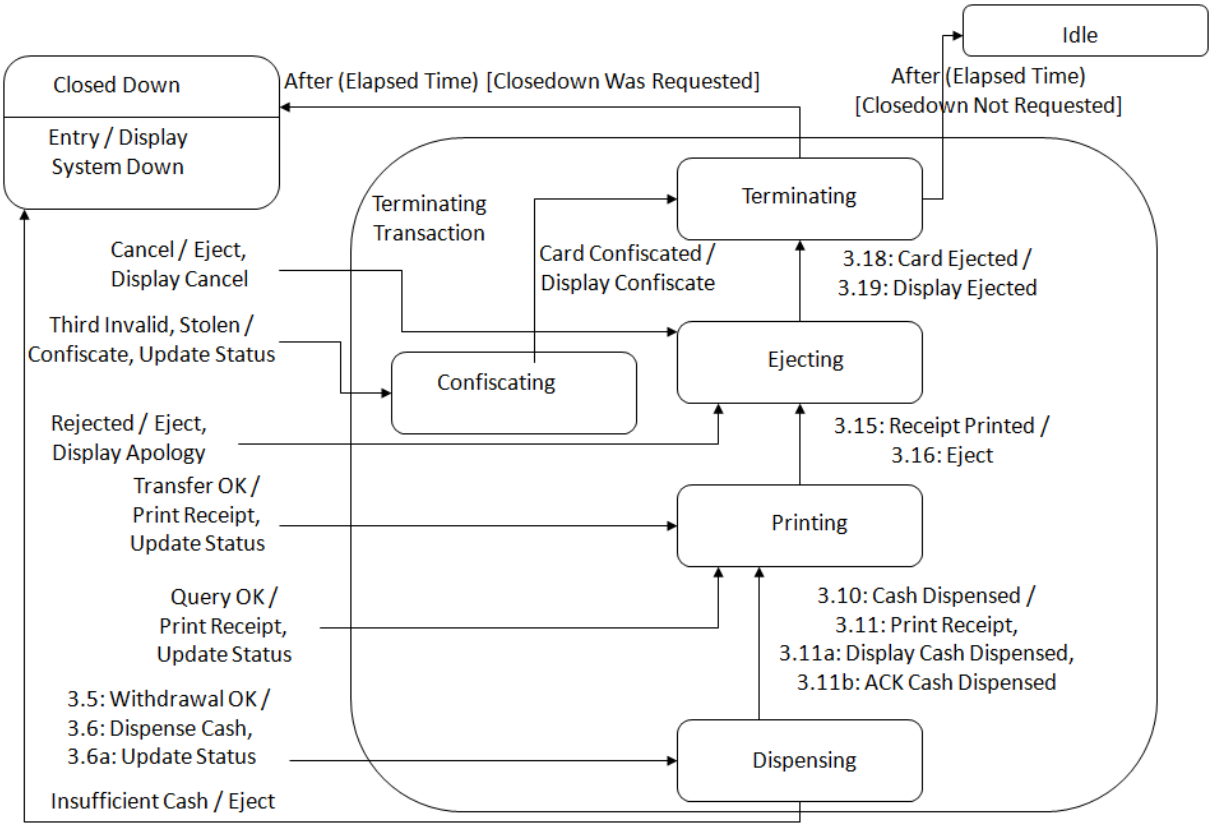
ATM Control Statechart: Processing Customer Input Superstate



ATM Control Statechart: Processing Transaction Superstate

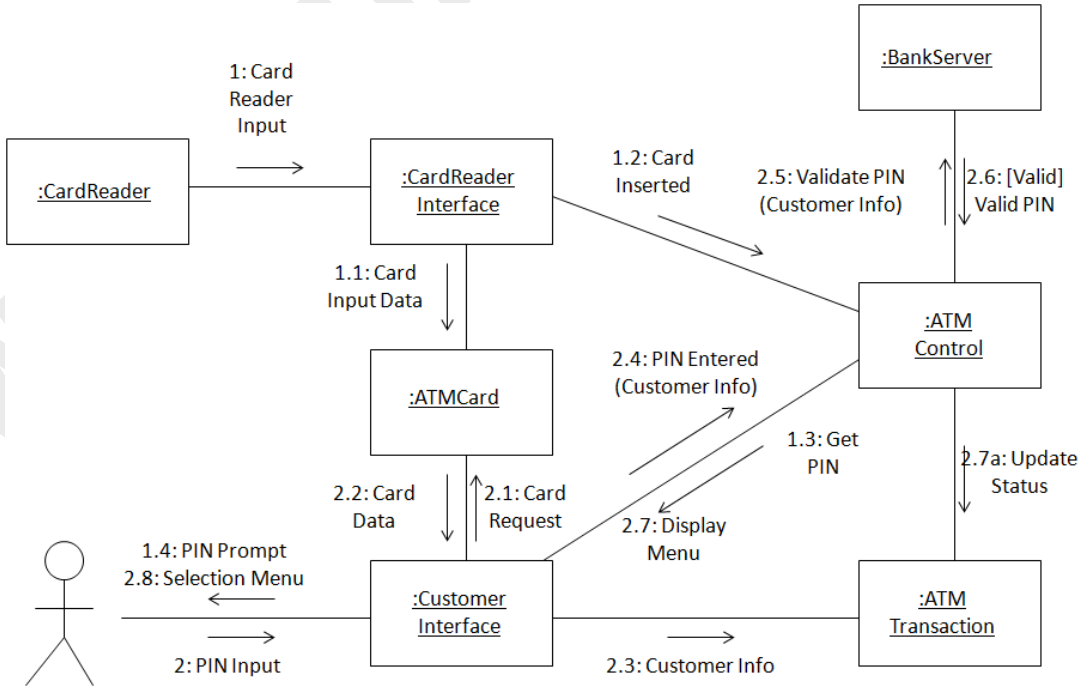


ATM Control Statechart: Terminating Transaction Superstate

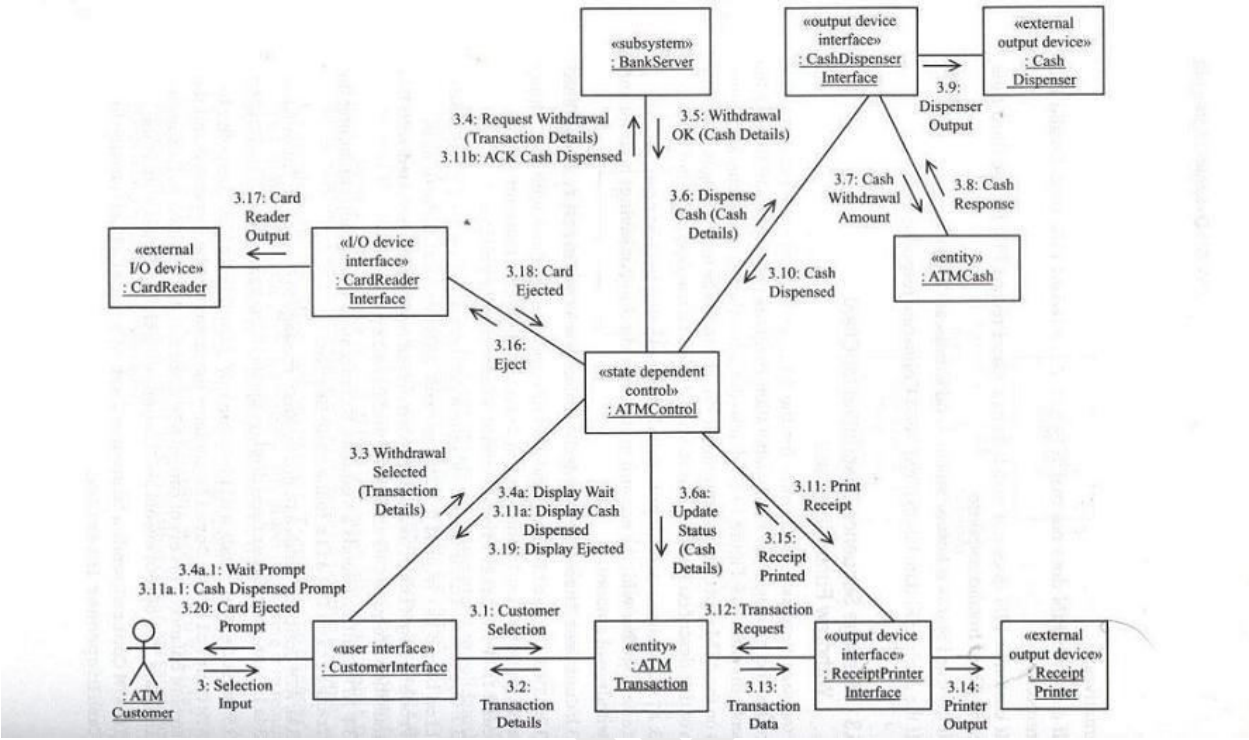


Collaboration Diagrams

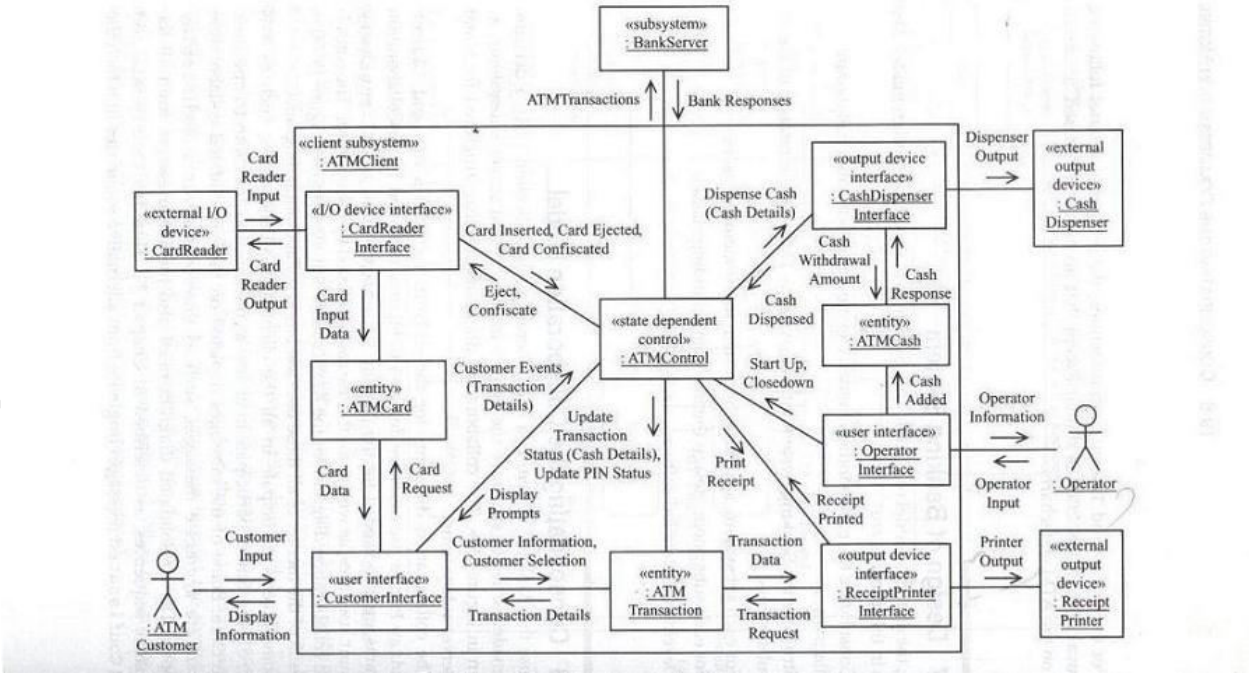
Collaboration Diagram: ATM Client Validate PIN Use Case



Collaboration Diagram: ATM Client Withdraw Funds Use Case

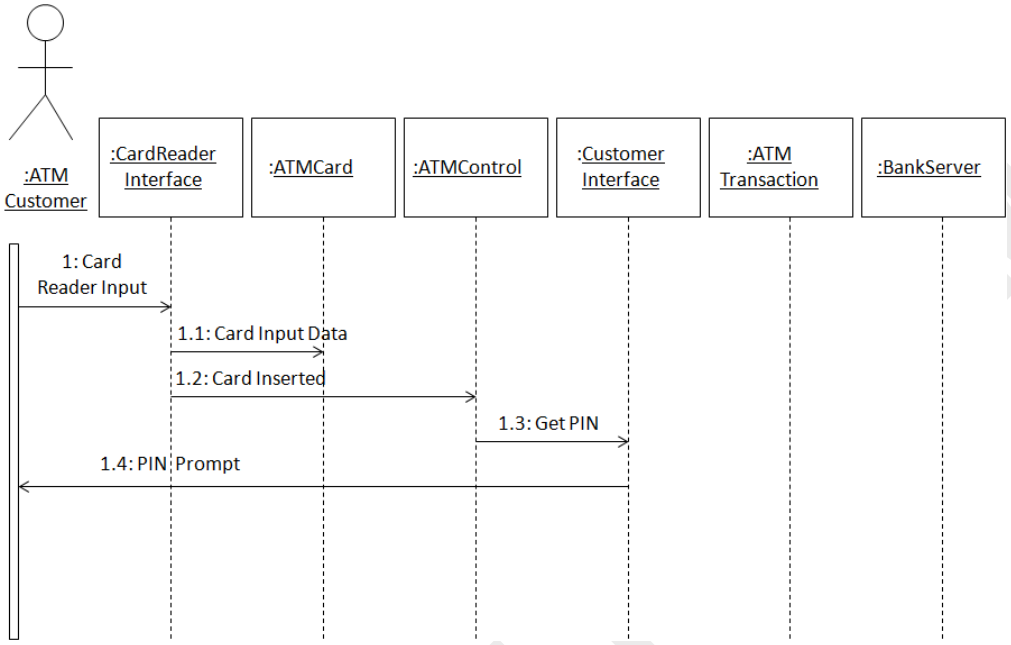


Consolidated Collaboration Diagram for ATM Client Subsystem

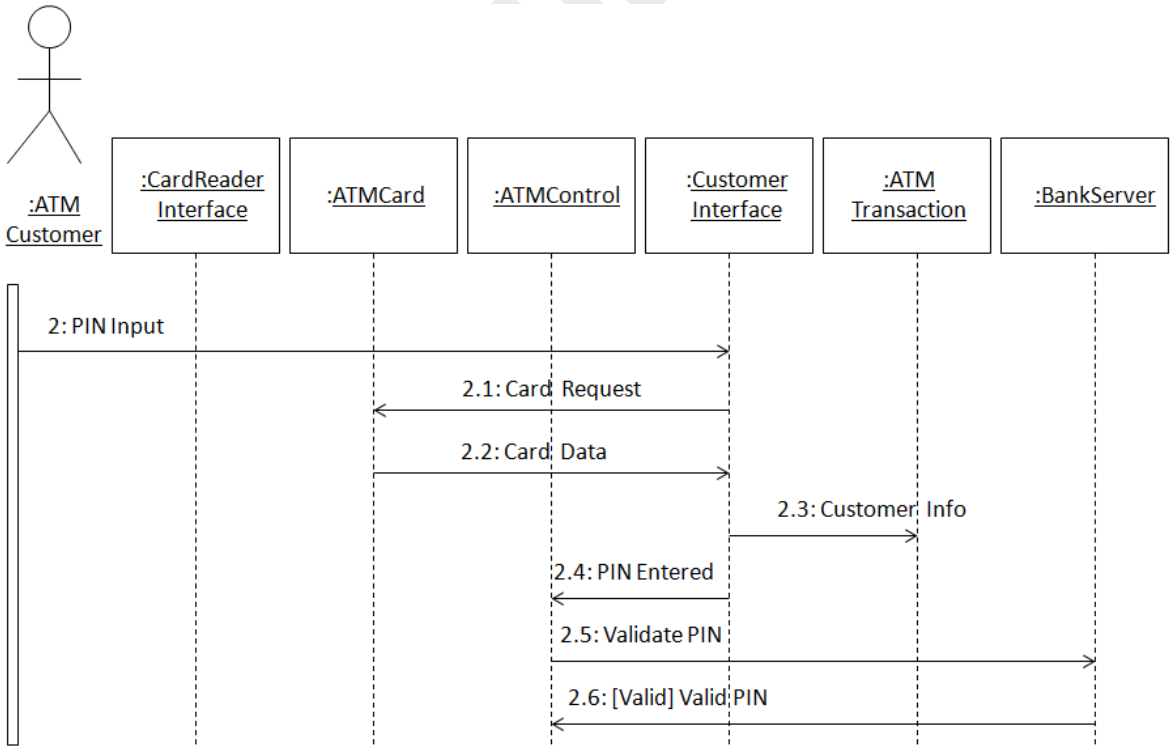


Sequence Diagram

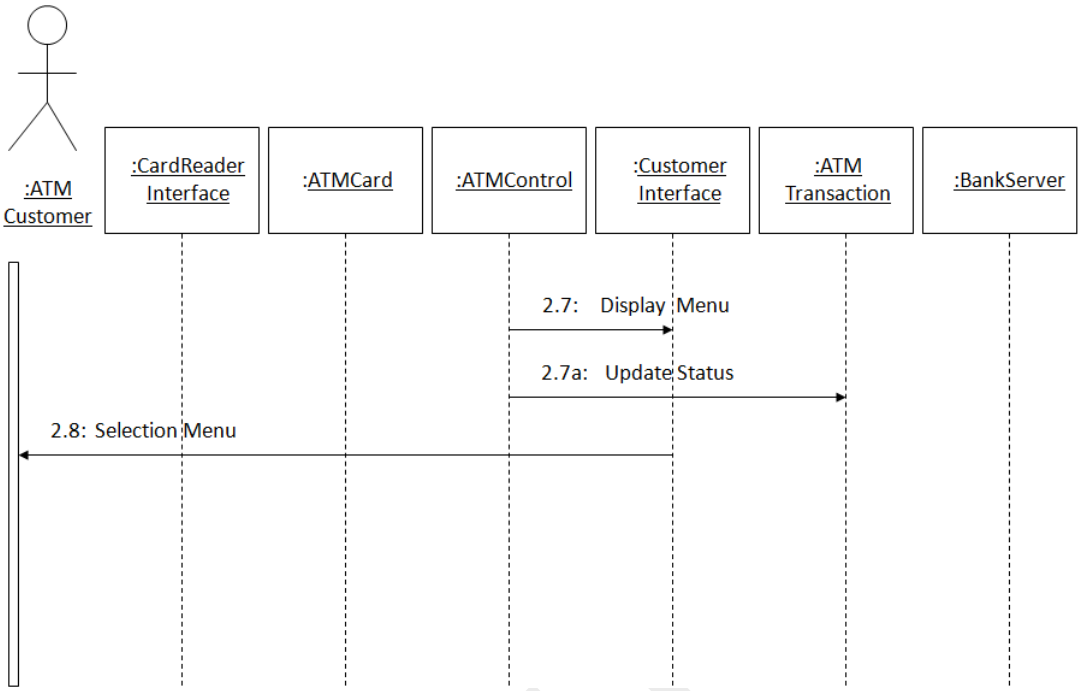
Sequence Diagram: ATM Client Validate PIN Use Case – 1



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