BVRIT
BV Raju Institute of Technology
(UGC-Autonomous)
ESTD 1997

Department of Information Technology

B V Raju Institute of Technology (BVRIT)

Vishnupur, Narsapur - 502 313

Dist Medak(T.S.)

CASE TOOLS (CT) LAB MANUAL

(Course Code A362G - R18 Syllabus)

III B.Tech.(IT) II SEM

Prepared By :

Vijaykumar Mantri,

Associate Professor, IT Dept.

BVRIT, NSP.

vijay_mantri.it@bvrit.ac.in

Introduction to CASE Tools (CT) Lab

- **CASE** tools known as **Computer-Aided Software Engineering** tools is a kind of component-based development which allows its users to rapidly develop information systems. The main goal of CASE technology is the automation of the entire information systems development life cycle process using a set of integrated software tools, such as modeling, methodology and automatic code generation.
- Component based manufacturing has several advantages over custom development. The main advantages are the availability of high quality, defect free products at low cost and at a faster time. The prefabricated components are customized as per the requirements of the customers. The components used are pre-built, ready tested and add value and differentiation by rapid customization to the targeted customers.
- However the products we get from case tools are only a skeleton of the final product required and allot of programming must be done by hand to get a fully finished, good product.

Characteristics of CASE: Some of the characteristics of CASE tools are:

- It is a graphics oriented tool.
- It supports decomposition of process.

Some typical CASE tools are:

- Unified Modeling Language (UML)
- Data Modeling Tools
- Source Code Generation Tools

Introduction to Unified Modeling Language (UML)

- **Unified Modeling Language (UML) is a language :** Unified Modeling Language (UML) is not simply a notation for drawing diagrams, but a complete language for capturing knowledge (Semantics) about a subject and expressing knowledge (Syntax) regarding the subject for the purpose of communication.
- **Applies to modeling and Systems :** Modeling involves a focus on understanding a subject (System), capturing and being able to communicated in this knowledge.
- It is the result of unifying the information systems and technology industry's best engineering practices (principals, techniques, methods and tools).
- The Unified Modelling Language (UML) is an industry standard for object oriented design notation, supported by the Object Management Group (OMG).

The artifacts of a software system:

UML is a language that provides vocabulary and the rules for combing words in that vocabulary for the purpose of communication.

A modeling language is a language whose vocabulary and rules focus on the concept and physical representation of a system. Vocabulary and rules of a language tell us how to create the real world well-formed models, but they don't tell you what model you should create and when should create them.

- **The Unified Modeling Language (UML)** is a standard language for writing software blue prints.

  The UML is a language for

  - Visualizing
  - Specifying
  - Constructing
  - Documenting

Visualizing:

The UML is more than just a bunch of graphical symbols. In UML each symbol has well defined semantics. In this manner one developer can write a model in the UML and another developer or even another tool can interpret the model unambiguously.

Specifying:

UML is used from specifying means building models that are precise, unambiguous and complete. UML addresses the specification of all the important analysis, design and implementation decisions that must be made in developing and deploying a software intensive system.

Constructing:

UML is not a visual programming language but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++ or Visual Basic or even to tables in a relational database or the persistent store of an object-oriented database.

The mapping permits forward engineering. The generation of code from a UML model into a programming language.

The reverse Engineering is also possible you can reconstruct a model from an implementation back into the UML.

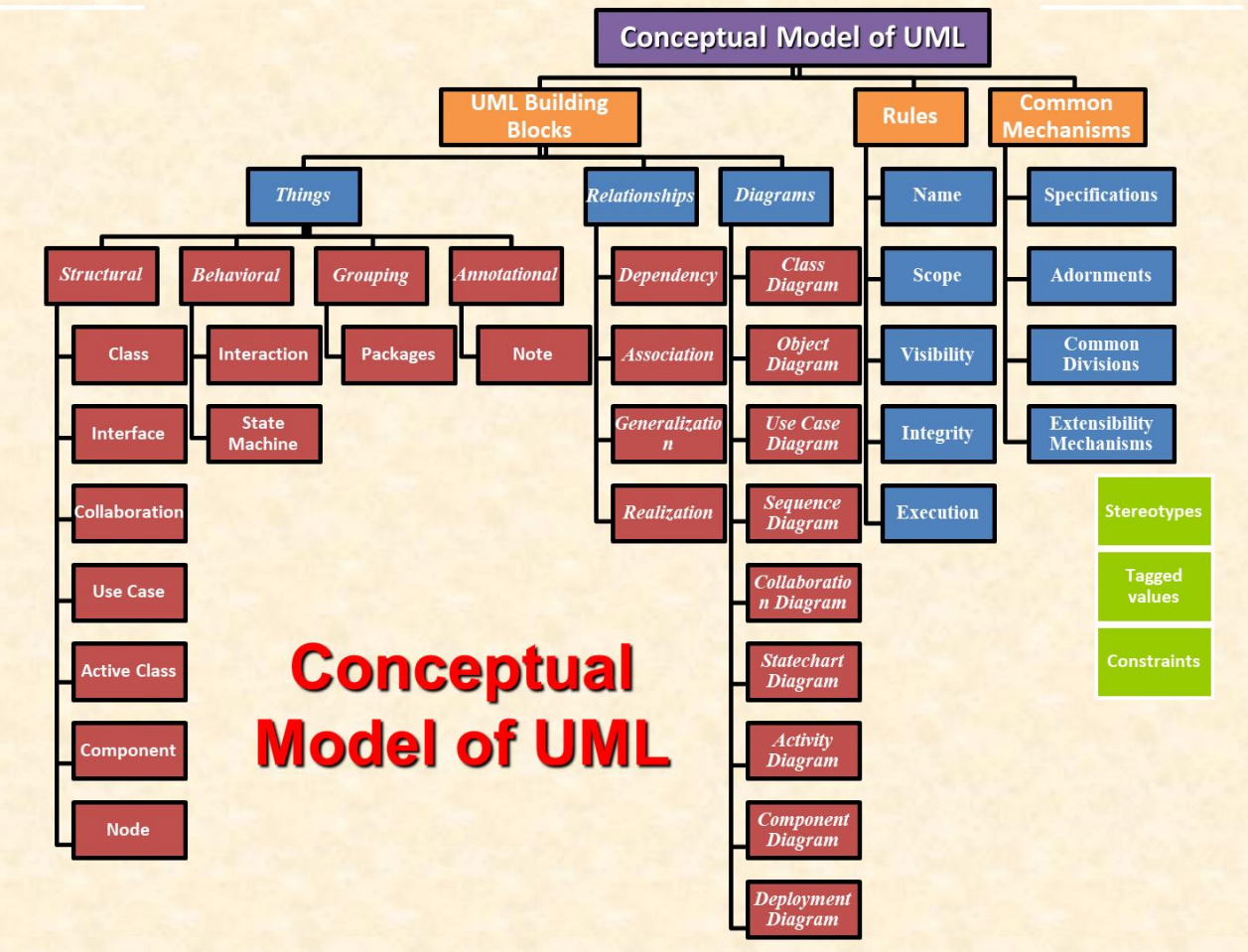
Documenting:

UML is a language for Documenting. A software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include Requirements, Architecture, Design, Source code, Project plans, Test, Prototype, and Release. Such artifacts are not only the deliverables of a project, they are also critical in controlling, measuring and communicating about a system during its development and after its deployment.

Conceptual Model of UML

Conceptual model of the UML:

To understand the UML, we need to form a conceptual model of the language and this requires learning three major elements : Building Blocks, Rules and Common Mechanisms.



The UML Basic Building Blocks.

The Rules that direct how those building blocks may be put together. Some common mechanisms there apply throughout the UML. As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

UML building blocks:

The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

Things:

Things are the most important building blocks of UML. Things can be:

- Structural
- Behavioral
- Grouping
- An notational

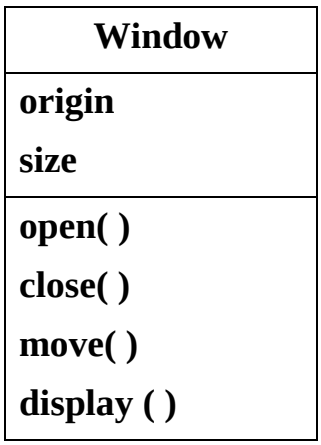
Structural things:

The Structural things define the static part of the model. They represent physical a conceptual elements. Structural things are the nouns of UML models; usually the static parts of the system in question.

Following are the brief descriptions of the structural things.

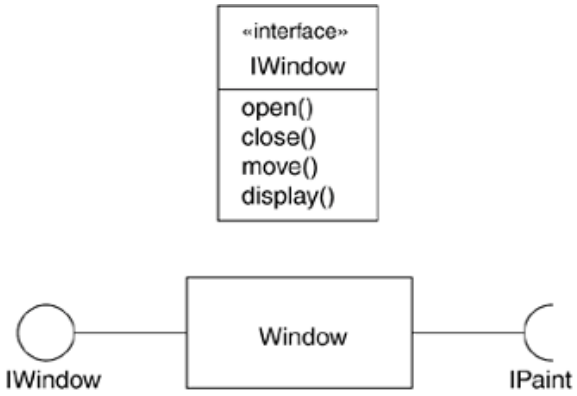
- **Class** – *Class is an abstraction of a set of things in the problem-domain that have similar properties and/or functionality.*

Notation:



- **Interface** - *Interface is a collection of operations that specify the services rendered by a class or component.*

Notation:

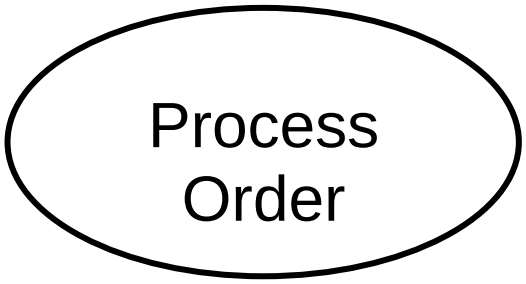


- **Collaboration** – *Collaboration is a collection of UML building blocks (classes, interfaces, relationships) that work together to provide some functionality within the system.*

Notation:

- **Use Case** – *Use case is an abstraction of a set of functions that the system performs; a use case is "realized" by Collaboration.*

Notation :

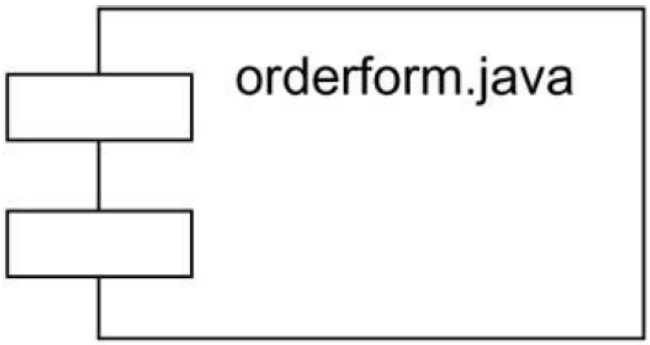


- **Active Class** – *Active class is a class whose instance is an active object; an active object is an object that owns a process or thread (units of execution)*

Notation:

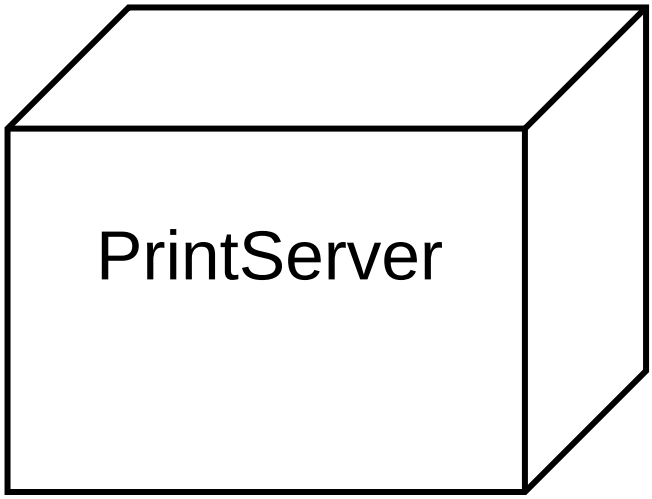
- **Component** – *Component is a physical part (typically manifests itself as a piece of software) of the system.*

Notation :



- **Node** – *A node is a physical element that exists at run-time and represents a computational resource (typically, hardware resources).*

Notation :



Behavioral things:

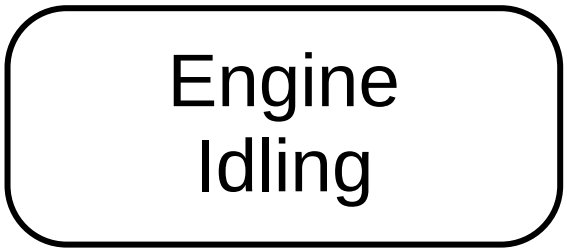
A behavioral thing consists of the dynamic parts of UML models. The verbs of UML models; usually the dynamic parts of the system in question. Following are the behavioral things:

- **Interaction** – *Interaction is some behavior constituted by messages exchanged among objects; the exchange of messages is with a view to achieving some purpose.*

Notation :

- **State Machine** – *State machine is a behavior that specifies the sequence of "States" an object goes through, during its lifetime. A "state" is a condition or situation during the lifetime of an object during which it exhibits certain characteristics and/or performs some function. State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.*

Notation :

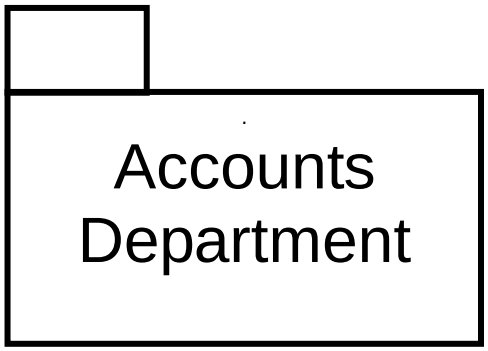


Grouping Things

The organizational part of the UML model; provides a higher level of abstraction (granularity). Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available.

- **Package** – *Package is a general-purpose element that comprises UML elements - structural, behavioral or even grouping things. Packages are conceptual groupings of the system and need not necessarily be implemented as cohesive software modules.*

Notation :

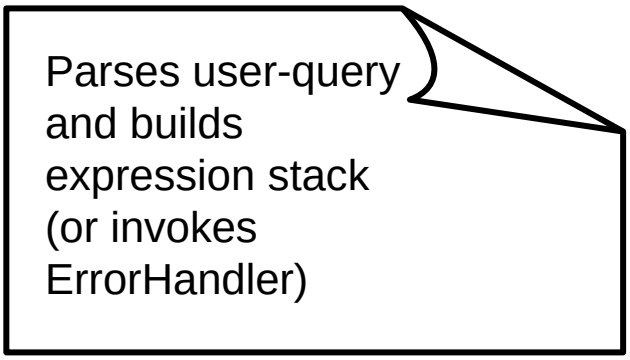


Annotational things:

The explanatory part of the UML model; adds information/meaning to the model elements. Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.

- **Note** - *a graphical notation for attaching constraints and/or comments to elements of the model.*

Notation :



Relationships in UML:

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application. Articulates the meaning of the links between things. There are four kinds of relationships.

- **Dependency** - *Dependency is a semantic relationship where a change in one thing (the independent thing) causes a change in the semantics of the other thing (the dependent thing).*

Notation: 

(Arrow-head points to the independent thing)

- **Association:** *Association is a structural relationship that describes the connection between two things. Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.*

Notation:

- **Generalization:** *Generalization is a relationship between a general thing (called "parent" or "superclass") and a more specific kind of that thing (called the "child" or "subclass"), such that the latter can substitute the former. Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.*

Notation: 

(Arrow-head points to the Superclass)

- **Realization:** *Realization is a semantic relationship between two things wherein one specifies the behavior to be carried out, and the other carries out the behavior. "Collaboration realizes a Use Case". The Use Case specifies the behavior (functionality) to be carried out (provided), and the collaboration actually implements that behavior. Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.*

Notation: 

(Arrow-head points to the thing being realized)

- **UML Diagrams :** UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system. The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

  UML includes the following nine diagrams - each capturing a different dimension of software-system architecture. The details are described in the following.

  1. **Class Diagram**
  2. **Object Diagram**
  3. **Use Case Diagram**
  4. **Sequence Diagram**
  5. **Collaboration Diagram**
  6. **Statechart Diagram**
  7. **Activity Diagram**
  8. **Component Diagram**
  9. **Deployment Diagram**

1. **Class Diagram** – *A Class diagram is the most common diagram found in Object Oriented Analysis & Design. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.*

2. **Object Diagram** - *An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.*

3. **Use Case Diagram** – *A Use Case diagram shows a set of "Use Cases" (sets of functionality performed by the system), the "actors" (typically, people/systems that interact with this system [problem-domain]) and their relationships. Models WHAT the system is expected to do. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.*

   Both Sequence diagrams and Collaboration diagrams are kinds of Interaction diagrams.

   Interaction Diagram - *An interaction diagram shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. Sequence diagrams and Collaboration diagrams are isomorphic, meaning that we can take one & transform it into other.*

4. **Sequence Diagram** - *A Sequence Diagram models the flow of control by time-ordering; depicts the interaction between various objects by of messages passed, with a temporal dimension to it.*

5. **Collaboration Diagram** – *A Collaboration Diagram models the interaction between objects, without the temporal dimension; merely depicts the messages passed between objects.*

6. **Statechart Diagram** - *A statechart diagram shows a state machine, consisting of states, transitions, events that lead to each of these state machines, and activities. Statechart diagrams show the flow of control from state to state. A state diagrams shows the dynamic view of an object. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems*

7. **Activity Diagram** - *An Activity Diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. An "activity" is an ongoing non-atomic execution within a state machine. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.*

8. **Component Diagram** – *A Component Diagram shows the physical packaging of software in terms of components and the dependencies between them.*

9. **Deployment Diagram** - *A Deployment Diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. A node typically hosts one or more artifacts.*

Architecture of UML:

Any real world system is used by different users. The users can be developers, testers, business people, analysts and many more.

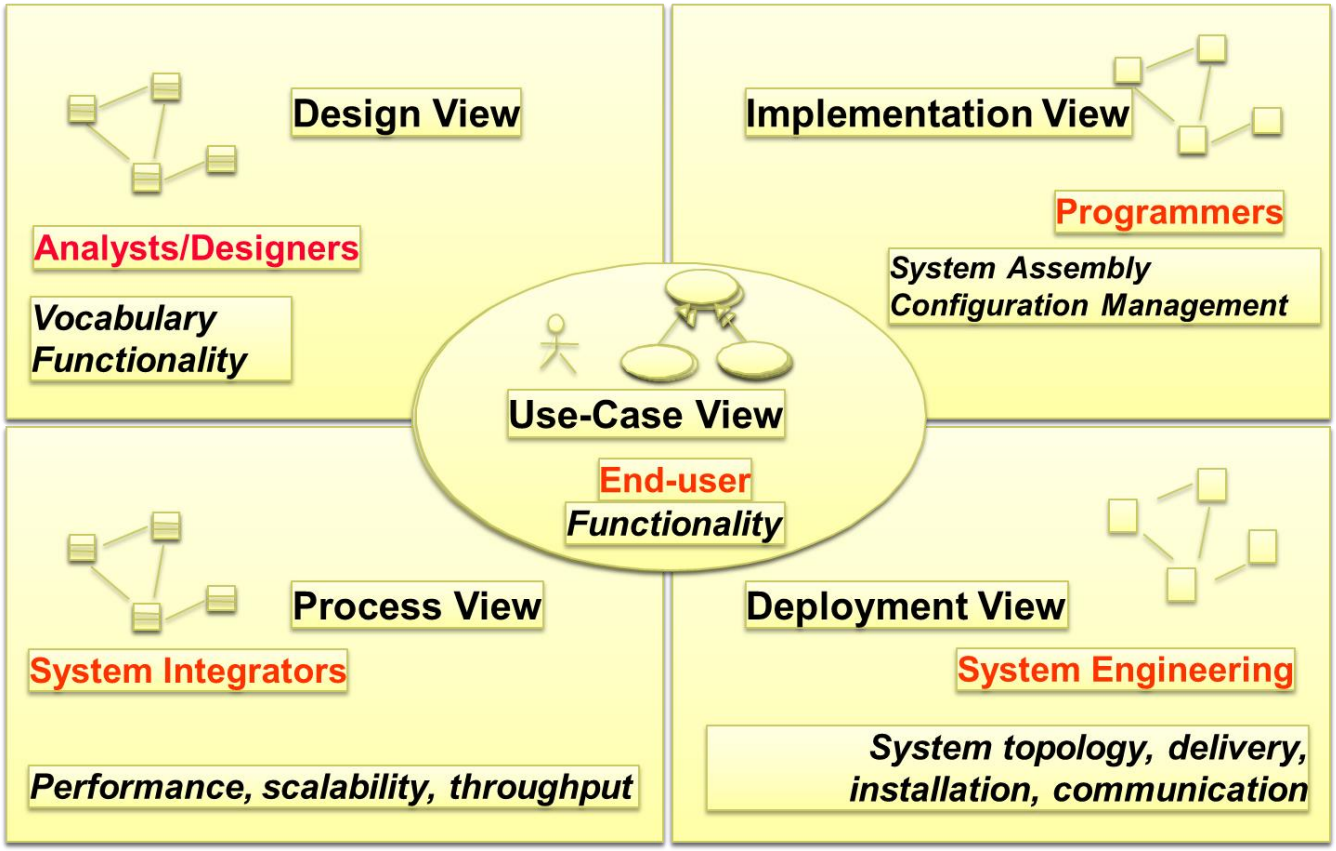
Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders end users, analysts, developers, system integrators, testers, technical writers, and project managers each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life.

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle. So before designing a system the architecture is made with different perspectives in mind. The most important part is to visualize the system from different viewer's perspective. The better we understand the system the better we make the system.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

UML plays an important role in defining different perspectives of a system. These perspectives are:

- **Use Case View**
- **Design View**
- **Process View**
- **Implementation View**
- **Deployment View**



At the center of architecture is the **Use Case view** which connects all these four. A Use case represents the functionality of the system and the other perspectives are connected with use case.

- **Use Case View :** The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.
- **Design View** : The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.
- **Process View** : The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

- **Implementation View :** The implementation view of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.
- **Deployment View :** The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits us to express every one of these five views and their interactions.

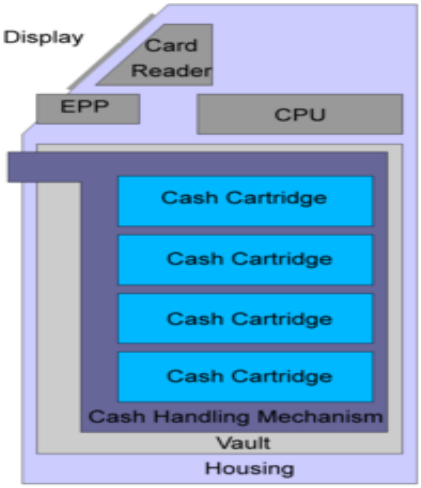
Case Study : Automated Teller Machine (ATM)

Automated Teller Machine (ATM)

An **Automated Teller Machine** (**ATM**), also known as Cash **Machine**, or **Cash point**, is a computerized telecommunications device that provides the end user with access to financial transactions in a public space without the need for a cashier, human clerk or bank teller.

On most modern ATMs, the customer is identified by inserting a plastic ATM card with a **magnetic stripe** or a plastic smart card with a chip that contains a unique card number and some security information. Authentication is provided by the customer entering a personal identification number (PIN).Using ATM, customers can access their bank accounts in order to make cash withdrawals, credit card cash advances, and check their account balances.

Hardware Structure of ATM:



An ATM is typically made up of the following devices:

- CPU (to control the user interface and transaction devices)
- Magnetic and/or Chip card reader (to identify the customer)
- PIN Pad (similar in layout to a Touch tone or Calculator keypad), often manufactured as part of a secure enclosure.
- Secure crypto processor, generally within a secure enclosure.
- Display (used by the customer for performing the transaction)
- Function key buttons (usually close to the display) or a Touch screen (used to select the various aspects of the transaction)
- Record Printer (to provide the customer with a record of their transaction)
- Vault (to store the parts of the machinery requiring restricted access)
- Housing (for aesthetics and to attach signage to)

Recently, due to heavier computing demands and the falling price of computer-like architectures, ATMs have moved away from custom hardware architectures using microcontrollers and/or application-specific integrated circuits to adopting the hardware architecture of a personal computer, such as, USB connections for peripherals, Ethernet and IP communications, and use personal computer operating systems. Although it is undoubtedly cheaper to use commercial off-the-shelf hardware, it does make ATMs potentially vulnerable to the same sort of problems exhibited by conventional computers.

The vault of an ATM is within the footprint of the device itself and is where items of value are kept. Scrip cash dispensers do not incorporate a vault.

Mechanisms found inside the vault may include:

- Dispensing mechanism (to provide cash or other items of value)
- Deposit mechanism including a Cheque Processing Module and Bulk Note Acceptor (to allow the customer to make deposits)
- Security sensors (Magnetic, Thermal, Seismic, Gas)
- Locks (to ensure controlled access to the contents of the vault)
- Journaling systems; many are electronic (a sealed flash memory device based on proprietary standards) or a solid-state device (an actual printer) which accrues all records of activity including access timestamps, number of bills dispensed, etc. - This is considered sensitive data and is secured in similar fashion to the cash as it is a similar liability.

Software used to design ATM:

With the migration to commodity PC hardware, standard commercial "off-the-shelf" operating systems and programming environments can be used inside of ATMs. Typical platforms previously used in ATM development include RMX or OS/2. Today the vast majority of ATMs worldwide use a Microsoft OS, primarily Windows. Linux is also finding some reception in the ATM marketplace. An example of this is Banrisul, the largest bank in the south of Brazil, which has replaced the MS-DOS operating systems in its ATMs with Linux. Banco do Brasil is also migrating ATMs to Linux.

Alternative uses of ATM:

Although ATMs were originally developed as just cash dispensers, they have evolved to include many other bank-related functions. In some countries, especially those which benefit from a fully integrated cross-bank ATM network, ATMs include many functions which are not directly related to the management of one's own bank account, such as :

1) Deposit currency recognition, acceptance, and recycling
2) Paying routine bills, fees, and taxes (utilities, phone bills, social security, legal fees, taxes, etc.)
3) Printing bank statements

4) Updating passbooks

5) Purchasing

- Postage stamps.
- Lottery tickets
- Train tickets

Description of an ATM System:

The software to be designed will control a simulated **Automated Teller Machine (ATM)** having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

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

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

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of Rs.100 or Rs.500 or Rs.1000. Approval must be obtained from the bank before cash is dispensed.
2. A customer must be able to make a transfer of money between any two accounts linked to the card.
3. A customer must be able to make a balance inquiry of any account linked to the card.
4. A customer must be able to change ATM Card PIN.
5. A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine (Or disabled for the Day), and the customer will have to contact the bank to get it back (or unblock card).

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction. The ATM

will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position; the machine will shut down, so that the operator may reload the machine with cash, blank receipts, etc.

Week 1 (Experiment No 1)

Class Diagram & Use Case Diagram for an ATM System

CLASS DIAGRAM :

Class diagrams are the most common diagram found in modeling object- oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

We use class diagrams to model the static design view of a system. For the most part, this involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas. Class diagrams are also the foundation for a couple of related diagrams : component diagrams and deployment diagrams.

Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.

Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagram shows a collection of classes, interfaces, associations, collaborations and constraints. It is also known as a structural diagram.

Purpose:

The purpose of the class diagram is to model the static view of an application. The class diagrams are the only diagrams which can be directly mapped with object oriented languages and thus widely used at the time of construction. The UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application but class diagram is a bit different. So it is the most popular UML diagram in the coder community. So the purpose of the class diagram can be summarized as:

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for Component and Deployment diagrams.
- Forward and Reverse Engineering.

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

USE CASE DIAGRAM:

A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems (activity diagrams, statechart diagrams, sequence diagrams, and collaboration diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems). Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. Each one shows a set of use cases and actors and their relationships.

To model a system the most important aspect is to capture the dynamic behavior. To clarify a bit in details, dynamic behavior means the behavior of the system when it is running/operating.

So only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior.

In UML there are five diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

Purpose:

The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose. Because other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

USE CASES FOR THE ATM :

- **Withdrawal Use Case:** A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a Rupees amount from a menu of possible amounts or enter in multiple of 100. The system verifies that it has sufficient money on hand to satisfy the request as well as minimum balance is there after withdrawal before sending the transaction to the bank. If not, the customer is informed and asked to enter a different amount. If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the Rupees amount.

- **Transfer Use Case:** A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking) from a menu of possible accounts, to choose a different account to transfer to, and to type in a Rupees amount on the keyboard. No further action is required once the transaction is approved by the bank before printing the receipt. A transfer transaction can be cancelled by the customer pressing the Cancel key any time prior to entering a dollar amount.

- **Inquiry Use Case:** An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt. An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.

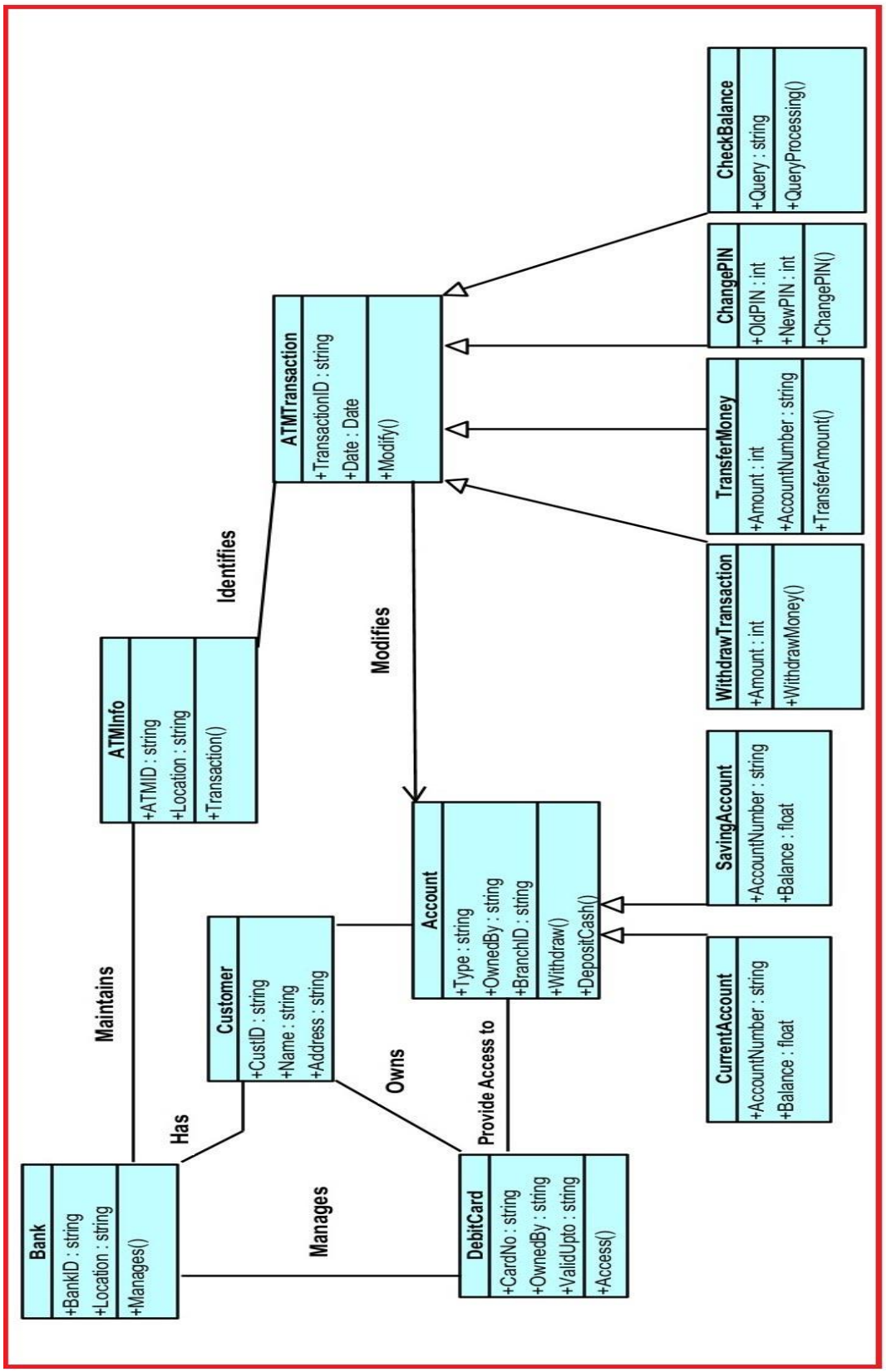
- **Validate User Use Case:** This use case is for validate the user i.e check the pin number, when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to re- enter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered. If the customer fails three times to enter the correct PIN, the card is permanently retained (or blocked for the day), a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted.

- **Print Bill Use Case:** This use case is for printing corresponding bill after transactions (withdraw or balance enquiry, transfer) are completed.

- **Update Account Use Case:** This use case is for updating corresponding user accounts after transactions (withdraw or deposit or transfer) are completed.

Class Diagram for ATM (Sample Class Diagram)

ATM Class Diagram – Drawn using Visual Paradigm



Forward Engineering - Class Diagram

Forward Engineering Java Code generated by Visual Paradigm for Sample ATM Class Diagram.

1) Bank.java

```
public class Bank {
    public string BankID;
    public string Location;

    public void Manages( ) {
        // TODO - implement Bank.Manages
        throw new UnsupportedOperationException( );
    }
}
```

2) ATMInfo.java

```
public class ATMInfo {
    public string Location;
    public string ATMID;

    public void Transaction( ) {
        // TODO - implement ATMInfo.Transaction
        throw new UnsupportedOperationException( );
    }
}
```

3) Customer.java

```
public class Customer {
    public string CustID;
    public string Name;
    public string Address;
}
```

4) Account.java

```
public class Account {
        public string Type;
        public string OwnedBy;
        public string BranchID;

        public void Withdraw( ) {
                // TODO - implement Account.Withdraw
                throw new UnsupportedOperationException( );
        }

        public void DepositCash( ) {
                // TODO - implement Account.DepositCash
                throw new UnsupportedOperationException( );
        }
}
```

5) CurrentAccount.java

```
public class CurrentAccount extends Account {
        public string AccountNumber;
        public float Balance;
}
```

6) SavingAccount.java

```
public class SavingAccount extends Account {
        public string AccountNumber;
        public float Balance;
}
```

7) DebitCard.java

```
public class DebitCard {
        public string CardNo;
        public string OwnedBy;
        public string ValidUpto;

        public void Access( ) {
                // TODO - implement DebitCard.Access
                throw new UnsupportedOperationException( );
        }
}
```

8) ATMTransaction.java

```
public class ATMTransaction {
        public string TransactionID;
        public Date Date;

        public void Modify( ) {
                // TODO - implement ATMTransaction.Modify
                throw new UnsupportedOperationException( );
        }
}
```

9) WithdrawTransaction.java

```
public class WithdrawTransaction extends ATMTransaction {
        public int Amount;

        public void WithdrawMoney( ) {
                // TODO - implement WithdrawTransaction.WithdrawMoney
                throw new UnsupportedOperationException( );
        }
}
```

10) TransferMoney.java

```
public class TransferMoney extends ATMTransaction {
        public int Amount;
        public string AccountNumber;

        public void TransferAmount( ) {
                // TODO - implement TransferMoney.TransferAmount
                throw new UnsupportedOperationException( );
        }
}
```

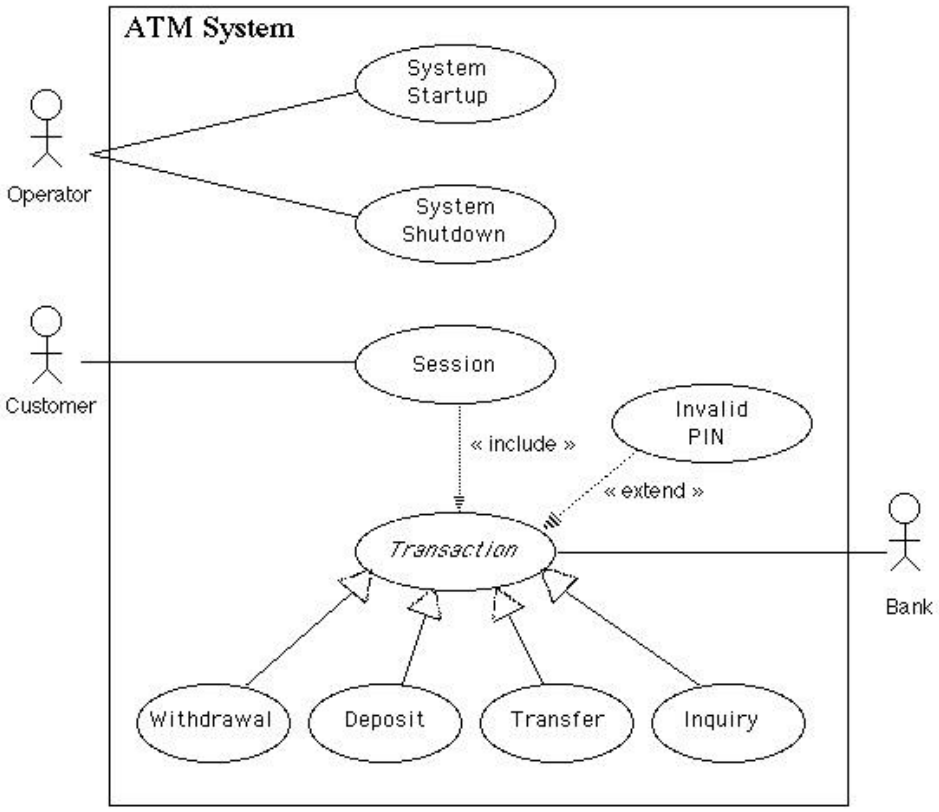
11) ChangePIN.java

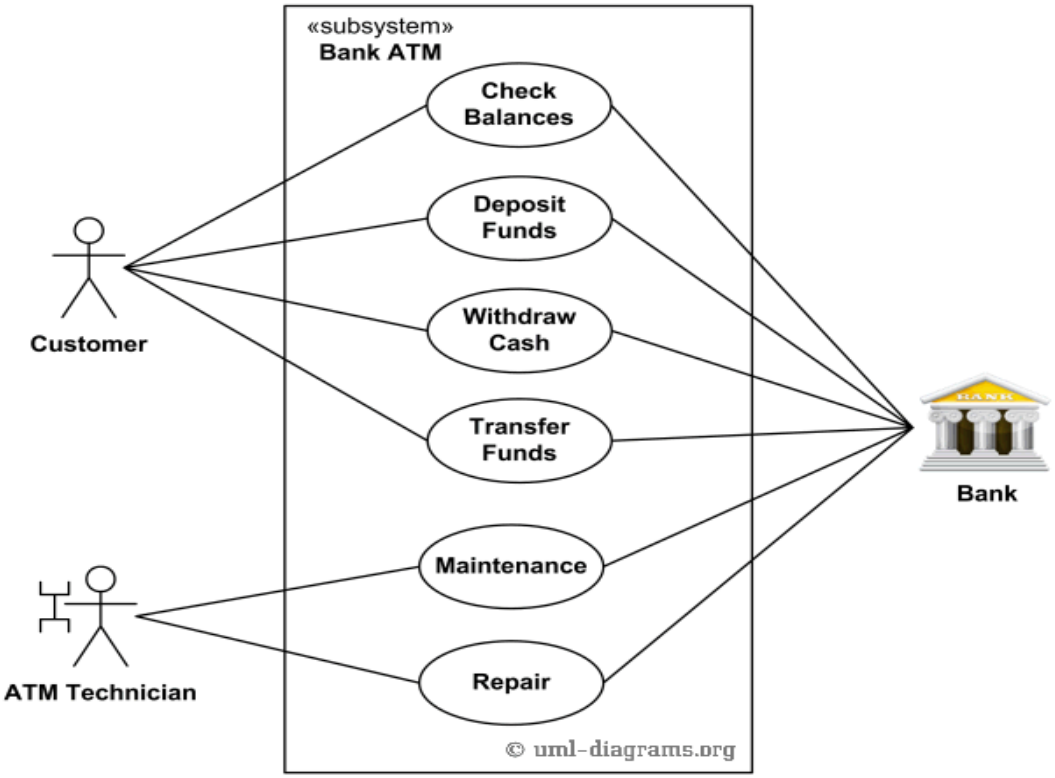
```
public class ChangePIN extends ATMTransaction {
        public int OldPIN;
        public int NewPIN;

        public ChangePIN( ) {
                // TODO - implement ChangePIN.ChangePIN
                throw new UnsupportedOperationException( );
        }
}
```

12) CheckBalance.java

```
public class CheckBalance extends ATMTransaction {
        public string Query;

        public void QueryProcessing( ) {
                // TODO - implement CheckBalance.QueryProcessing
                throw new UnsupportedOperationException( );
        }
}
```

Use Case Diagram for ATM (Sample Use Case Diagrams)





Week 2 (Experiment No 2)

Sequence Diagram & Collaboration Diagram for an ATM System

INTERACTION DIAGRAMS

- *Both **Sequence Diagrams and Collaboration Diagrams** are kinds of **Interaction Diagrams**.*
- ***An interaction diagram** shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them.*
- *Interaction diagrams address the dynamic view of a system.*
- *Sequence diagrams and Collaboration diagrams are isomorphic, meaning that we can take one & transform it into other.*
- ***Sequence Diagram** - models the flow of control by time-ordering; depicts the interaction between various objects by of messages passed, with a temporal dimension to it.*
- ***Collaboration Diagram** - models the interaction between objects, without the temporal dimension; merely depicts the messages passed between objects.*

SEQUENCE DIAGRAM

- We use sequence diagrams to illustrate the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes

Contents of a Sequence Diagram

- Objects
- Focus of control
- Messages
- Life line
- Contents

COLLABORATION DIAGRAM

- We use collaboration diagrams to illustrate the dynamic view of a system.
- **Collaboration Diagram -** A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Contents of a Collaboration Diagram

- Objects
- Links
- Messages

Note : Sequence and Collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.

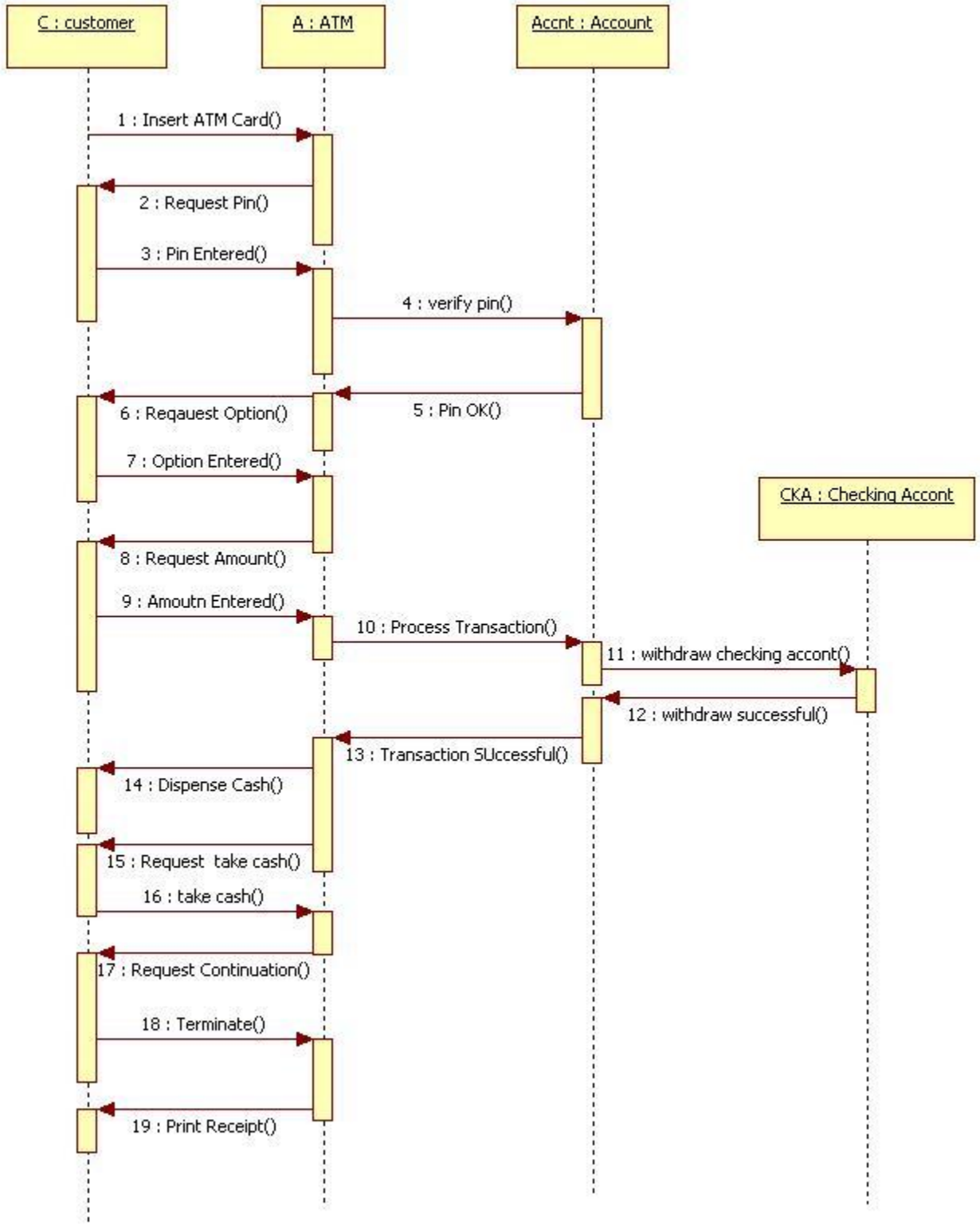
The following things are identified clearly before drawing the interaction diagram:

1. Objects taking part in the interaction.
2. Message flows among the objects.
3. The sequence in which the messages are flowing.
4. Object organization.

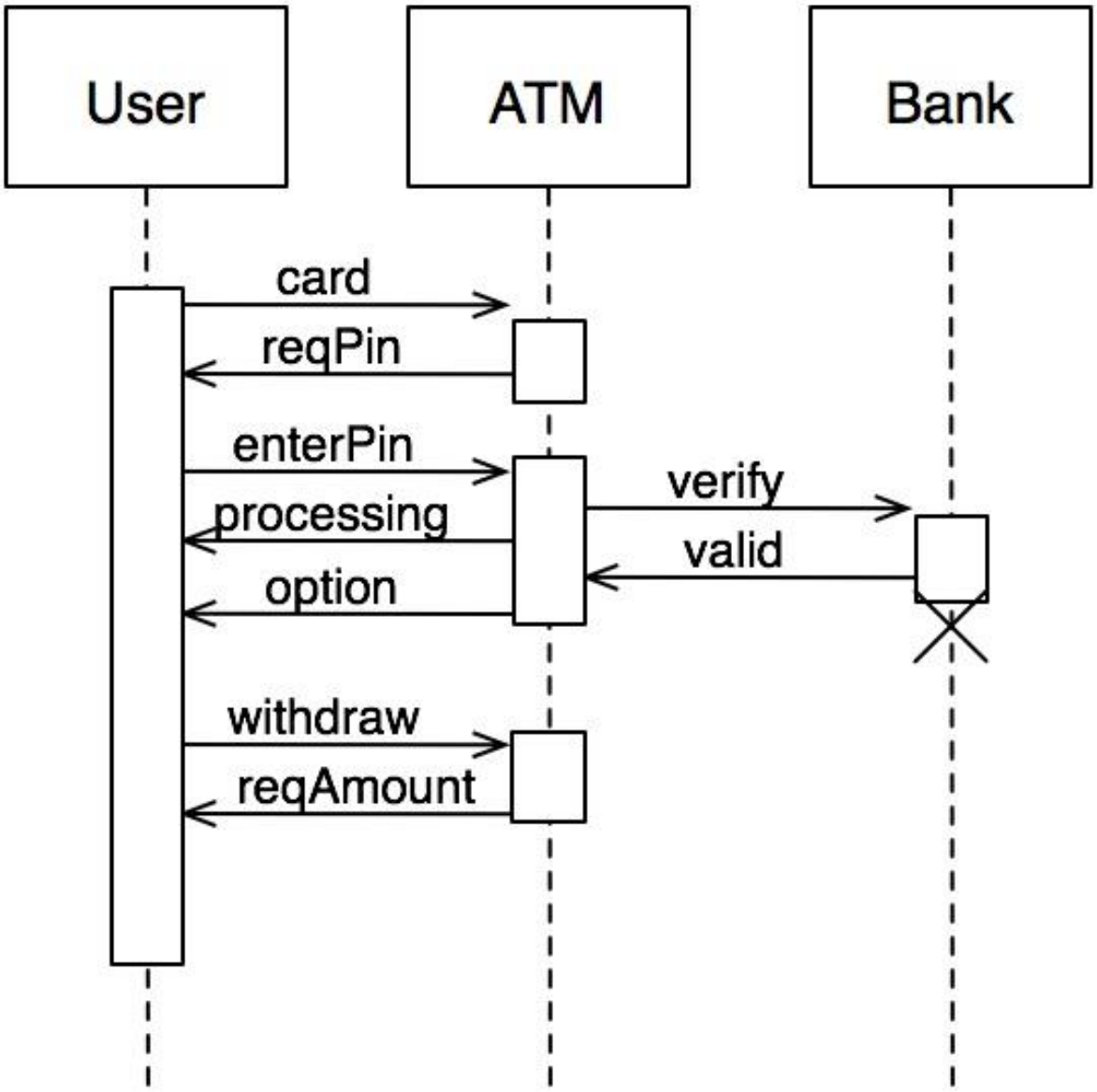
Purpose:

1. To capture dynamic behavior of a system.
2. To describe the message flow in the system.
3. To describe structural organization of the objects.
4. To describe interaction among objects.

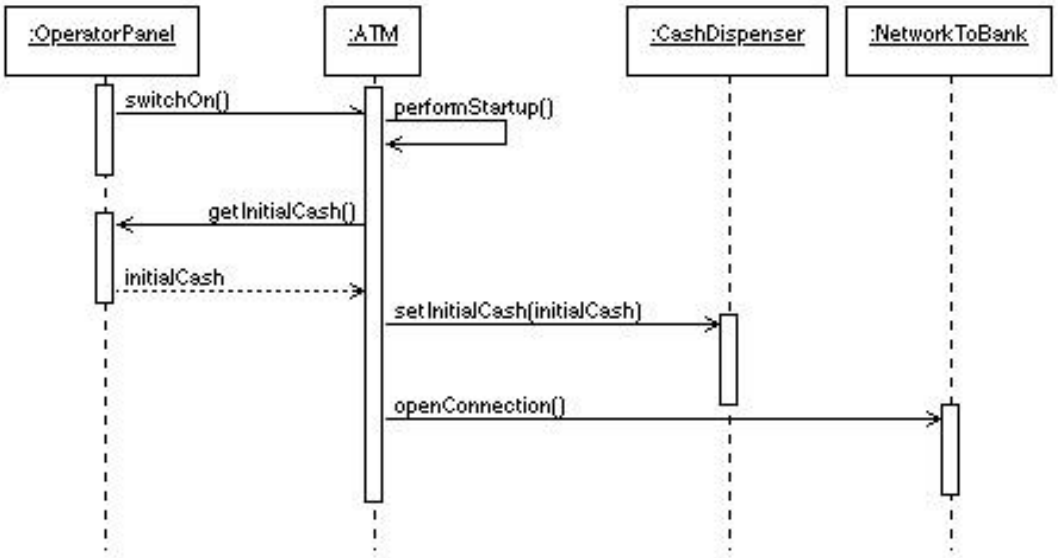
Sequence Diagram for ATM (Sample Sequence Diagrams)



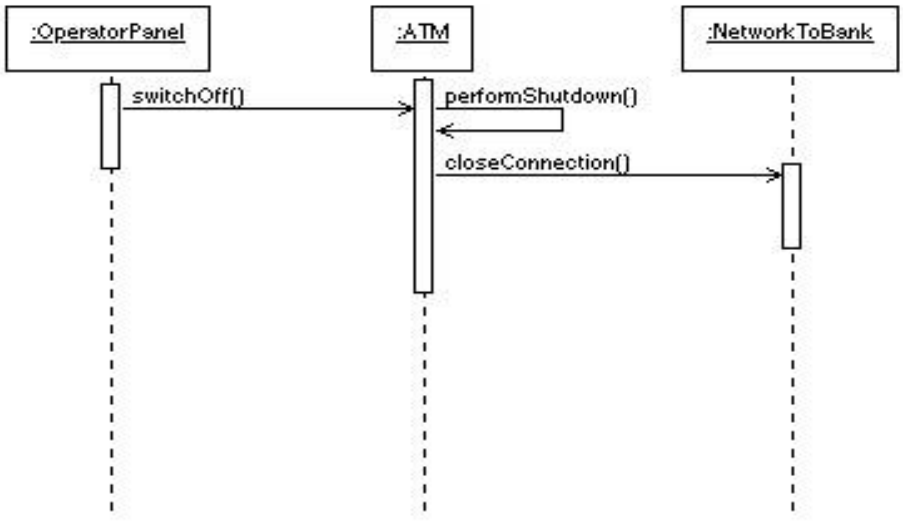
Sequence Diagram for ATM Amount Withdrawal



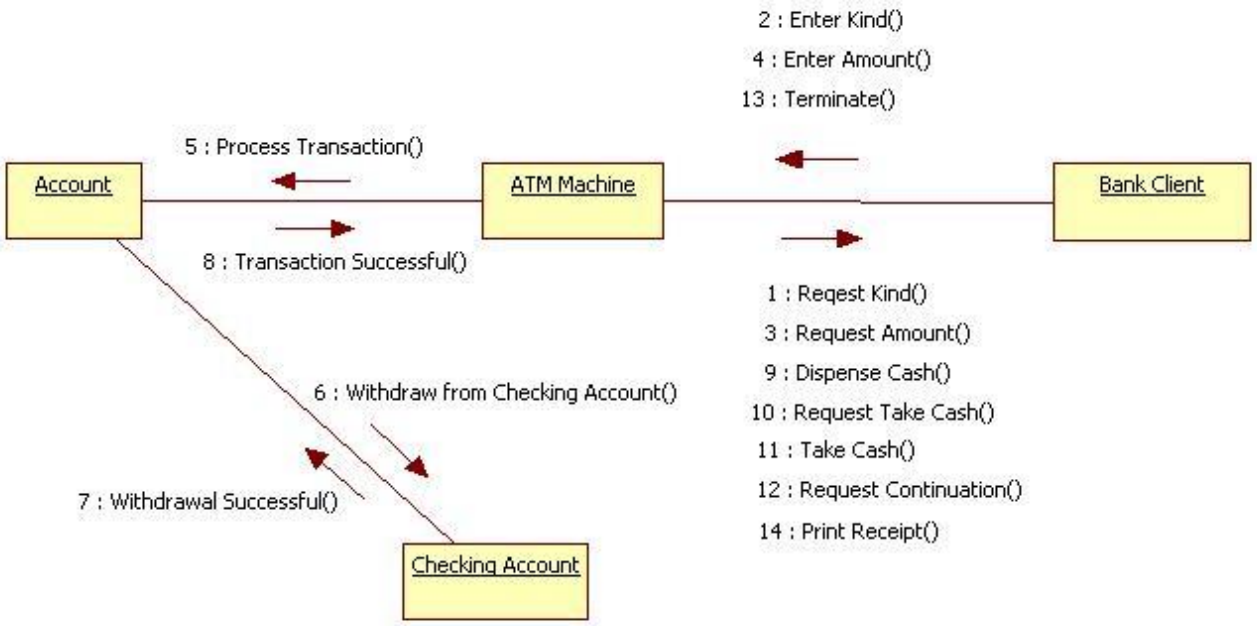
System Startup Sequence Diagram



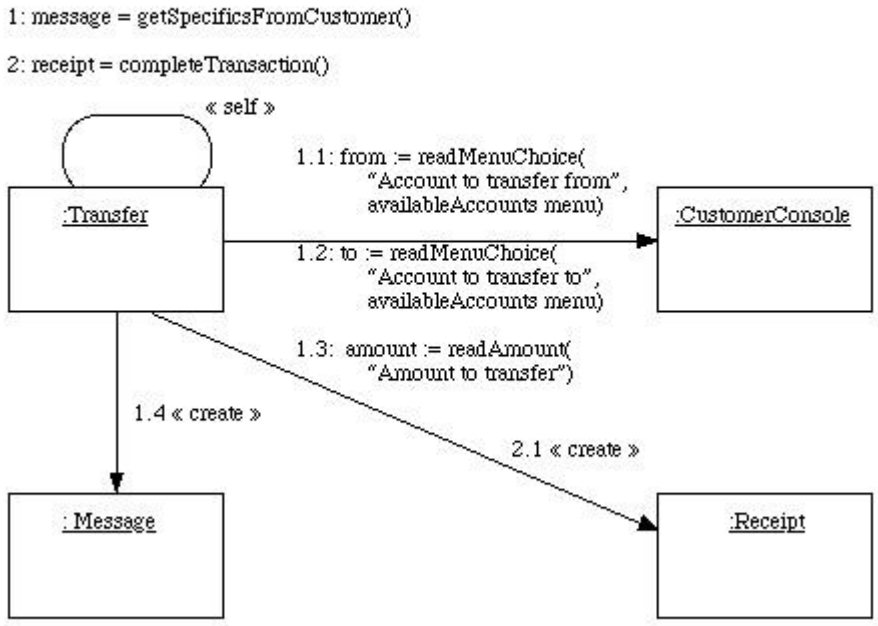
System Shutdown Sequence Diagram



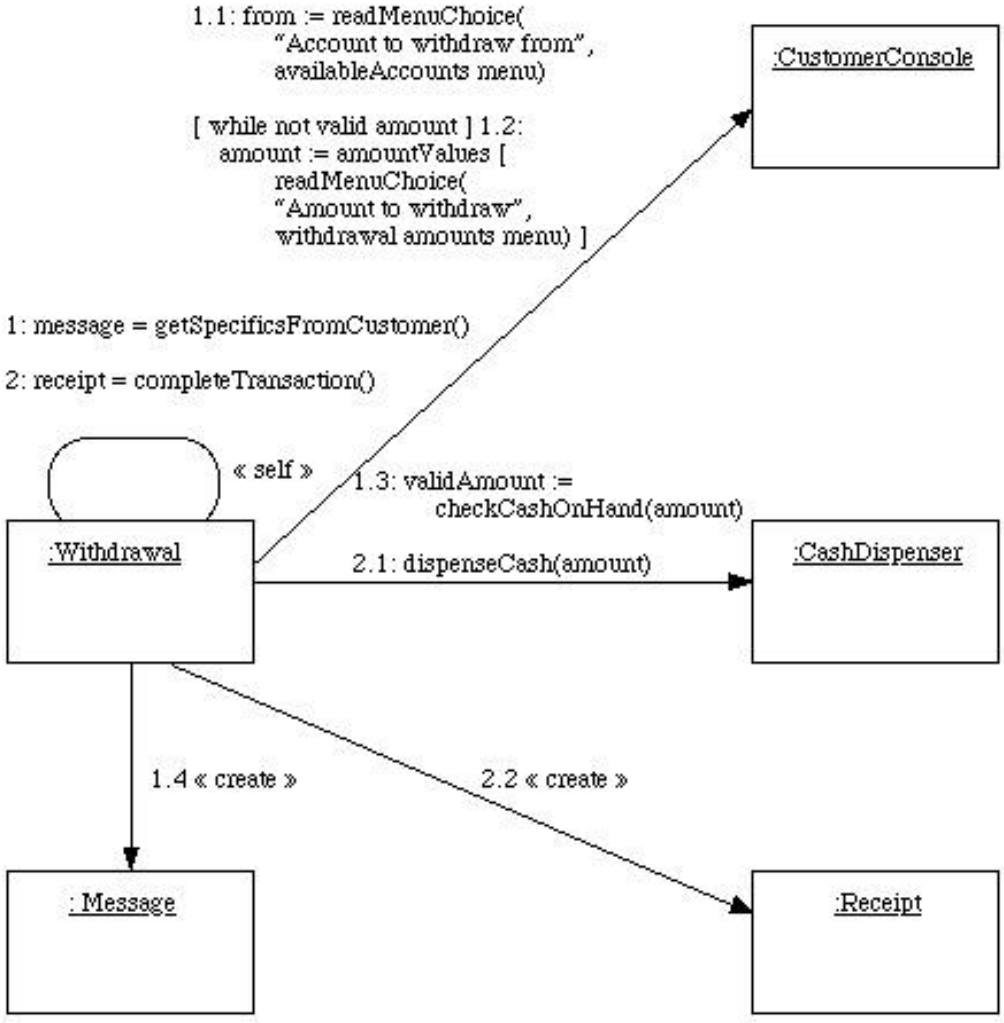
Collaboration Diagram for ATM (Sample Collaboration Diagrams)



Transfer Transaction Collaboration



Withdrawal Transaction Collaboration



Week 3 (Experiment No 3)

Statechart Diagram & Activity Diagram for an ATM System

STATECHART DIAGRAM

- Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems.
- **Statechart Diagram -** A statechart diagram shows a state machine, consisting of states, transitions, events that lead to each of these state machines, and activities.
- Statechart diagram shows the flow of control from state to state.
- The most important purpose of Statechart diagram is to model life time of an object from creation to termination
- A statechart diagrams shows the dynamic view of an object.
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.
- Reactive systems can be defined as a system that responds to external or internal events.
- Statechart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, construct, and document the dynamics of an individual object.
- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- An activity is ongoing nonatomic execution within a state machine.
- An action is an executable atomic computation that results in a change in state of the model or the return of a value.
- Statechart diagrams are also used for forward and reverse engineering of a system. But the main purpose is to model reactive system.

Contents of a Statechart Diagram

- Simple states and composite states
- Transitions, including events and actions

Purpose:

Following are the main purposes of using Statechart Diagrams:

1. To model dynamic aspect of a system.
2. To model life time of a reactive system.
3. To describe different states of an object during its life time.
4. Define a state machine to model states of an object.

ACTIVITY DIAGRAM

- An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state
- Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity.
- **Activity Diagram** - An activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation.
- This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow by using elements like fork, join etc.
- An "activity" is an ongoing non-atomic execution within a state machine.
- Activity diagrams address the dynamic view of a system.
- They are especially important in modeling the function of a system and emphasize the flow of control among objects.

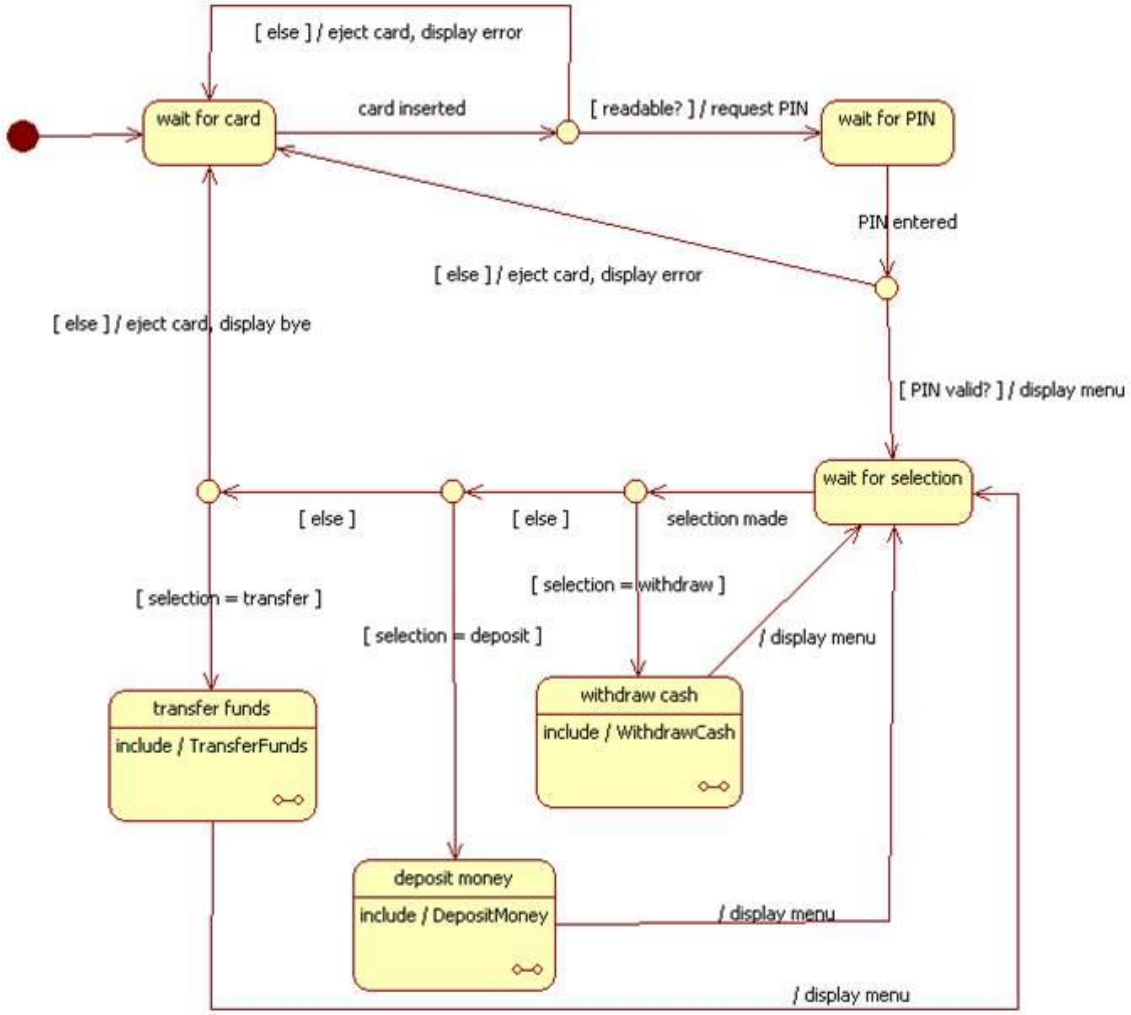
Contents of a Statechart Diagram

- Activity states and action states
- Transitions
- Objects

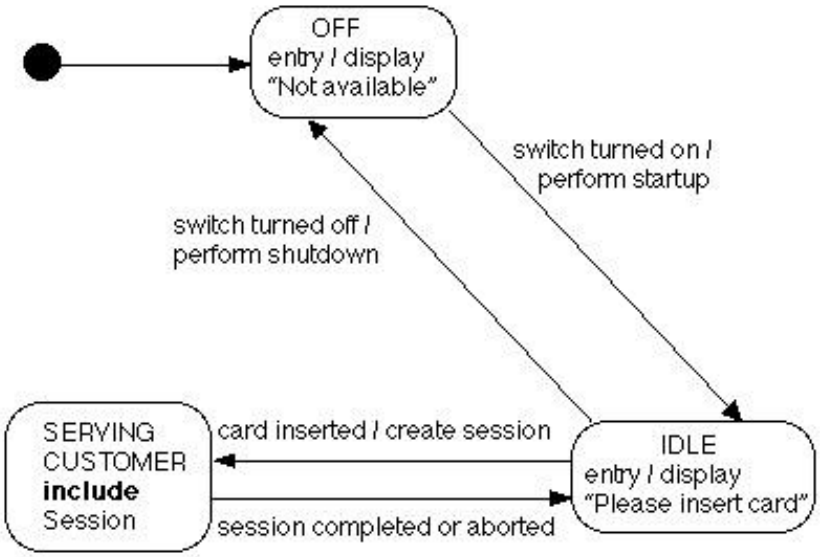
Few terminology :

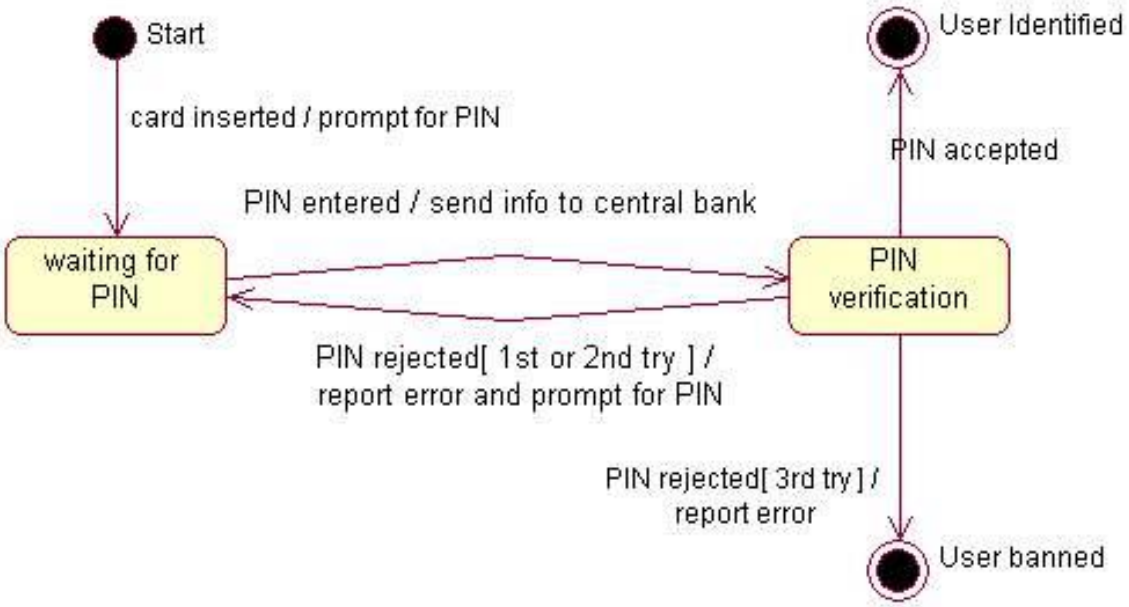
- **Fork :** A fork represents the splitting of a single flow of control into two or more concurrent flow of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below fork the activities associated with each of these path continues in parallel.
- **Join :** A join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transition and one outgoing transition. Above the join the activities associated with each of these paths continues in parallel.
- **Branching :** A branch specifies alternate paths takes based on some Boolean expression Branch is represented by diamond. Branch may have one incoming transition and two or more outgoing one on each outgoing transition, you place a Boolean expression shouldn't overlap but they should cover all possibilities.
- **Swimlane:** Swimlanes are useful when we model workflows of business processes to partition the activity states on an activity diagram into groups. Each group representing the business organization responsible for those activities, these groups are called Swimlanes.

Statechart Diagram for ATM (Sample Statechart Diagrams)

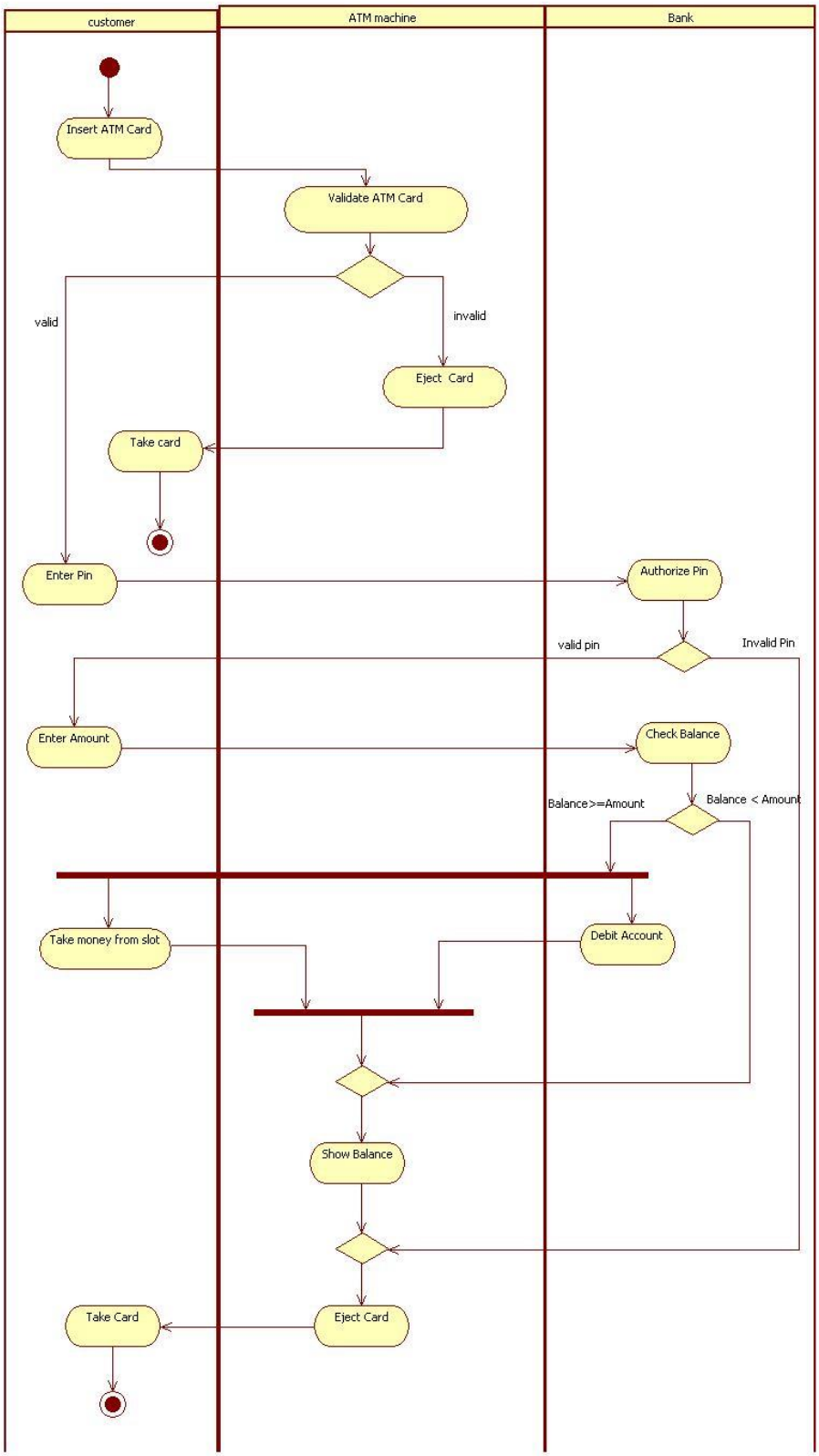


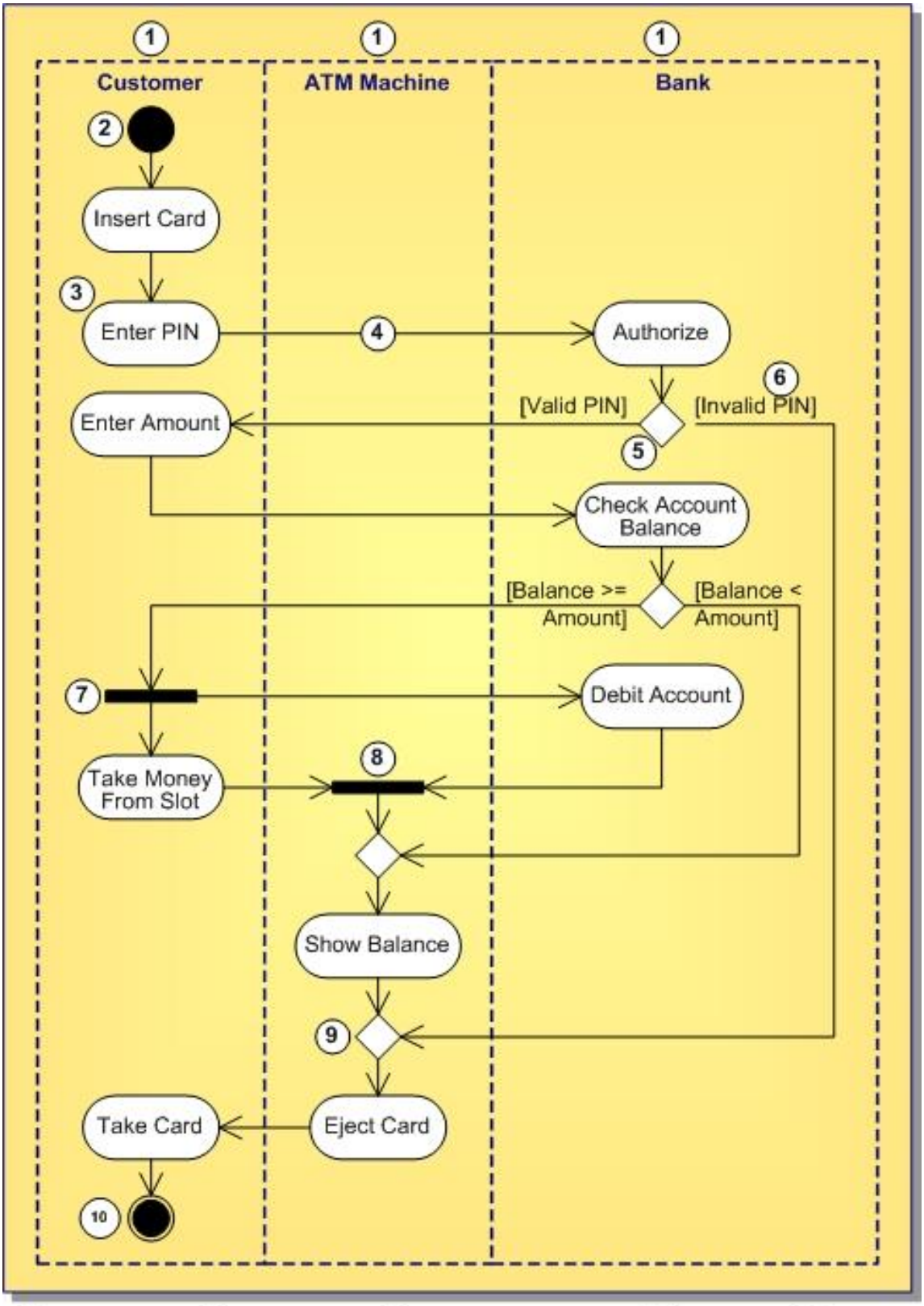
State-Chart for Overall ATM (includes System Startup and System Shutdown Use Cases)





Activity Diagram for ATM (Sample Activity Diagrams)




1
1
1
Customer
ATM Machine
Bank
2
Insert Card
3
Enter PIN
4
Authorize
6
[Valid PIN]
[Invalid PIN]
Enter Amount
5
Check Account Balance
[Balance >= Amount]
[Balance < Amount]
7
Debit Account
Take Money From Slot
8
Show Balance
9
Take Card
Eject Card
10
1 Swimlane
2 Start
3 Activity
4 Transisiton
5 Branch
6 Guard Expression
7 Fork
8 Join
9 Merge
10 End

Week 4 (Experiment No 4)

Component Diagram & Deployment Diagram for an ATM System

COMPONENT DIAGRAM

- Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems. A component diagram shows the organization and dependencies among a set of components.
- **Component Diagram** - shows the physical packaging of software in terms of components and the dependencies between them.
- We use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents. Component diagrams are essentially class diagrams that focus on a system's components.

Contents of a Component Diagram

- Components
- Interfaces
- Dependency, generalization, association, and realization relationships

Purpose:

Following are the main purposes of using Component Diagrams:

1. To model source code.
2. To model executable releases.
3. To model physical databases.
4. To model adaptable systems.

DEPLOYMENT DIAGRAM

- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.
- **Deployment Diagram** - A deployment diagram shows the configuration of run-time processing nodes and the components that live on them.
- A node typically hosts one or more artifacts.
- Deployment diagrams address the static deployment view of architecture.
- For the most part, this involves modeling the topology of the hardware on which your system executes. Deployment diagrams are essentially class diagrams that focus on a system's nodes.

Contents of a Deployment Diagram

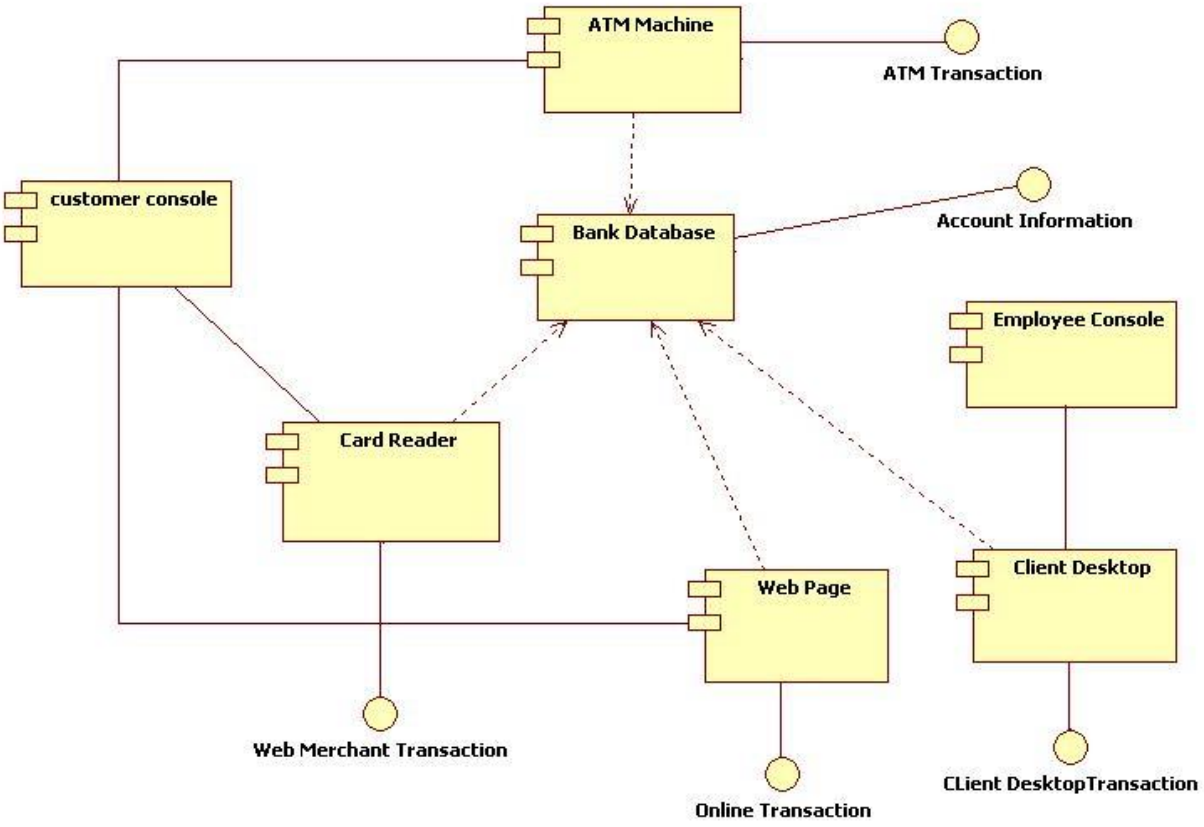
- Nodes
- Dependency and association relationships

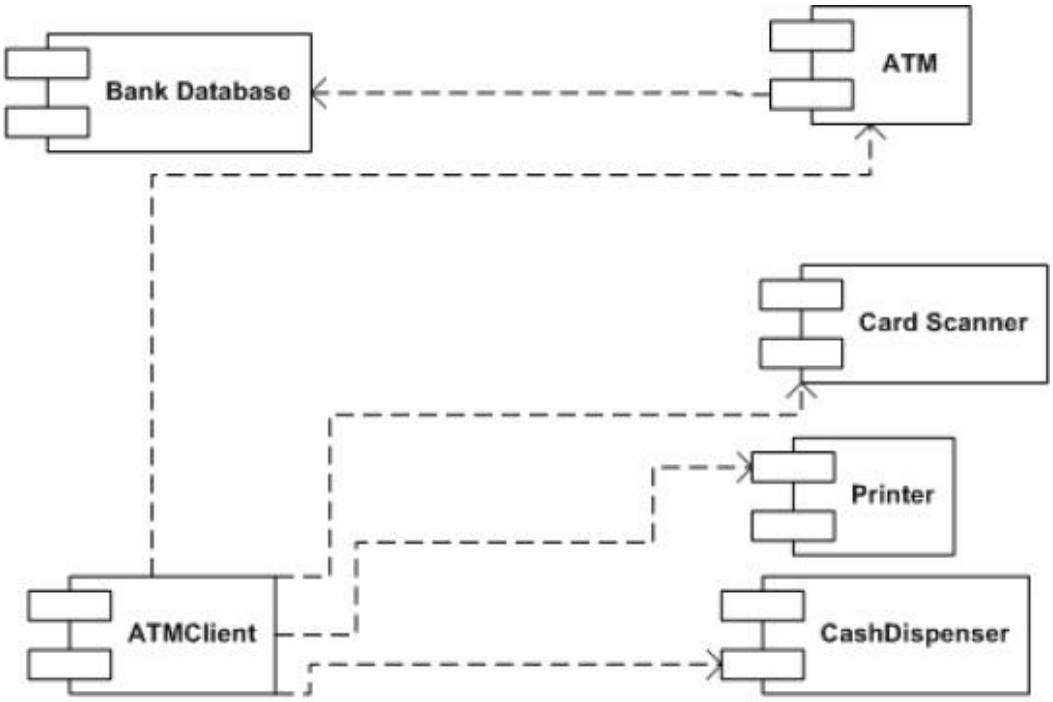
Purpose:

Following are the main purposes of using Deployment Diagrams:

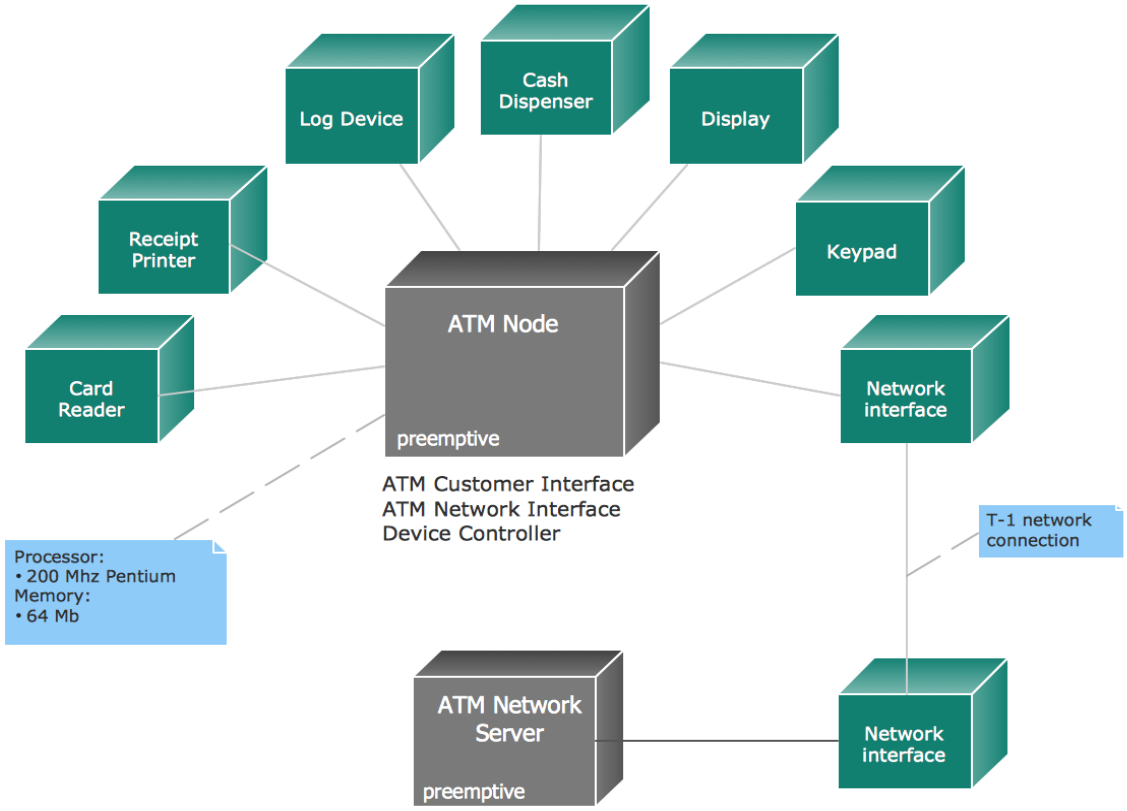
1. To model embedded systems.
2. To model client/server systems.
3. To model fully distributed systems.

Component Diagram for ATM (Sample Component Diagrams)

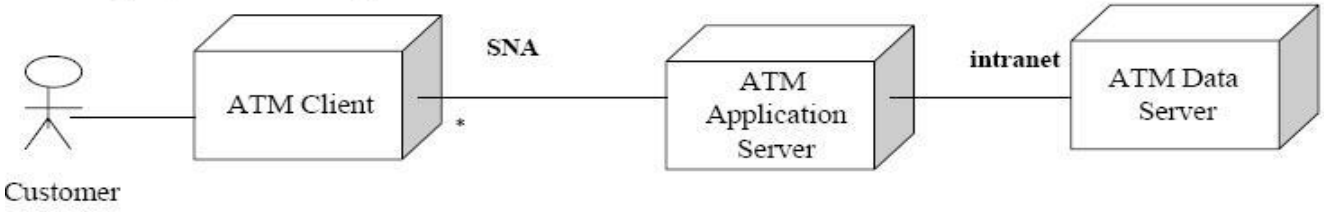




Deployment Diagram for ATM (Sample Deployment Diagrams)



- *Deployment Diagram*



- *Deployment of Active Objects*