Unit III Requirement Engineering

Prof. M.P.Karnik
madhuri.chavan@viit.ac.in
Department of Computer Engineering

VISHWAKARMA INSTITUTES
VI

BRACT'S, Vishwakarma Institute of Information Technology, Pune-48

(An Autonomous Institute affiliated to Savitribai Phule Pune University)
(NBA and NAAC accredited, ISO 9001:2015 certified)

Contents

- Requirements Elicitation : Concept of Software Requirement
- Categories and types of Requirements
- Elicitation Techniques- real life application case study.
- Requirements Analysis and Documentation: Textual and Graphical Documentation
- UML models: Use Case Diagram and class diagram
- data modelling, data and control flow model
- behavioral modelling using state diagrams - real life application case study
- Software Requirement Specifications (SRS).

Requirements Elicitation : Concept of Software Requirement

- The process to gather the software requirements from client, analyze and document them is known as **requirement engineering.**

- The goal of requirement engineering is to develop and maintain sophisticated and descriptive ‘System Requirements Specification’ document.

Requirement Engineering Process

- It is a four step process, which includes –

- **Feasibility Study-** This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization.

- **Requirement Gathering-** If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

- **Software Requirement Specification:** SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

- **Software Requirement Validation:** Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

Requirement elicitation

- **Requirement elicitation** is the process of collecting the requirements of a system or requirement gathering from user, customers and stakeholders by conducting meetings, interviews, questionnaires, brainstorming sessions, prototyping etc.

- Elicitation is gathering of all the system requirements from the stakeholders and it encompasses all activities involved in discovering the requirements of a system.
- The system developers and engineers work in a close relationship with the customers and end-users to determine more about the problem to be solved and to bridge the gap between the stakeholders and the developers.
- **Elicitation techniques facilitates this process by,**

✓ Finding out more about the problem to be solved.

✓ Describing the functionalities of the system and non functional attributes.

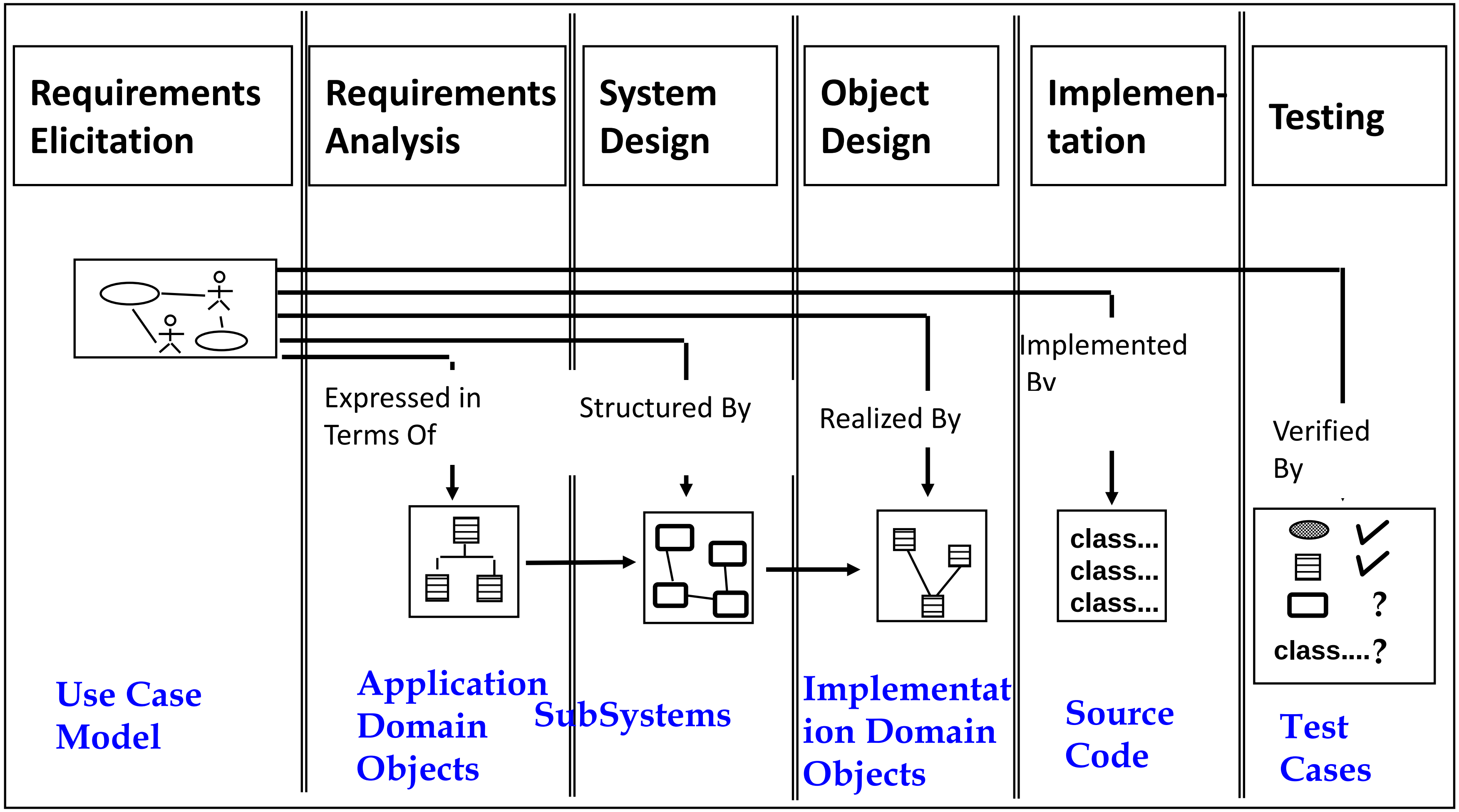
✓ Enhances the performance of the system.

✓ Overcomes hardware constraints.

✓ Bridges the gap between the stakeholders and the developers.

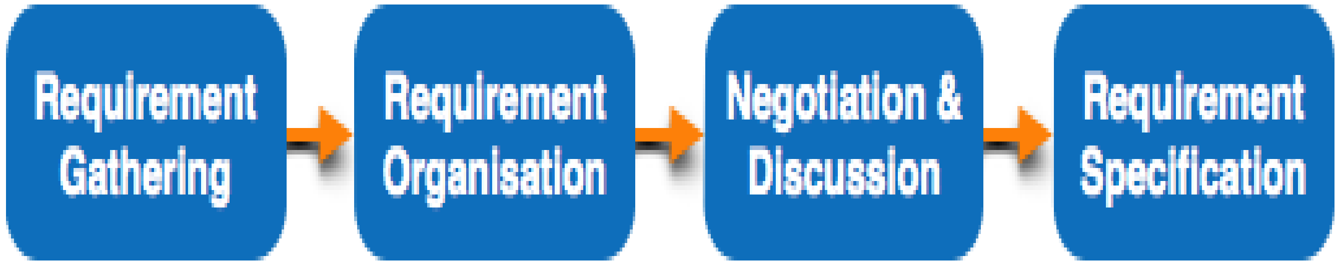
- The requirement elicitation process involves effectively designing the requirements by thorough understanding. After thorough understanding of requirements, build a use case model which is a part of analysis model.
- Once the use case model is defined it is expressed in terms of application domain objects which help in understanding the domain in which it has to be applied.
- The definition of the use case model can facilitate the process of dividing the system into subsystems using block diagrams and it is done after the analysis phase.
- In object design, the solutions provided by the domain objects can be realized from the plotting of the use case models.
- Further from the use case models, it is easy to develop and to implement the source code by dividing each and every class; further more it helps us to verify by defining the test cases early in the **development cycle.**

Software Lifecycle Activities



REQUIREMENT ELICITATION PROCESS

- Requirement elicitation process can be depicted using the following diagram:



Requirements gathering - The developers discuss with the client and end users and know their expectations from the software.
Organizing Requirements - The developers prioritize and arrange the requirements in order of importance, urgency and convenience.

- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.

The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.

- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques

- Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.
- There are various ways to discover requirements

- **Interviews**

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- Oral interviews

- Written interviews
- One-to-one interviews which are held between two persons across the table.
- Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

▪ **Surveys**

- Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

▪ **Questionnaires**

- A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

- A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

- **Task analysis**

- Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

- **Domain Analysis**

- Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

- **Brainstorming**

- An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

- **Prototyping**
- Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements. If there is no software installed at client's end for developer's reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.
- **Observation**
- Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

ELICITATION TECHNIQUES

- Interview
- Questionnaire
- Background Reading
- Introspection
- Social Analysis
- Requirement workshop.
- Brainstorming
- Idea Generation
- Story Boarding
- Scripted Walkthrough
- CRC Cards
- Ethnography
- Natural language descriptions
- Form Analysis

PROBLEMS OF REQUIREMENT ELICITATION

- **Some of the problems encountered during requirement elicitation are as follows:**

- Stakeholders don’t know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organizational and political factors may influence the system requirements.
- The requirements change during the analysis process.
- New stakeholders may emerge and business environment may also likely to change.

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

Categories and types of Requirements

➢ **Functional Requirements**

- Requirements, which are related to functional aspect of software fall into this category. They define functions and functionality within and from the software system.
- **Examples -**
- Search option given to user to search from various invoices.
- User should be able to mail any report to management.
- Users can be divided into groups and groups can be given separate rights.
- Should comply business rules and administrative functions.
- Software is developed keeping downward compatibility intact.
- A **library system** that provides a single interface to a number of databases of articles in different libraries. Users can search for, download and print these articles for personal study.

Non-Functional Requirements

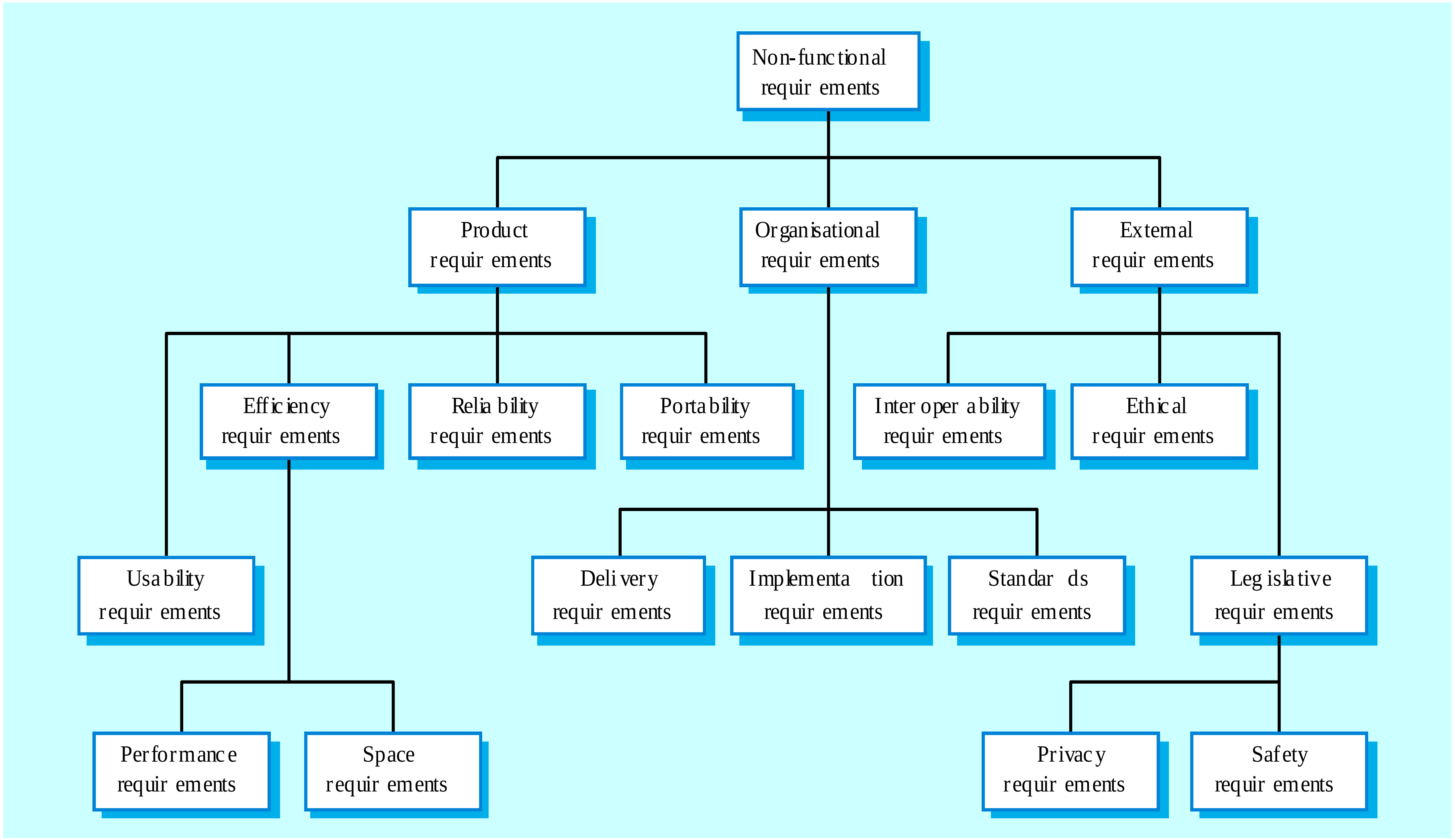
- Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.
- Non-functional requirements include -
- Security
- Logging
- Storage
- Configuration
- Performance
- Cost
- Interoperability
- Flexibility
- Disaster recovery
- Accessibility

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional classifications

- **Product requirements**
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- **Organisational requirements**
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- **External requirements**
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, etc.

Non-functional requirement types



Non-functional requirements examples

- Product requirement

  The user interface for library system shall be implemented as simple HTML without frames or Java applets.

- Organisational requirement

- External requirement

  The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Functional and non-functional requirements

- **Functional requirements**
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- **Non-functional requirements**
  - constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- **Domain requirements**
  - Requirements that come from the application domain of the system and that reflect characteristics of that domain.

Types of requirement

- **User requirements**
  - ➢ Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

- **System requirements**
  - ➢ A structured document setting out detailed descriptions of the system’s functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

REQUIREMENT ANALYSIS

- After requirement gathering, requirement analysis is done to check on why the product is needed. Requirement analysis includes:
- Necessity checking – requirements that don’t contribute to the business goals of the organization or specific problem that has to be addressed by the system are checked for necessity for the development of the system.
- Consistency and completeness checking – no contradictions and feasibility checking in terms of budget and schedule.
- Requirement discussion – highlight the requirement and discussion of problematic elements through introspection and interviews.
- Requirement prioritization – requirements are prioritized based on criticality and the highly complex and critical activities are identified for risk plot and low level activities are considered for easy implementation.
- Requirement agreement – compromised set of requirements based on feasibility checking.

UML models: Use Case Diagram and class diagram

Unified modeling Language overview

- UML stands for “Unified Modeling Language”
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the objects of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design.
- The notation used for the COMET(Concurrent Object Modeling and Architectural Design Method) method is the Unified Modeling Language (UML).

- UML is not a language in the same way that we view programming languages such as ‘C++’, ‘Java’ or ‘Basic’.

- UML is however a language in the sense that it has syntax and semantics which convey meaning, understanding and constraints (i.e. what is right and wrong and the limitations of those decisions) to the reader and thereby allows two people fluent in that language to communicate and understand the intention of the other.

- UML represents a collection of essentially graphical (i.e. drawing) notations supplemented by textual descriptions designed to capture requirements and design alternatives.

Why UML for Modeling

- Use graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).

- Help acquire an overall view of a system.

- UML is *not* dependent on any one language or technology.

- UML moves us from fragmentation to standardization.

- Open Standard, Graphical notation for
  - Specifying, visualizing, constructing, and documenting software systems

- Language can be used from general initial design to very specific detailed design across the entire software development lifecycle.

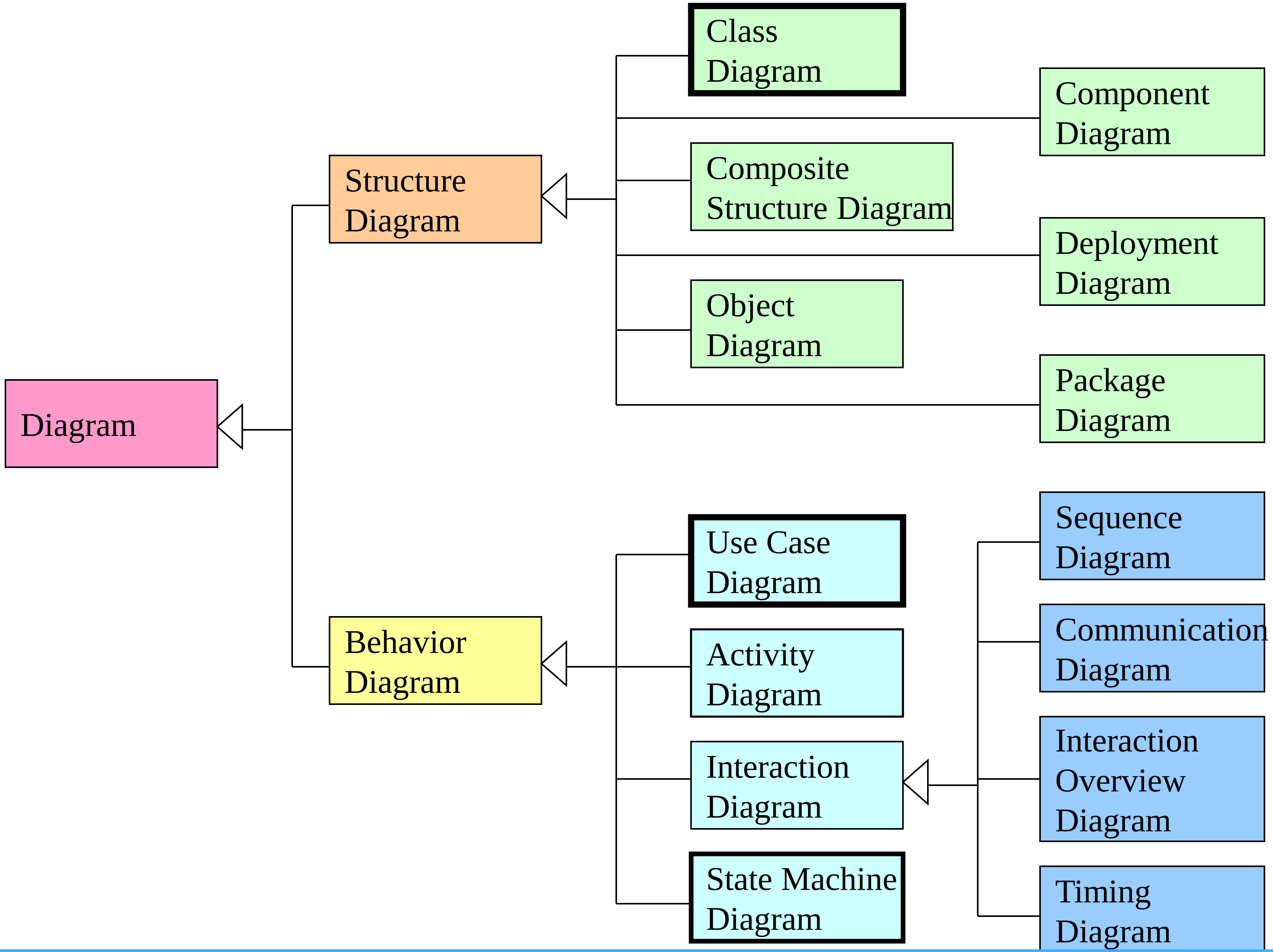
- Increase understanding/communication of product to customers and developers.

- Support for UML in many software packages today (e.g. Rational, plugins for popular IDE's like NetBeans, Eclipse).

UML Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

Classification of Diagram Types



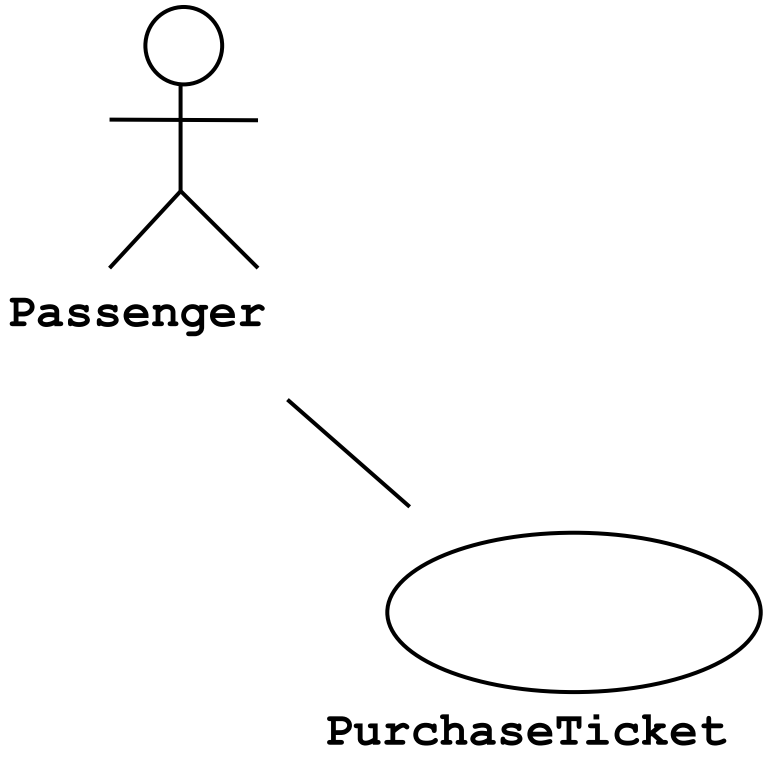
Basic Modeling Steps

- Use Cases
  - Capture requirements
- Analysis Model
  - Capture process, key classes
- Design Model
  - Capture details and behaviors of use cases and domain objects
  - Add classes that do the work and define the architecture

Use Case Diagram

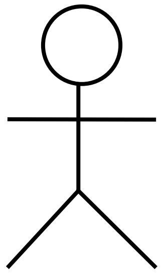
- Used for describing a set of user **scenarios**
- Mainly used for capturing user requirements
- Work like a **contract** between end user and software developers

Use Case Diagrams



- Used during requirements elicitation to represent external behavior.
- ***Actors*** represent roles, that is, a type of user of the system.
- ***Use cases*** represent a sequence of interaction for a type of functionality; summary of scenarios.
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



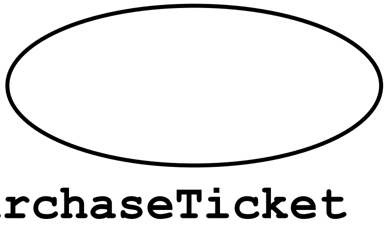
- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment

- An actor has a unique name and an optional description.

- **Examples:**
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

Use Case

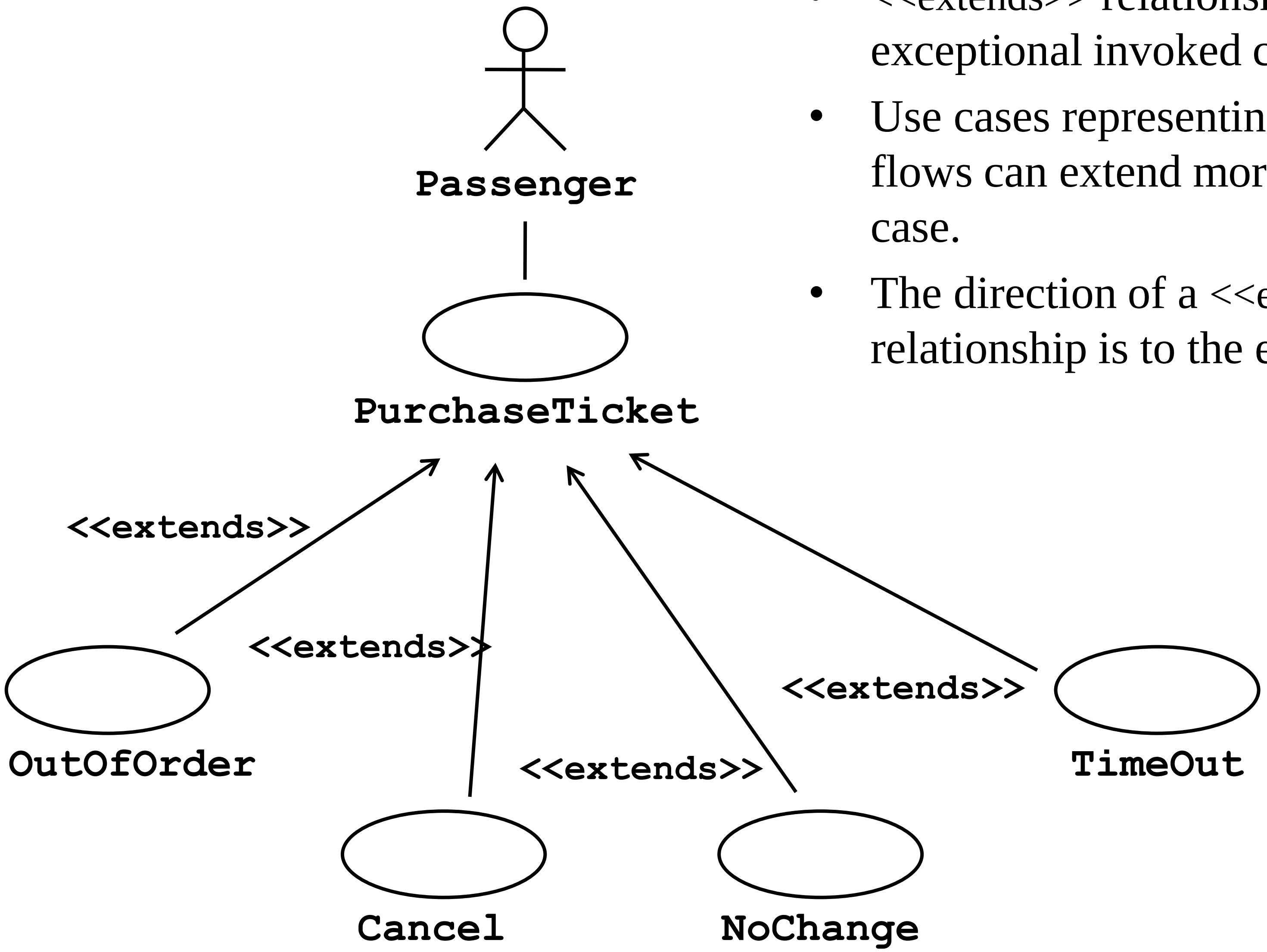
A use case represents a class of functionality provided by the system as an event flow.



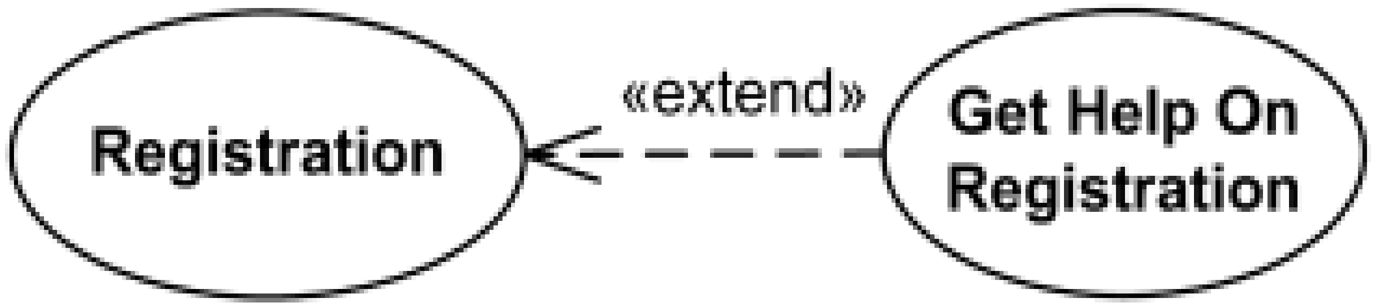
A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

The <<*extends*>> Relationship

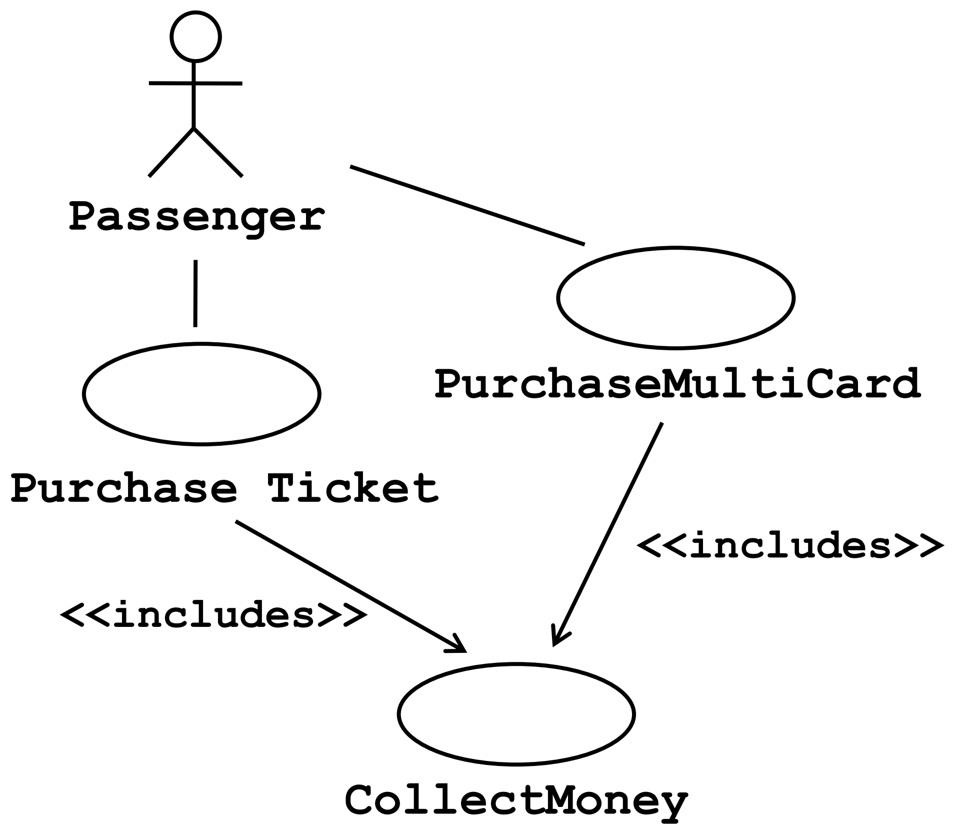


- <<extends>> relationships represent exceptional invoked cases.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

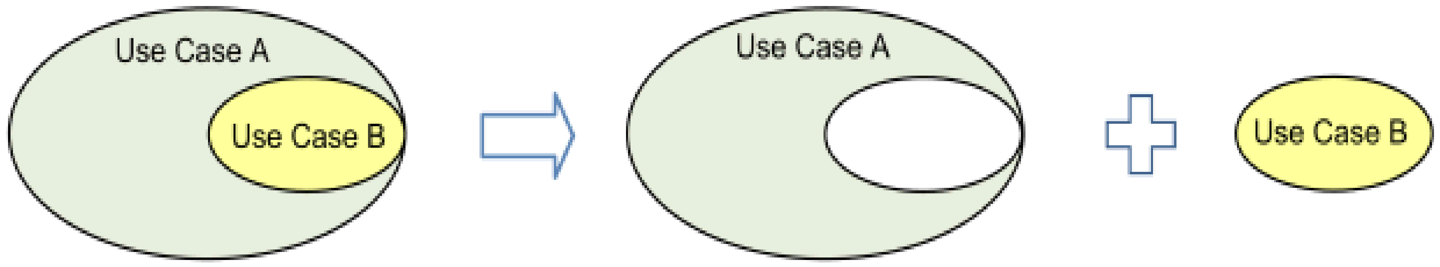


Registration *use case is complete and meaningful on its own.*
It could be extended with optional ***Get Help On Registration*** *use case.*

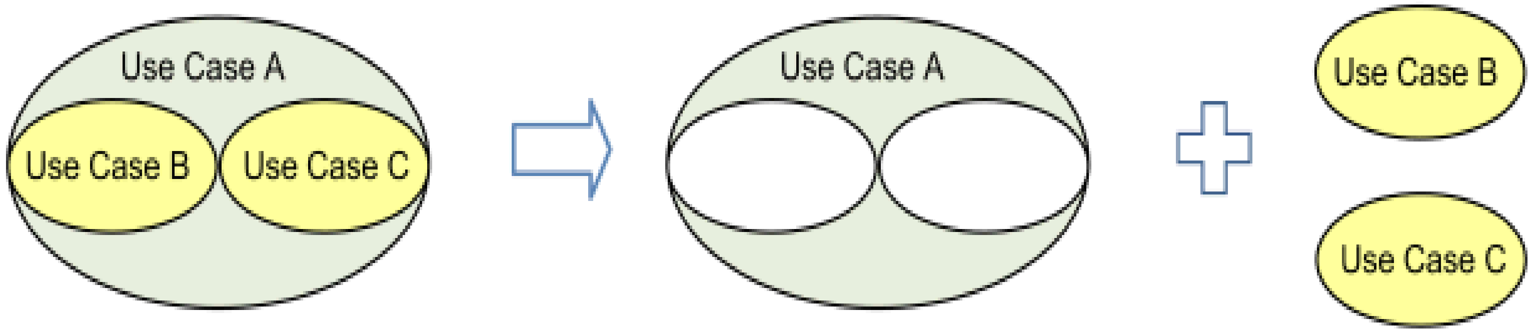
The <<*includes*>> Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).



Use case B is extracted from larger use case A into a separate use case.



Use cases B and C are extracted from larger use case A into separate use cases.

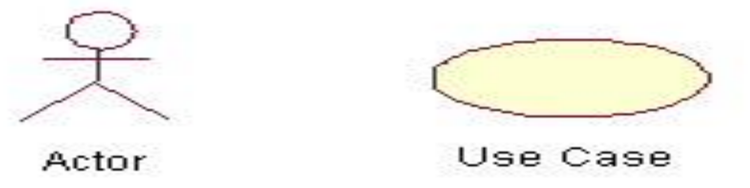
Use Cases are useful to…

- Determining requirements
  - New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients
  - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- Generating test cases
  - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Use Case Diagram (core components)

Actors: A role that a user plays with respect to the system,including human users and other systems. e.g.,in animate physical objects (e.g. robot); an external system that needs some information from the current system.

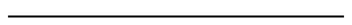
Use case: A set of scenarios that describing an interaction between a user and a system, including alternatives.



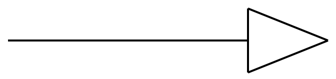
System boundary: rectangle diagram representing the boundary between the actors and the system.

Use Case Diagram(core relationship)

Association: communication between an actor and a use case; Represented by a solid line.



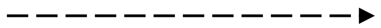
Generalization: relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.



Use Case Diagram(core relationship)

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” in stead of copying the description of that behavior.

<<include>>



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares “extension points”.

<<extend>>

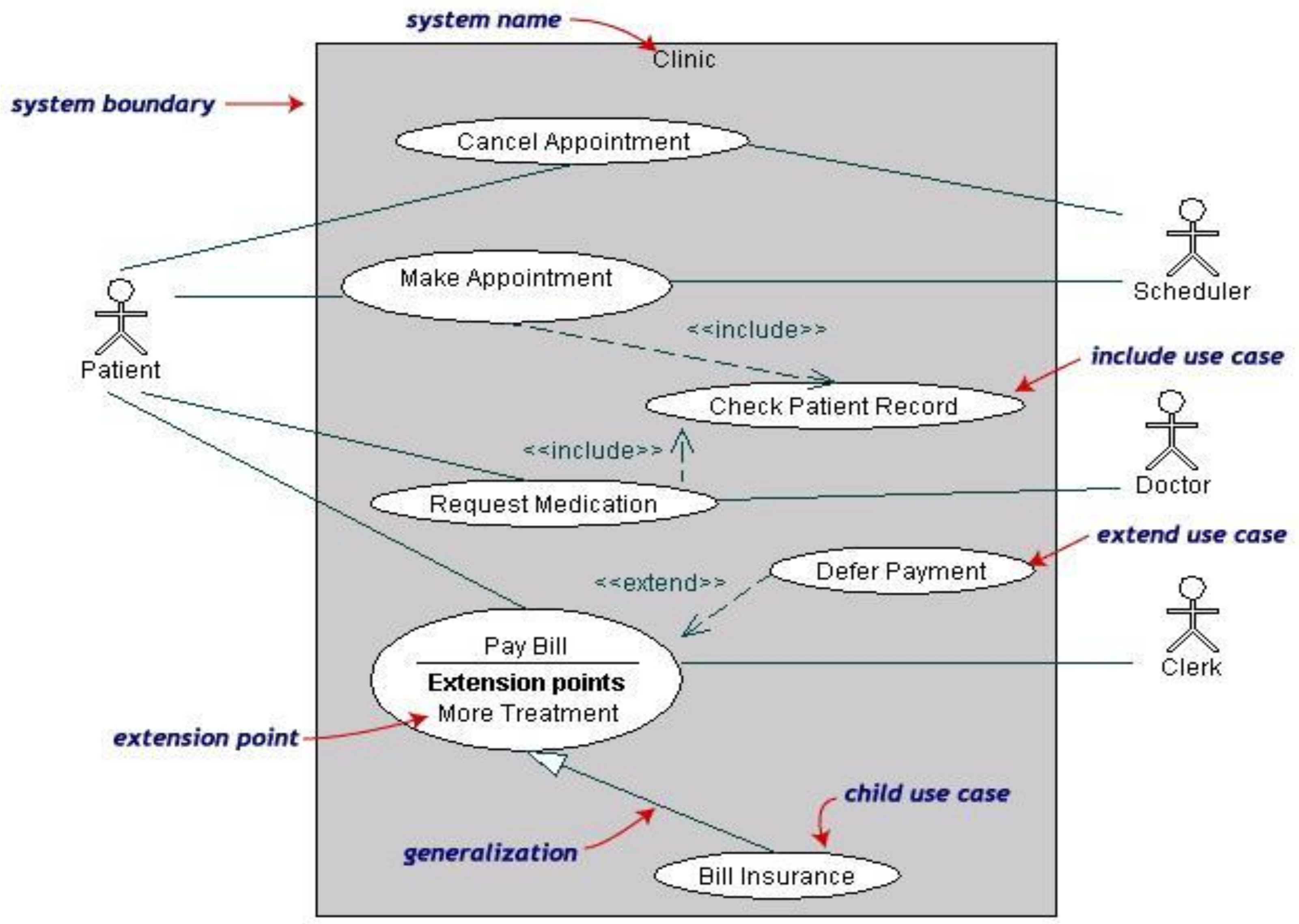
Use Case Diagrams(cont.)

•**Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)

•Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.(include)

•The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

Use Case Diagrams(cont.)



Use Case Diagrams

A <u>use case</u> is a *generic description of an entire transaction* involving several actors.

A <u>use case diagram</u> presents a *set of use cases* (ellipses) and the external actors that interact with the system.

Dependencies and *associations* between use cases may be indicated.

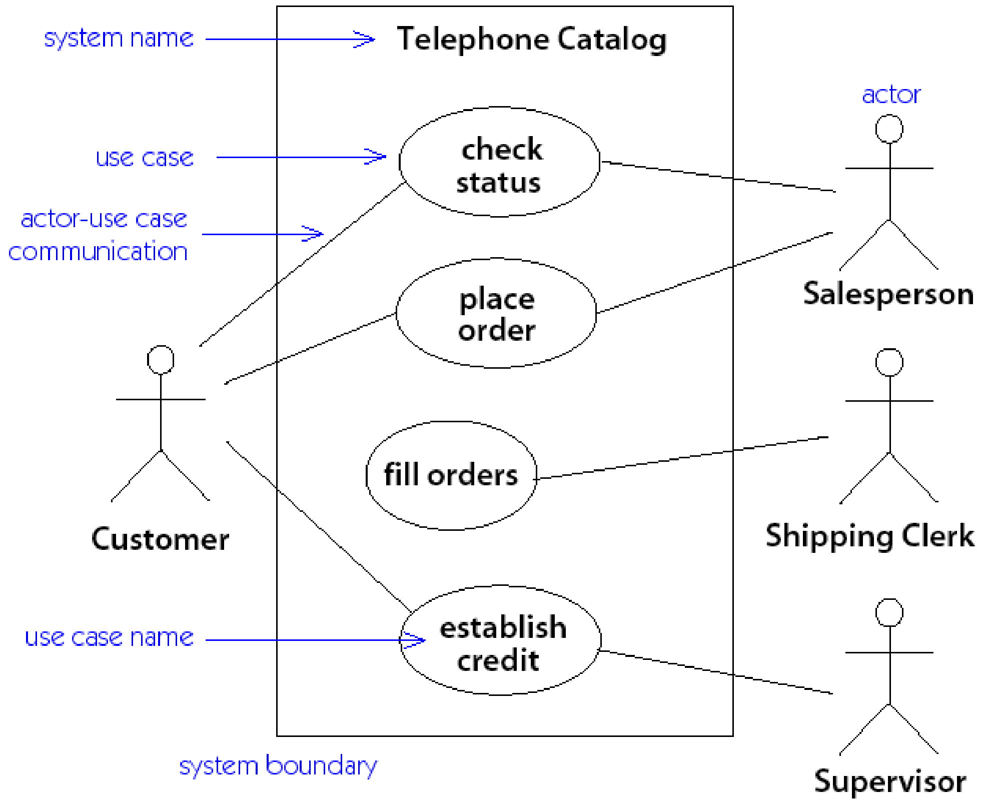
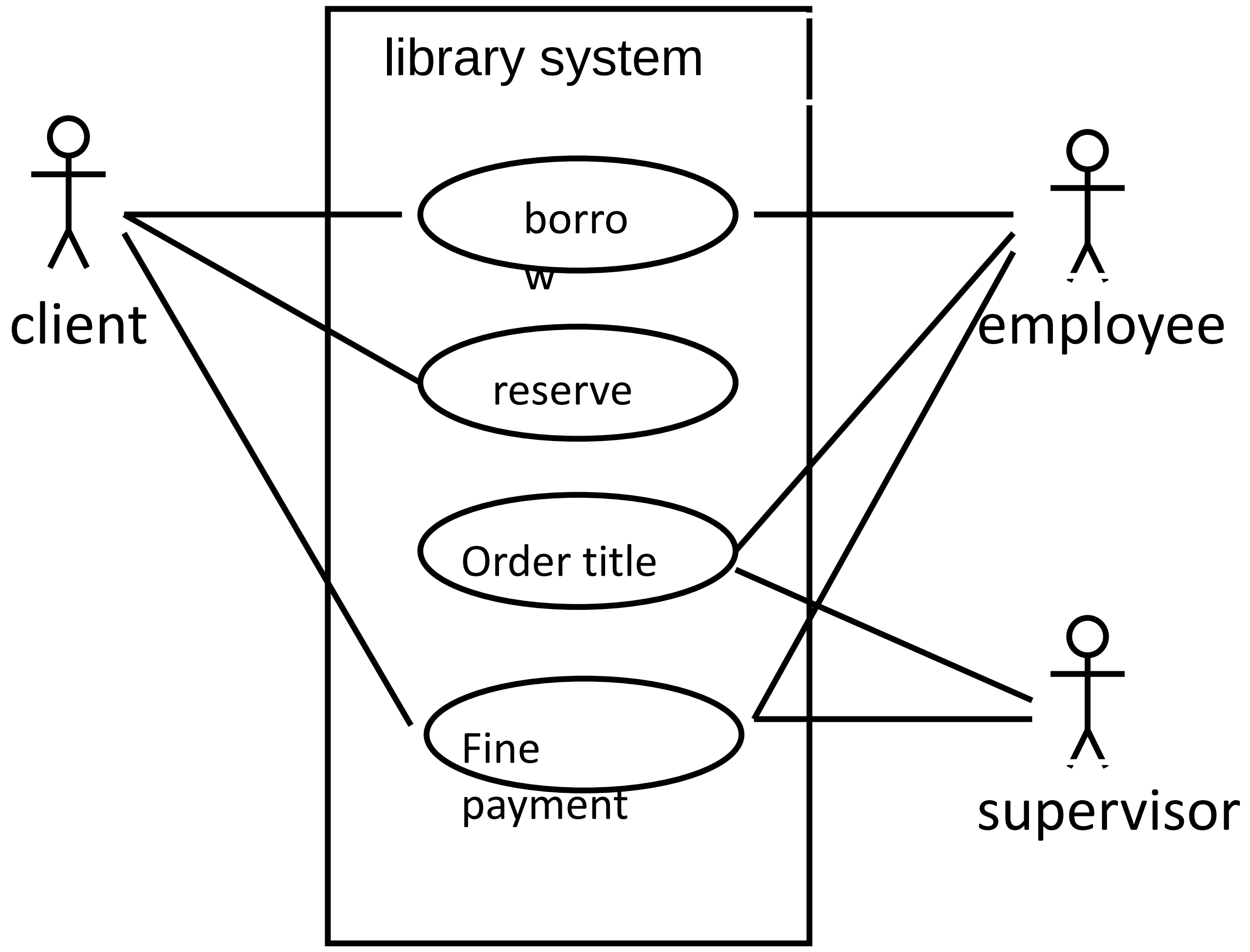


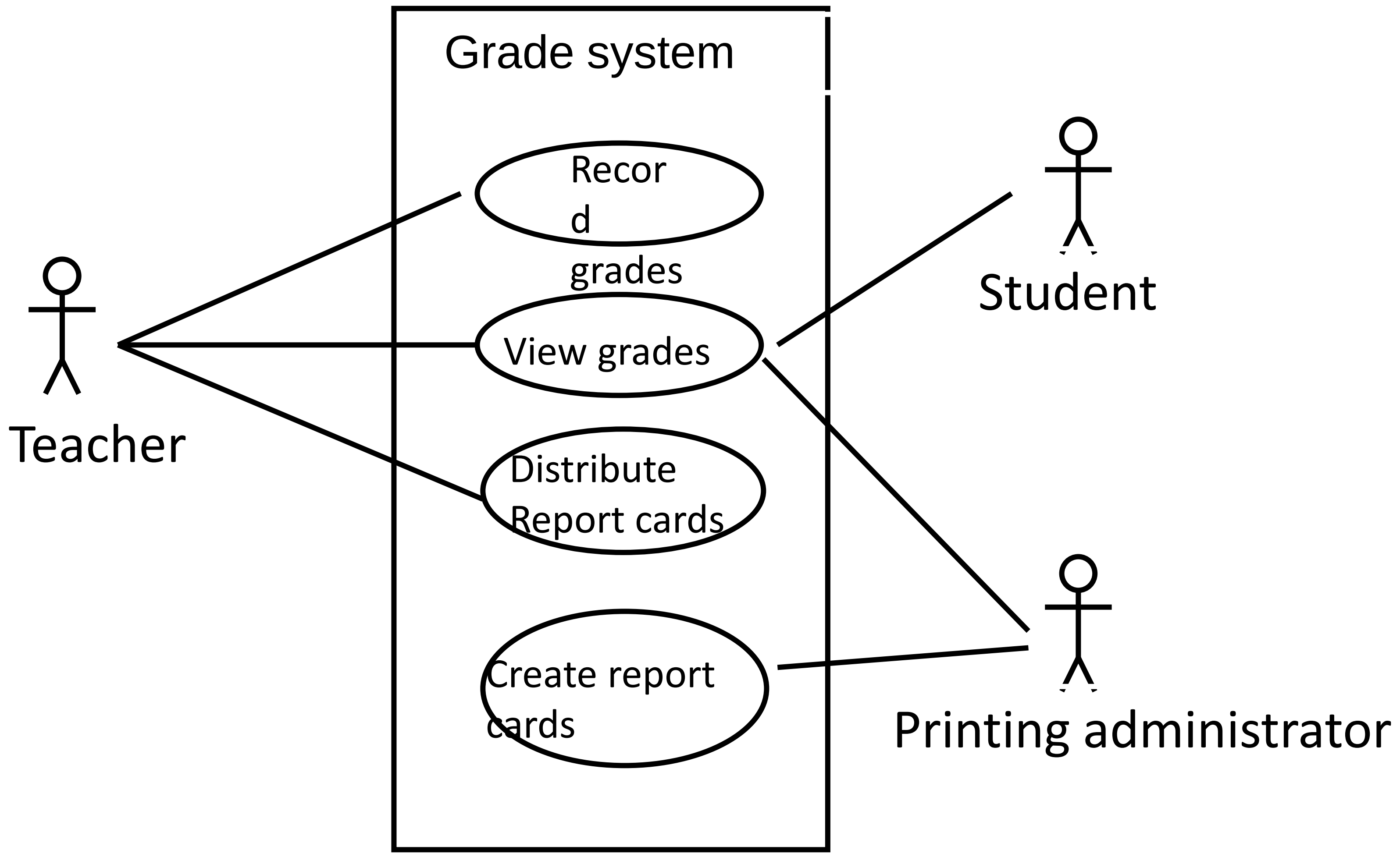
Figure 5-1. *Use case diagram*

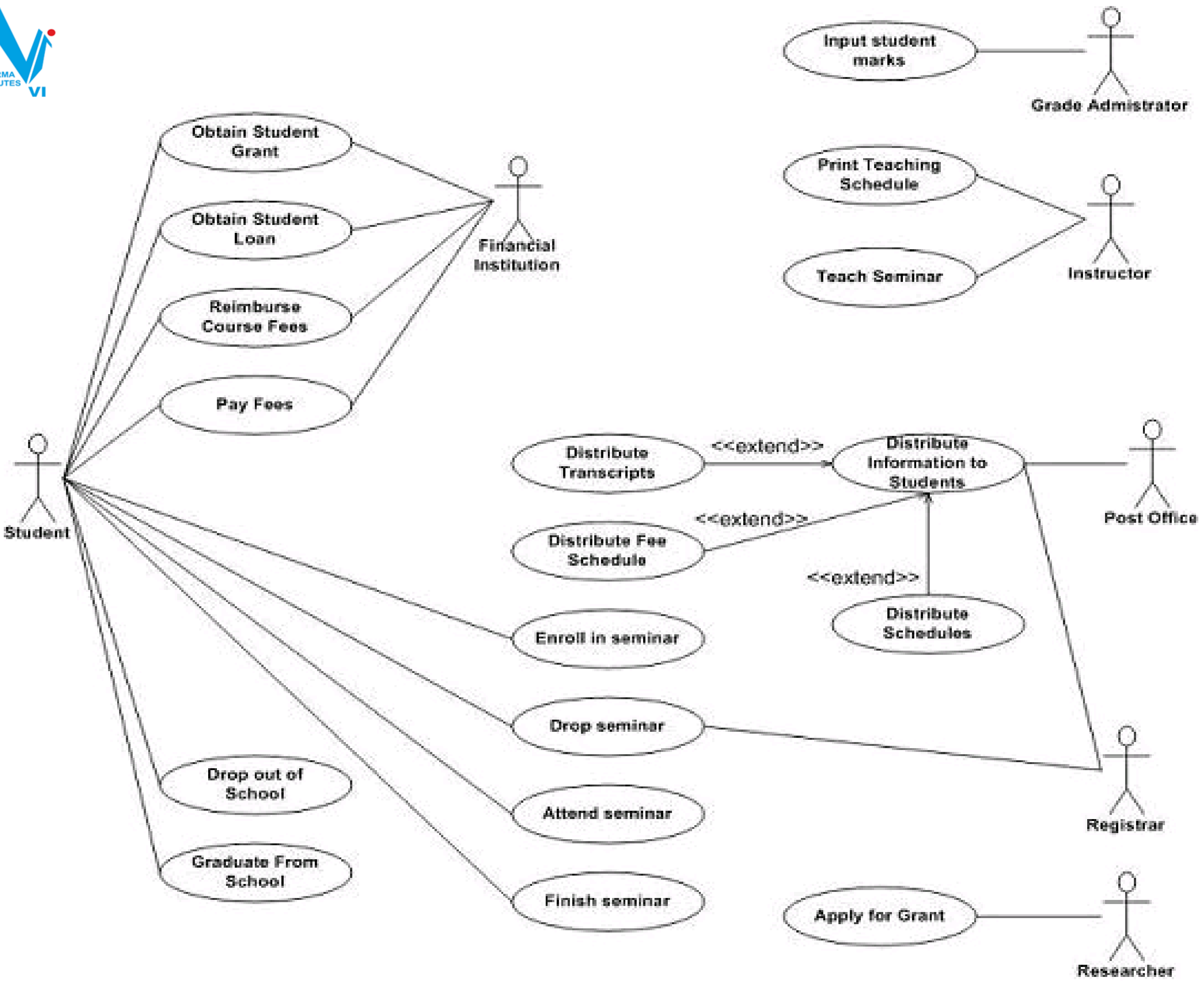
Use Case Diagram – Example1 (Library)



A Library System.

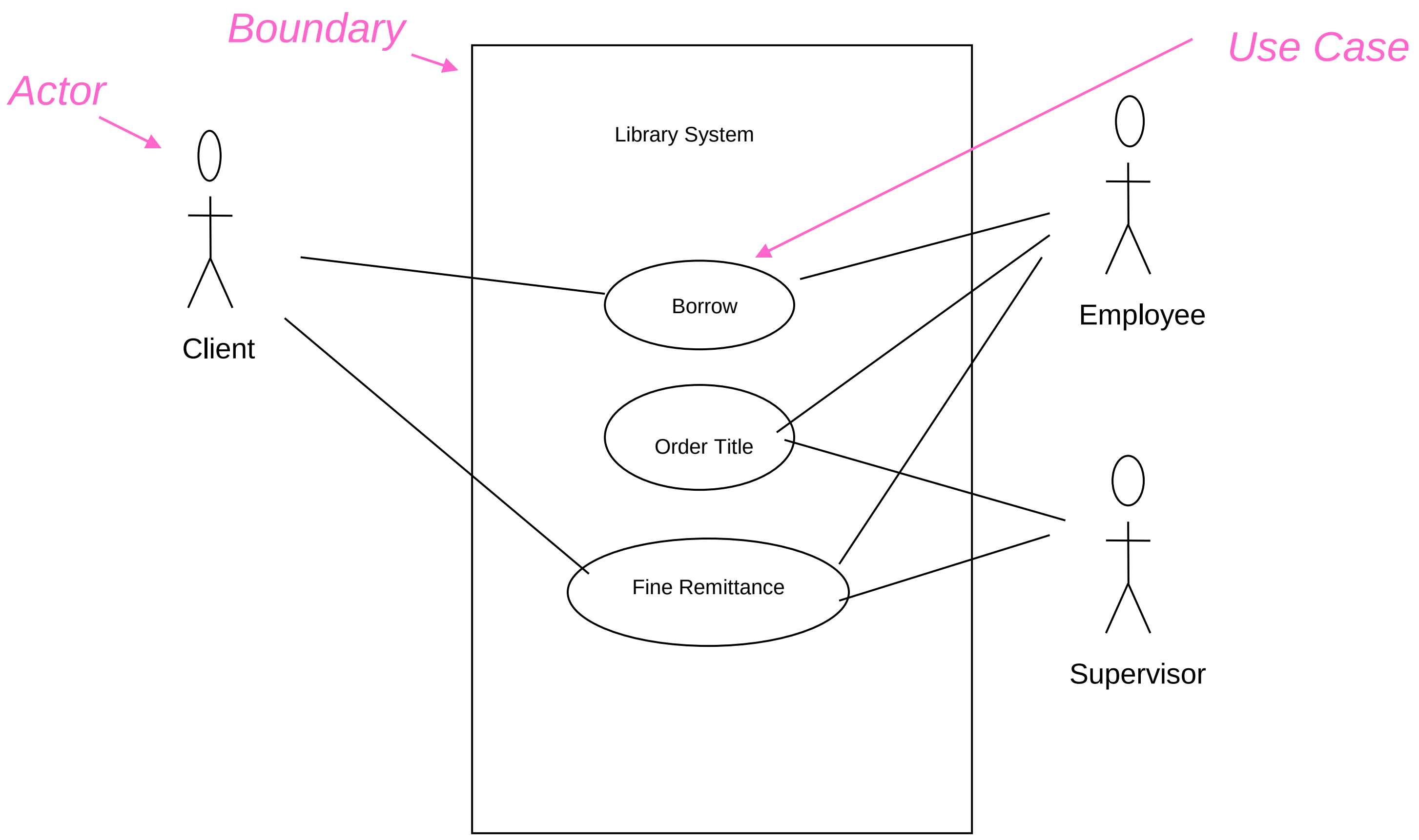
Use Case Diagram for Student Assessment Management System





Example Use-case Diagram for a student database

Use Case Diagrams



- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system

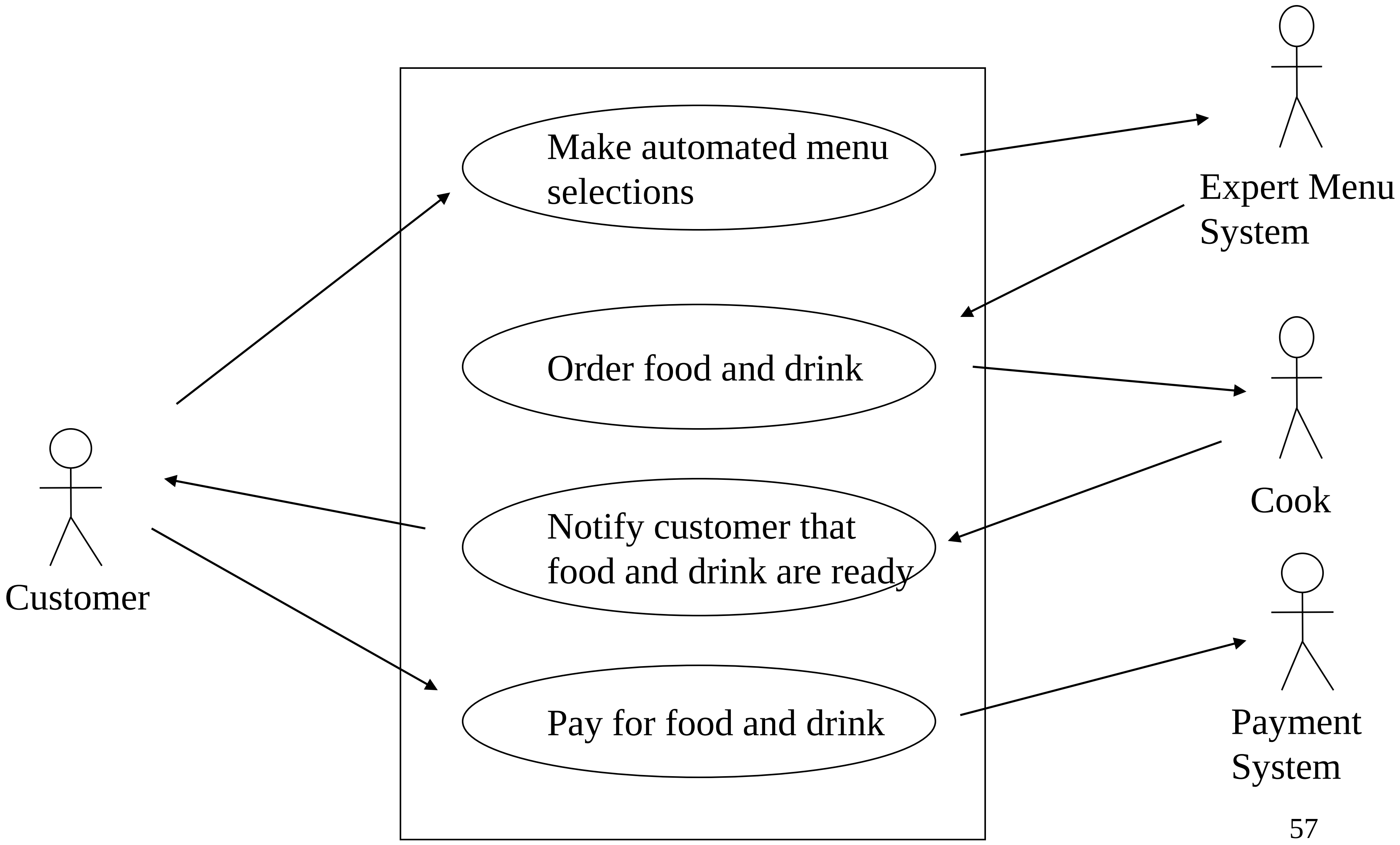
Use Case Diagram

- Use cases serve as a technique for capturing the functional requirements of a system
- Describes the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used
- A use case consists of a set of one or more scenarios tied together by a common user goal
- A scenario is a sequence of steps describing an interaction between a user and a system; some scenarios describe successful interaction; others describe failure or errors
- Users are referred to as actors; an actor is a role that carries out a use case
- An actor need not always be a person; it can also be an external system that is either automated or manual

- A use case diagram is like a graphical table of contents of the use cases for a system
  - It shows the use cases, the actors, and the relationships between them

- Use cases represent an external view of the system; consequently, they have no correlation to the classes in the system
  - They can serve as a starting point for writing software validation test cases

Example Use Case Diagram



Class diagram

- Class diagram is a **static diagram**. It represents the static view of an application.

- Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

- Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

- Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

➢ What is Class?

- A Class is a blueprint that is used to create Object. The Class defines what object can do.

➢ What is Class Diagram?

- Class Diagram gives the static view of an application. A class diagram describes the types of objects in the system and the different types of relationships that exist among them. This modeling method can run with almost all Object-Oriented Methods.
- UML Class Diagram gives an overview of a software system by displaying classes, attributes, operations, and their relationships. This Diagram includes the class name, attributes, and operation in separate designated compartments.
- Class Diagram helps construct the code for the software application development.

Benefits

- Class Diagram Illustrates data models for even very complex information systems.
- It provides an overview of how the application is structured before studying the actual code. This can easily reduce the maintenance time.
- It helps for better understanding of general schematics of an application.
- Allows drawing detailed charts which highlights code required to be programmed.
- Helpful for developers and other stakeholders.

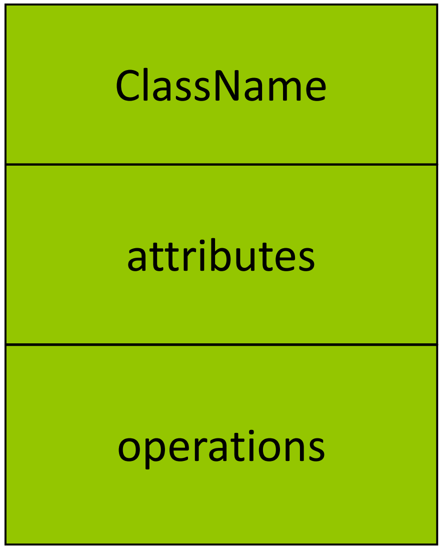
➢ **Essential elements of A UML class diagram are:**

✓ Class Name

✓ Attributes

✓ Operations

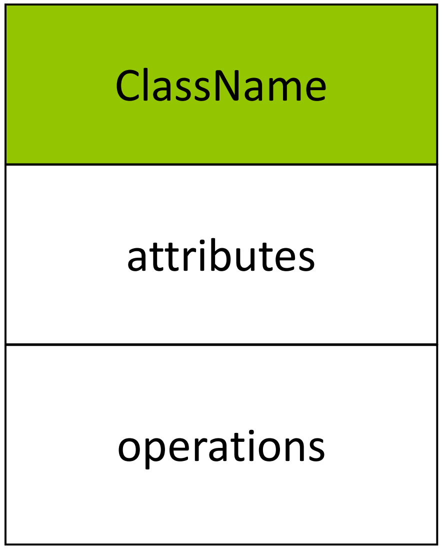
Classes



A ***class*** is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

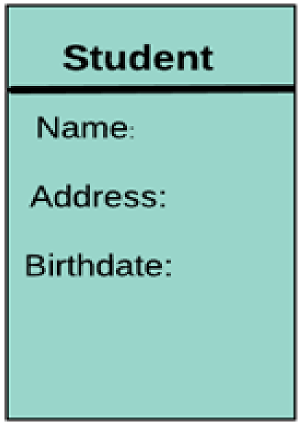
Class Names



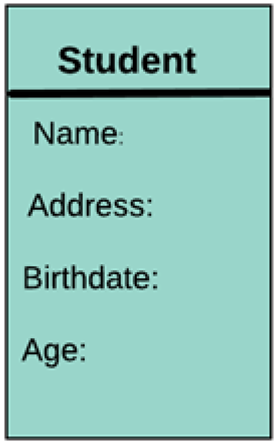
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Attributes

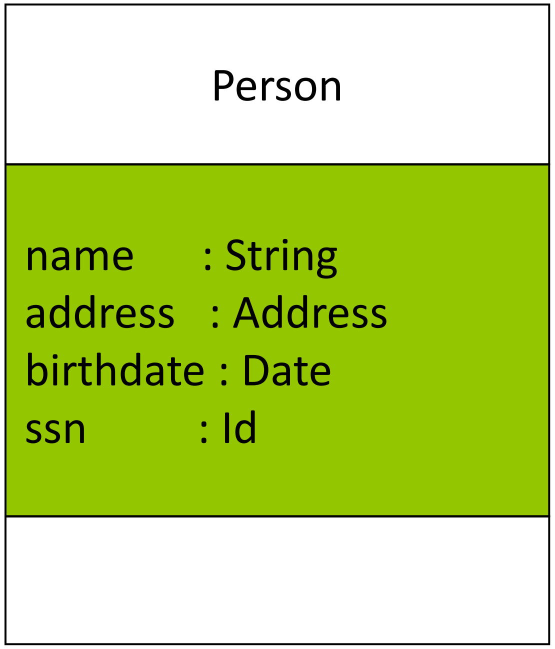
An attribute is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.



A derived attribute is computed from other attributes. For example, an age of the student can be easily computed from his/her birth date.



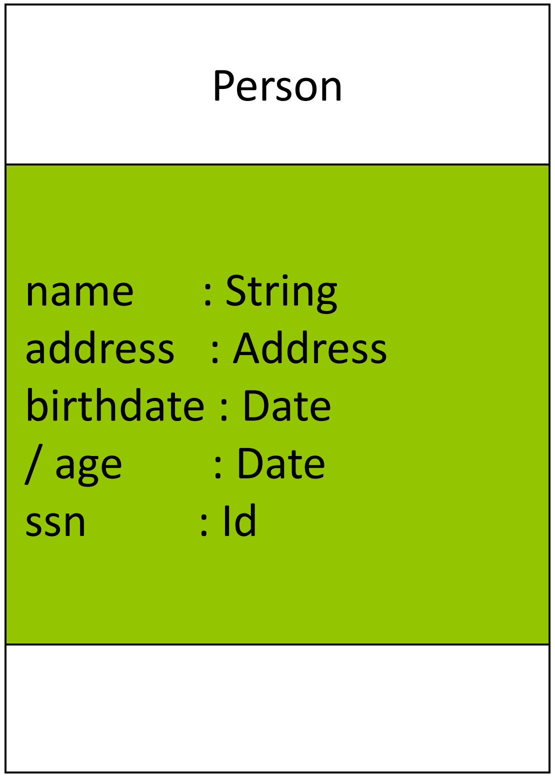
Class Attributes



An *attribute* is a named property of a class that describes the object being modeled.

In the class diagram, attributes appear in the second compartment just below the name-compartment.

Class Attributes (Cont'd)



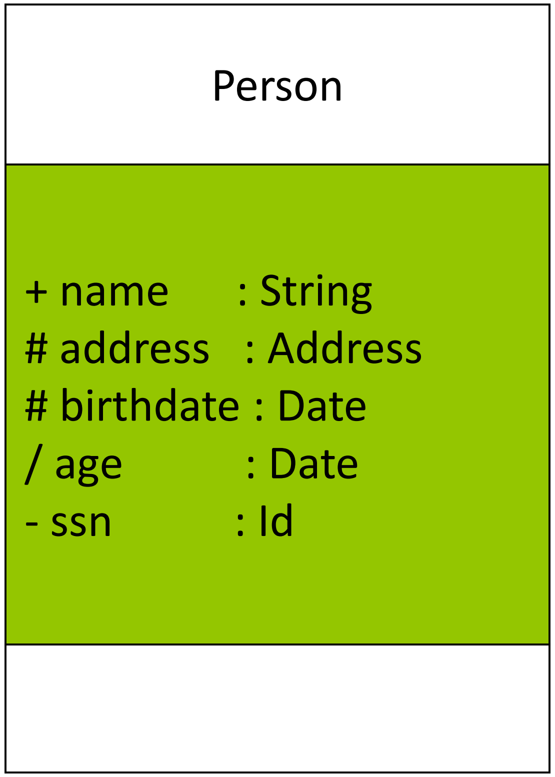
Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

Class Attributes (Cont'd)



Attributes can be:
+ public
# protected
- private
/ derived

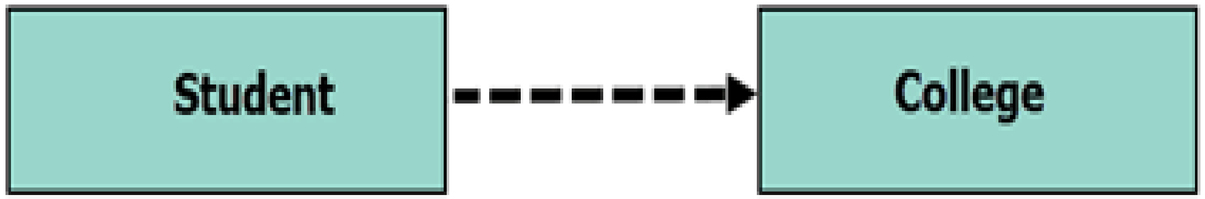
Relationships

There are mainly three kinds of relationships in UML:

- Dependencies
- Generalizations
- Associations

Dependency

- A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.
- In the following example, Student has a dependency on College



Generalization:

- A generalization helps to connect a subclass to its superclass.
- A sub-class is inherited from its superclass. Generalization relationship can't be used to model interface implementation.
- Class diagram allows inheriting from multiple superclasses.
- It indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and the superclass is considered a '***Generalization***' of the subclass.
- This means that any instance of the subtype is also an instance of the superclass.
- The UML graphical representation of a Generalization is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes.
- The generalization relationship is also known as the *inheritance* or *"is a"* relationship.

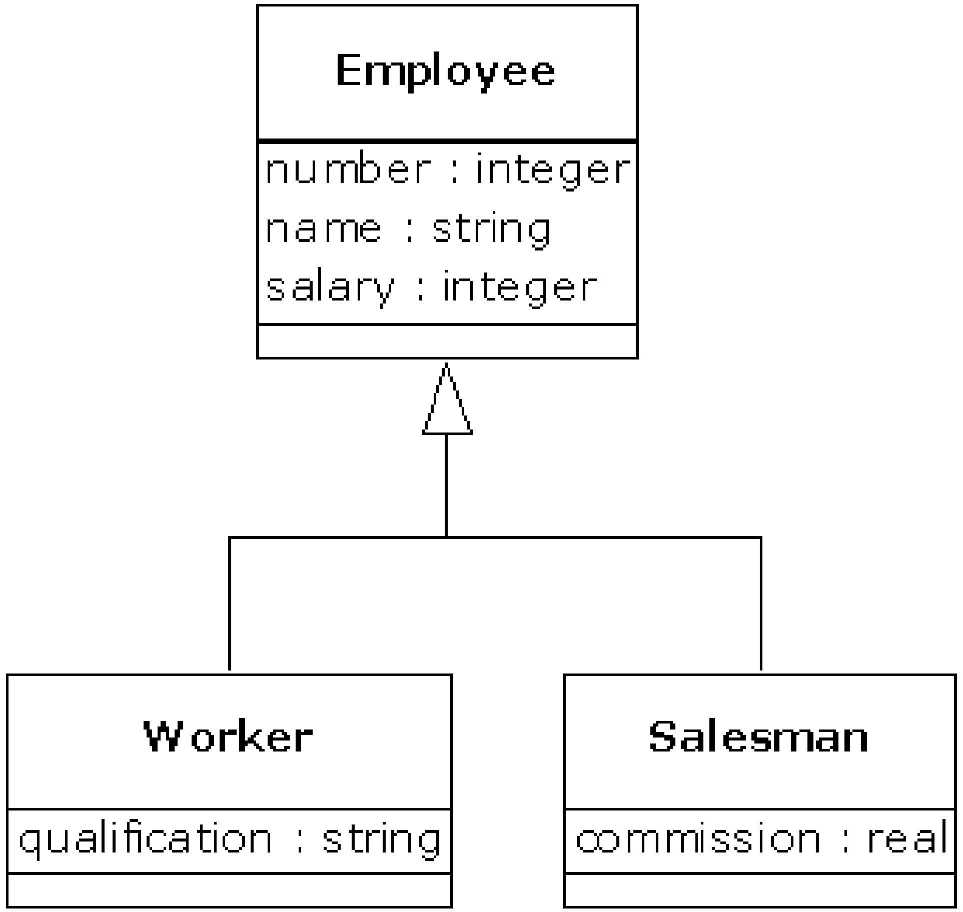
- The *superclass* (base class) in the generalization relationship is also known as the *"parent"*, *superclass*, *base class*, or *base type*.

- The *subtype* in the specialization relationship is also known as the *"child"*, *subclass*, *derived class*, *derived type*, *inheriting class*, or *inheriting type*.

- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships

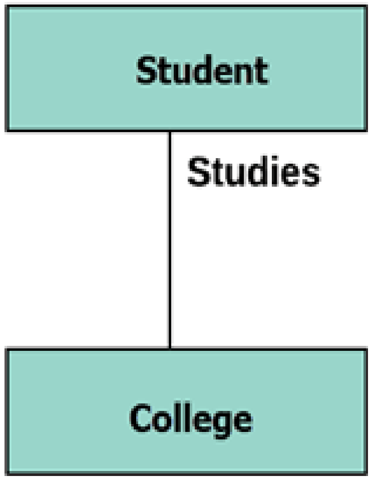
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operations

Example


Employee
number : integer
name : string
salary : integer
Worker
qualification : string
Salesman
commission : real

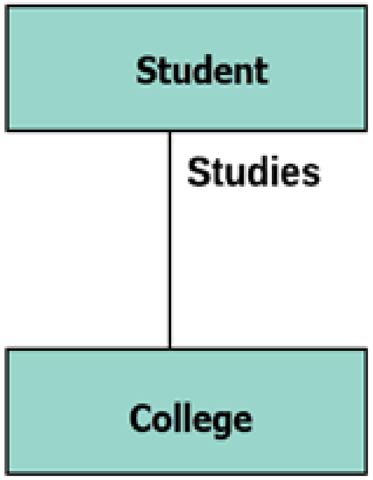
Association:

- This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization. Here are some rules for Association:
- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.
- Mandatory for reflexive associations
- In this example, the relationship between student and college is shown which is studies.



Association:

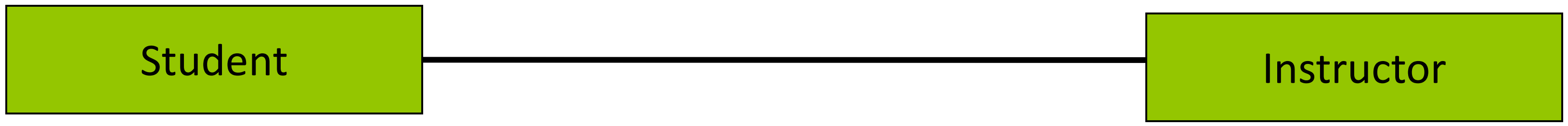
- This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization. Here are some rules for Association:
- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.
- Mandatory for reflexive associations
- In this example, the relationship between student and college is shown which is studies.



Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

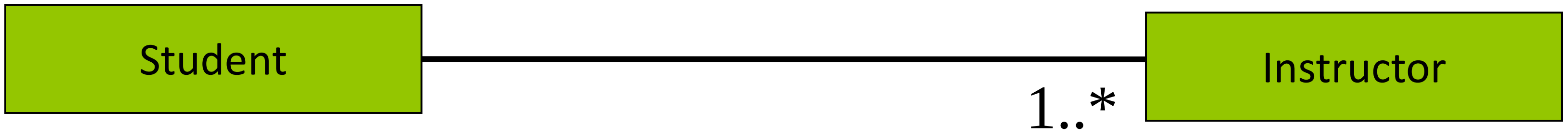
An *association* denotes that link.



Association Relationships (Cont'd)

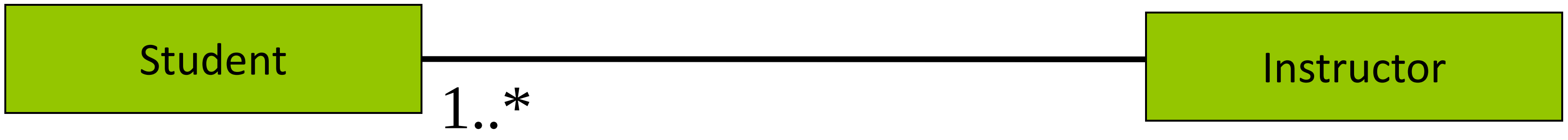
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a ***Student*** **has one or more *Instructors*:**



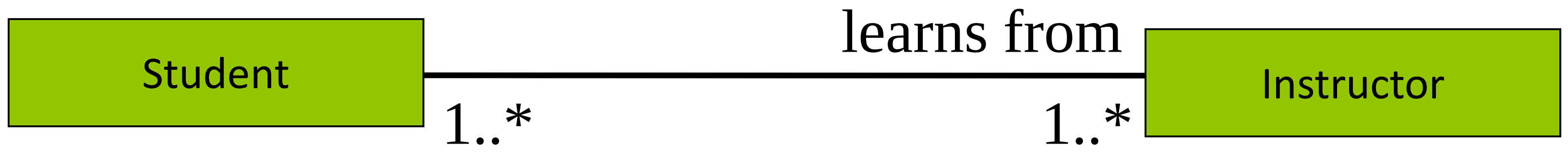
Association Relationships (Cont’d)

The example indicates that **every *Instructor* has one or more *Students*:**



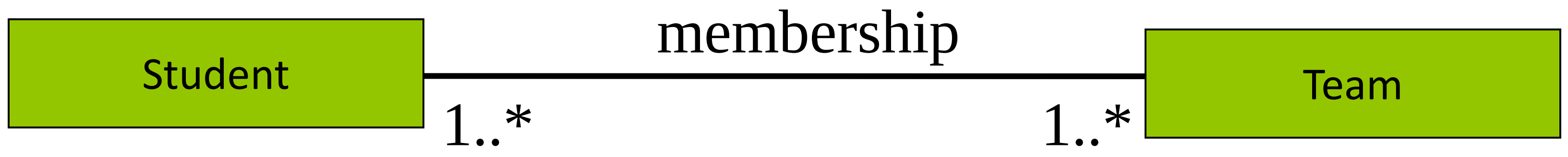
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



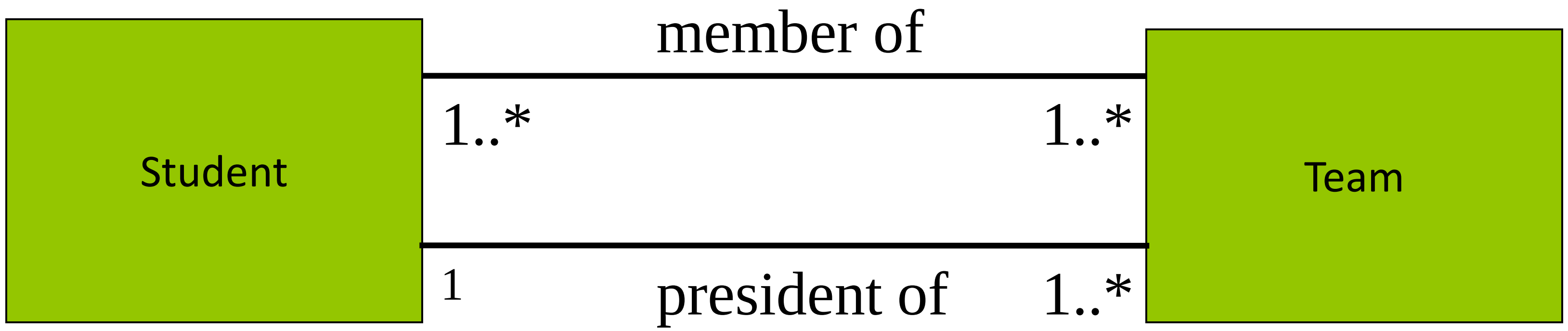
Association Relationships (Cont'd)

We can also name the association.



Association Relationships (Cont'd)

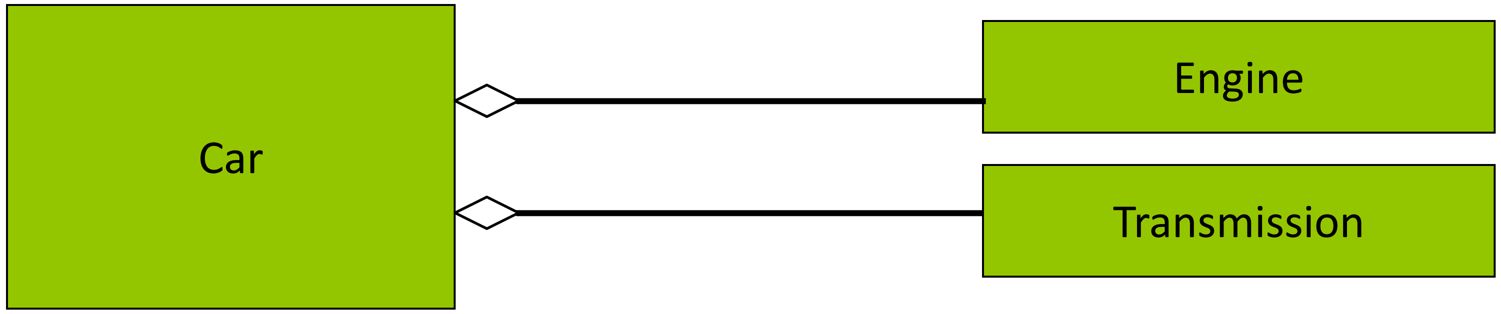
We can specify dual associations.



Association Relationships (Cont'd)

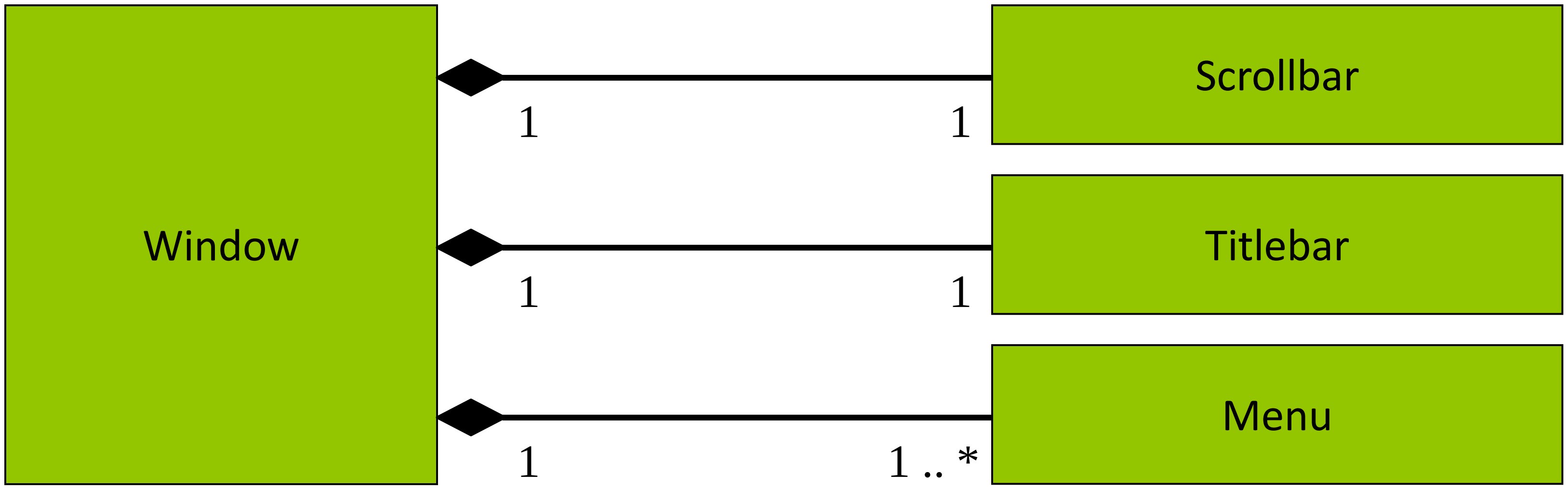
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

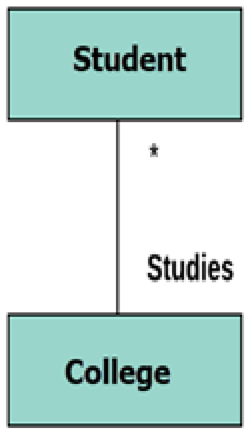


Association Relationships (Cont’d)

Compositions are denoted by a filled-diamond adornment on the association.

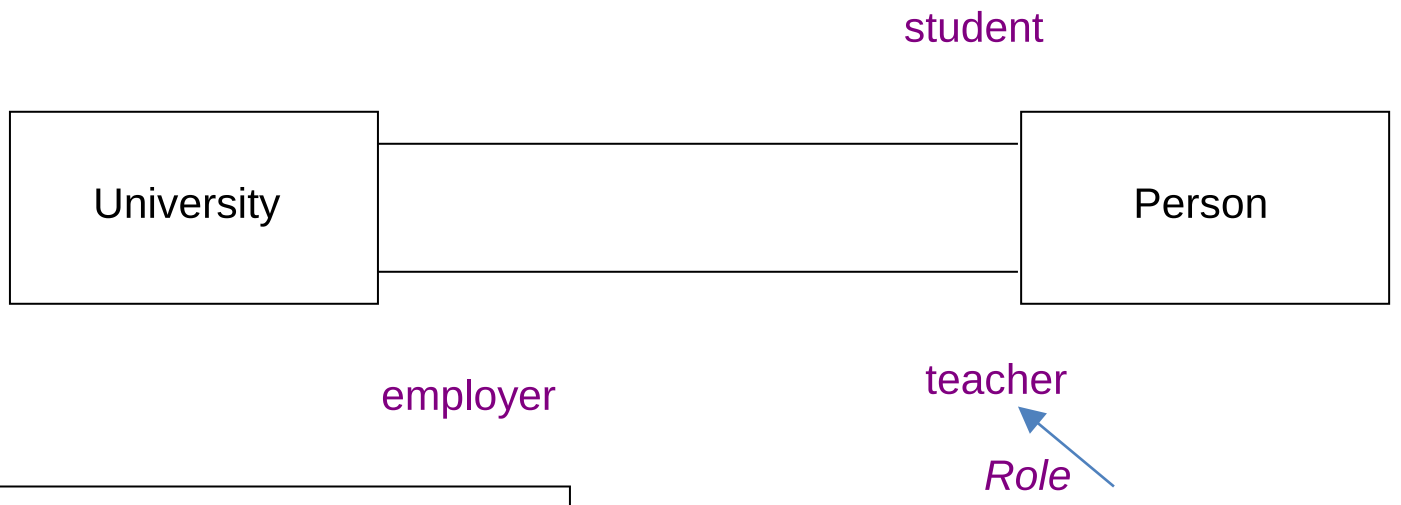


Multiplicity



A multiplicity specifies whether the association is mandatory or not. It has some lower bound and upper bound.
Let's say that that there are 100 students in one college. The college can have multiple students.

Association: Multiplicity and Roles



Multiplicity

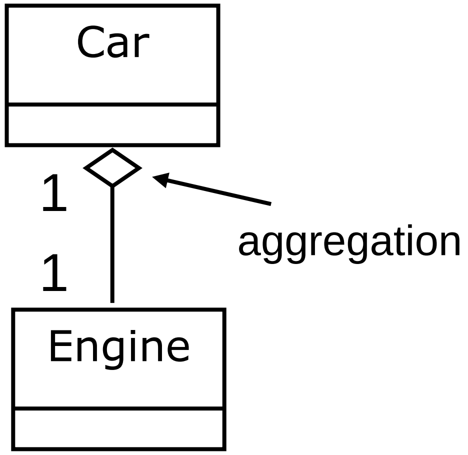
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

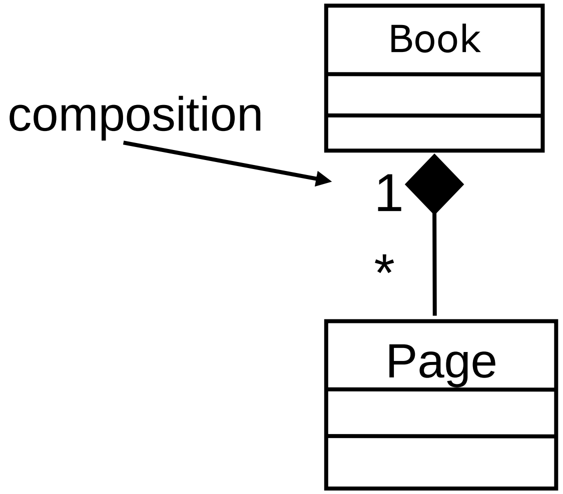
Role

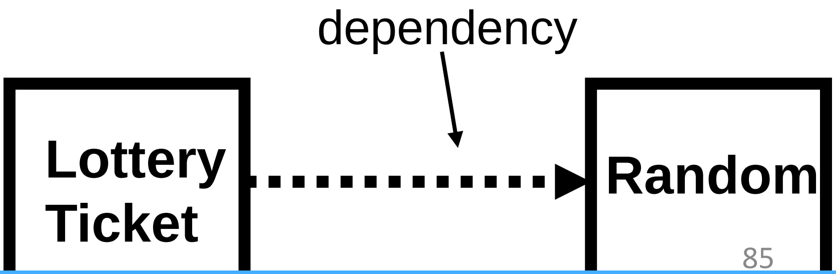
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Association types

- **aggregation**: "is part of"
  - symbolized by a clear white diamond
- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond
- **dependency**: "uses temporarily"
  - symbolized by dotted line
  - often is an implementation detail, not an core part of that object's state







Class Diagram-Example

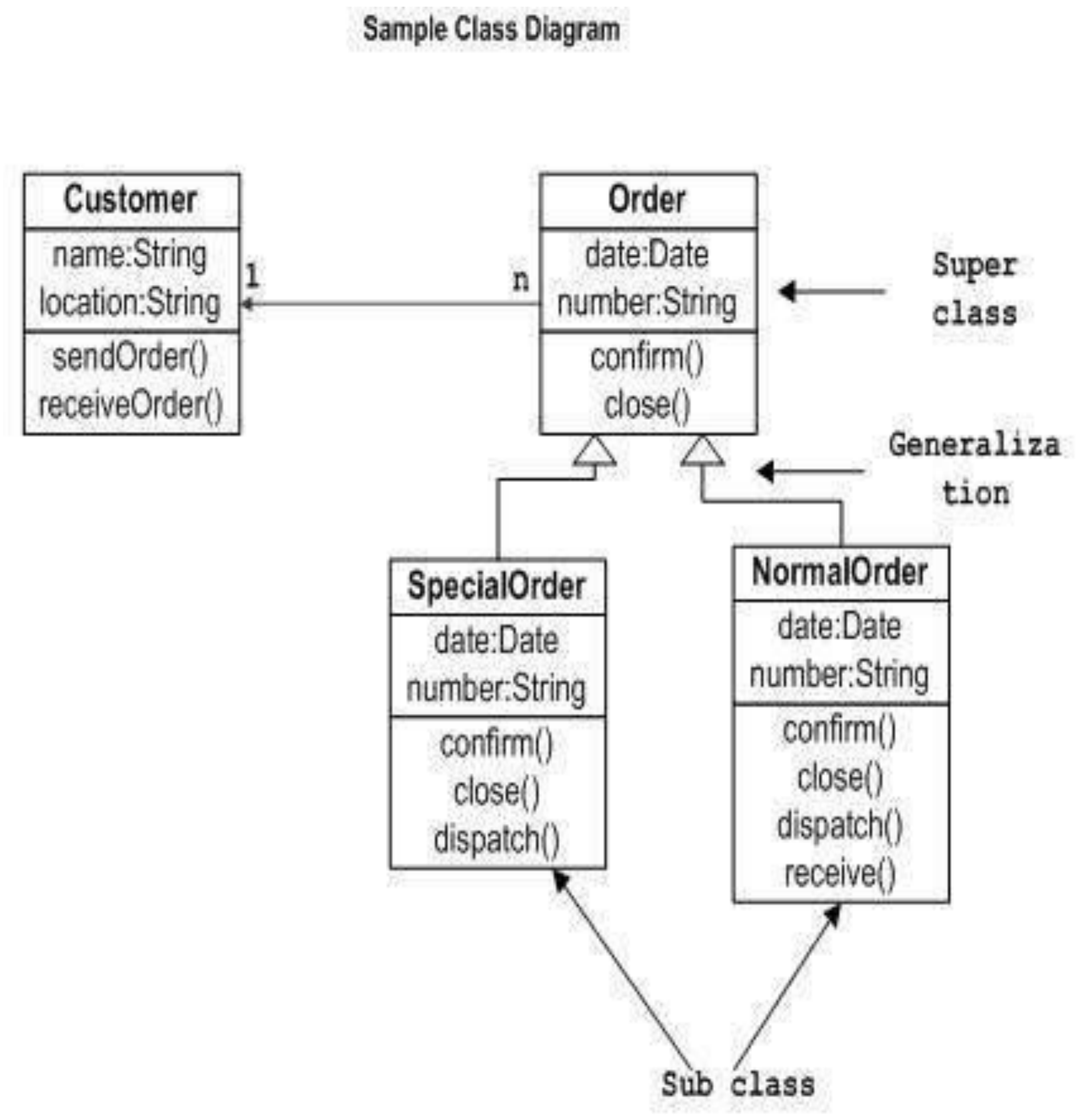
- An example of an ***Order System*** of an application. It describes a particular aspect of the entire application.

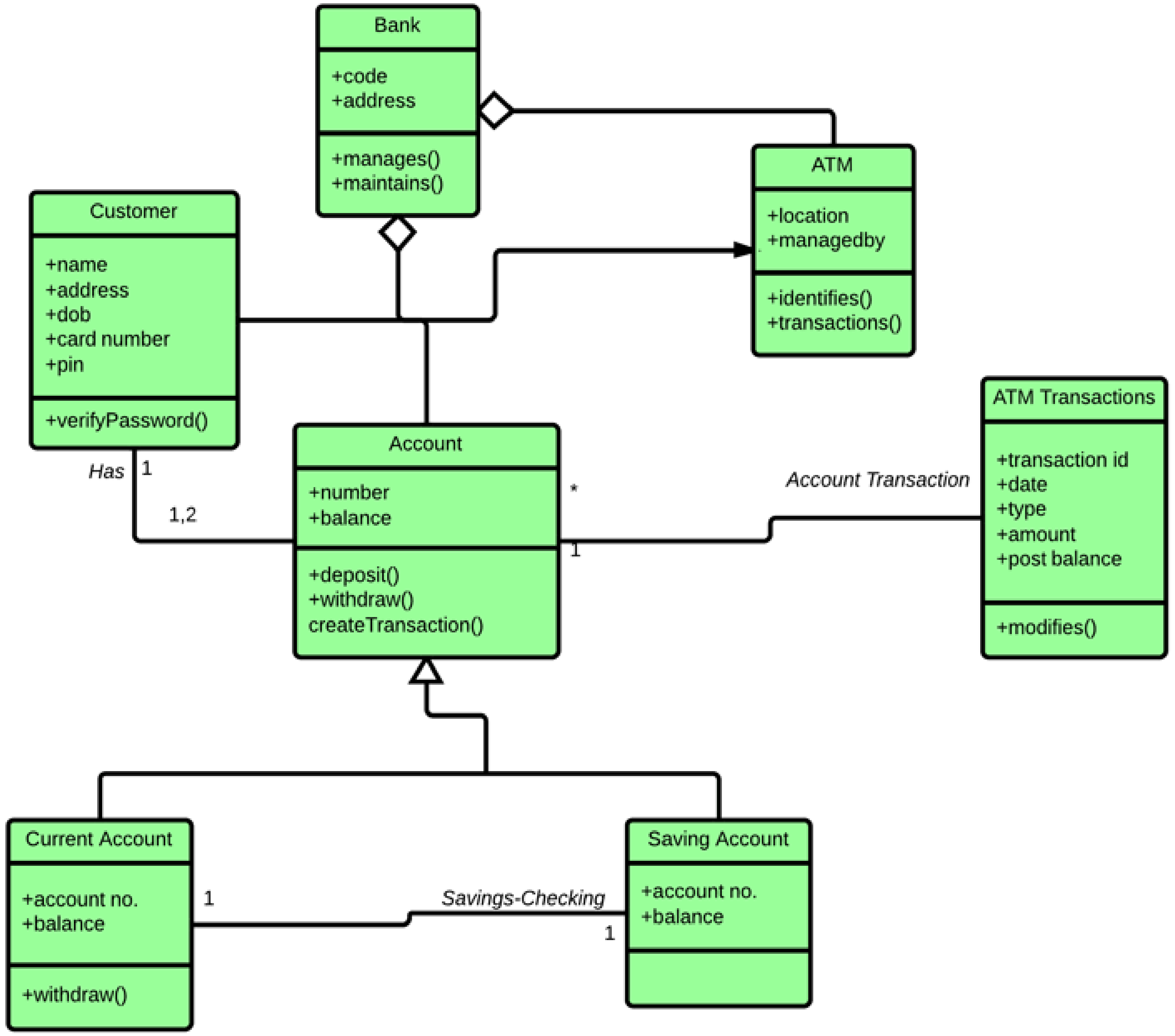
- First of all ***Order* and *Customer*** are identified as the two elements of the system and they have a ***one to many* relationship** because a customer can have multiple orders.

- Keep ***Order*** class is an abstract class and it has two concrete classes (inheritance relationship) ***SpecialOrder* and *NormalOrder*.**

- The two inherited classes have all the properties as the ***Order*** class. In addition they have additional functions like ***dispatch () and receive ()*.**

Class Diagram-Example




Bank
+code
+address
+manages()
+maintains()
ATM
+location
+managedby
+identifies()
+transactions()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
createTransaction()
*
1
Account Transaction
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Current Account
+account no.
+balance
+withdraw()
1
Savings-Checking
1
Saving Account
+account no.
+balance

Behavioral modelling using state diagrams - real life application case study

- A behavioral model shows the interactions between objects to produce some particular system behavior that is specified as a use-case, Sequence diagrams, Activity diagrams, state diagrams, collaboration diagrams etc.

- Above diagrams are used to model interaction between objects.

UML State diagrams

➢A state diagram, also called a state machine diagram or statechart diagram, is an illustration of the states an object can attain as well as the transitions between those states in the Unified Modeling Language (UML).

➢**The following are the basic notational elements that can be used to make up a diagram:**

•**Filled circle,** representing to the initial state. **Hollow circle** containing a smaller filled circle, indicating the final state (if any)

•**Rounded rectangle,** denoting a state.

•Top of the rectangle contains a name of the state.

•Can contain a horizontal line in the middle, below which the activities that are done in that state are indicated.

•**Arrow,** denoting transition. The name of the event (if any) causing this transition labels the arrow body.

•Any real time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.

•Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface etc.

•State chart diagram is used to visualize the reaction of a system by internal/external factors.

•Statechart diagram is used to describe the states of different objects in its life cycle. So the emphasis is given on the state changes upon some internal or external events.

•These states of objects are important to analyze and implement them accurately.

•Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

•Before drawing a Statechart clear the following points:

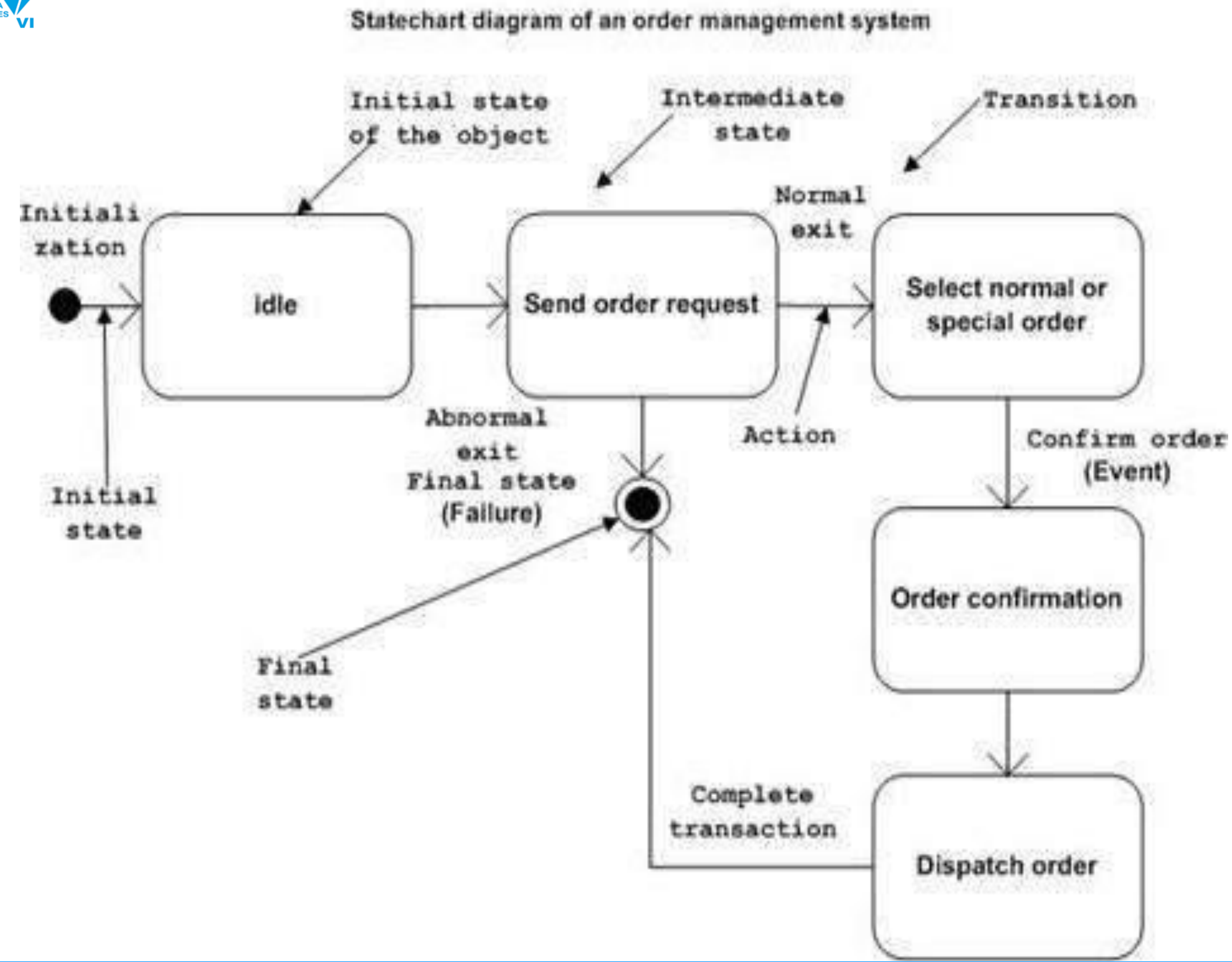
✓Identify important objects to be analyzed.

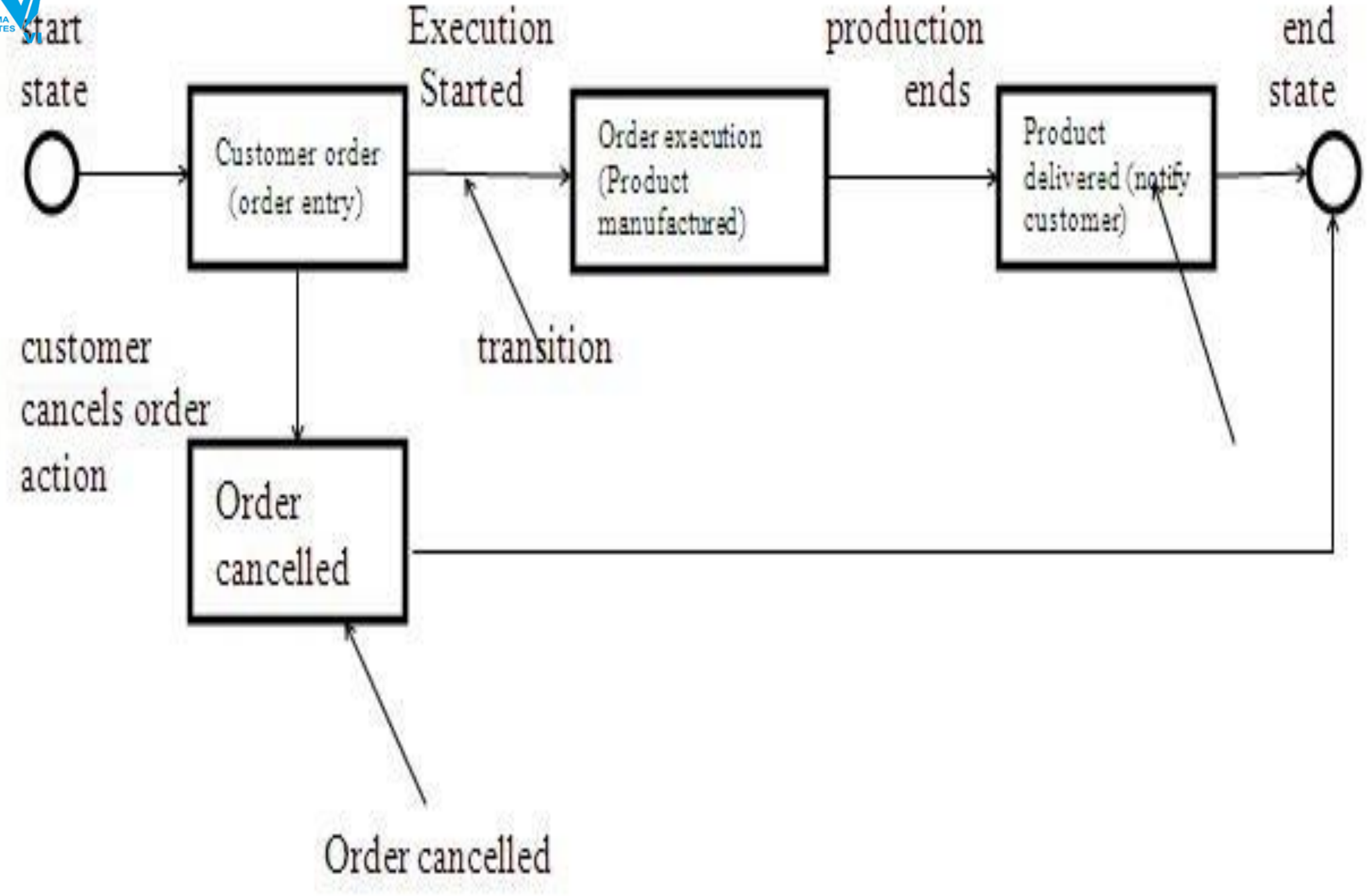
✓Identify the states.

✓Identify the events.

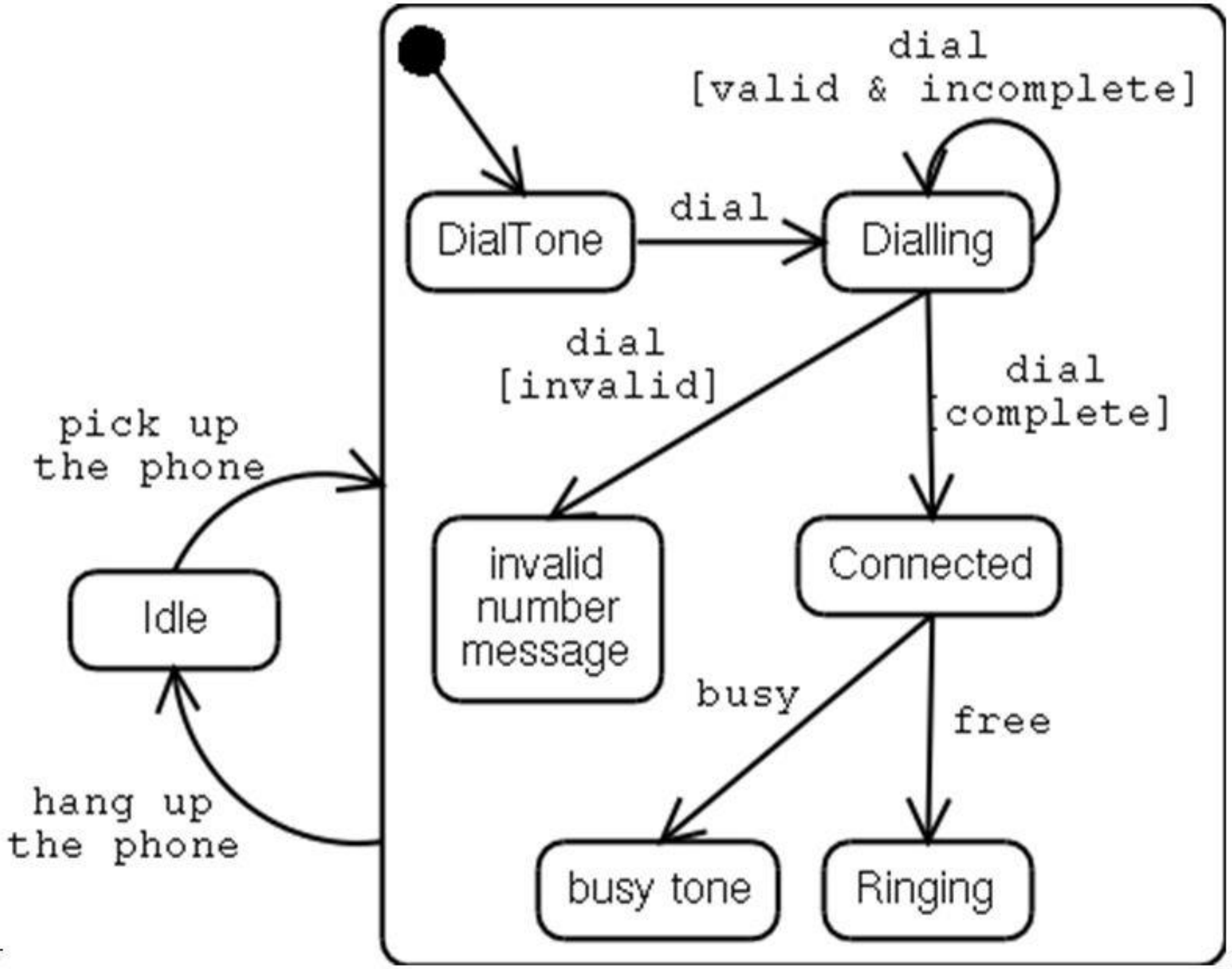
State diagram example

•An example of a **Statechart** diagram where the state of ***Order*** object is analyzed.
•The first state is an idle state from where the process starts.
•The next states are arrived for events like ***send request, confirm request,* and *dispatch order*.**
• These events are responsible for state changes of order object.
•During the life cycle of an object (here order object) it goes through the different states and there may be some abnormal exists also.
•This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete it is considered as the complete transaction.
•The initial and final state of an object is also shown in fig.


Statechart diagram of an order management system
Initial state of the object
Intermediate state
Transition
Initialization
idle
Send order request
Normal exit
Select normal or special order
Action
Abnormal exit
Final state (Failure)
Confirm order (Event)
Initial state
Order confirmation
Final state
Complete transaction
Dispatch order


start state
Execution Started
production ends
end state
Customer order (order entry)
Order execution (Product manufactured)
Product delivered (notify customer)
customer cancels order action
transition
Order cancelled
Order cancelled

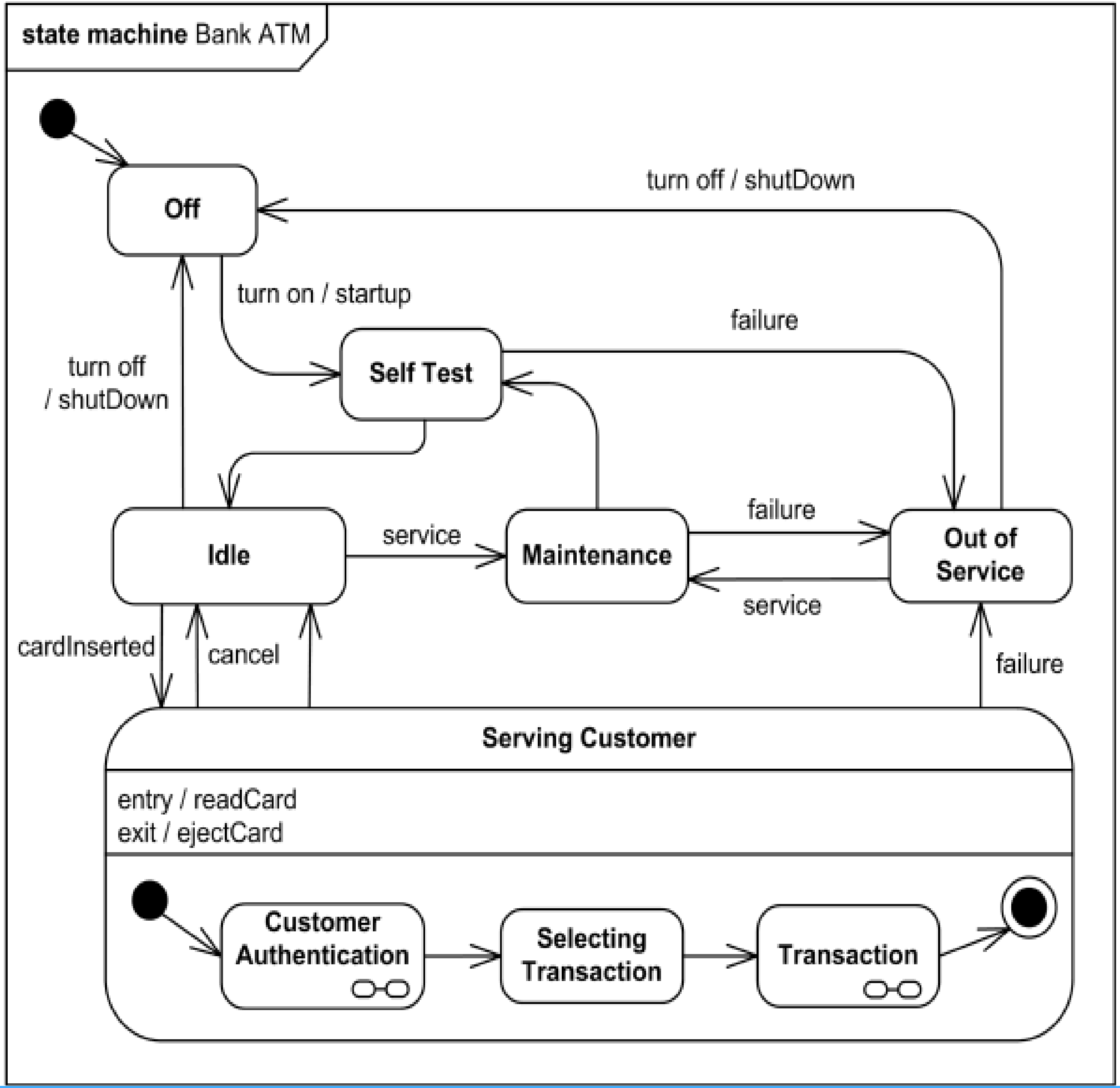
Statechart Diagram Example



Berr

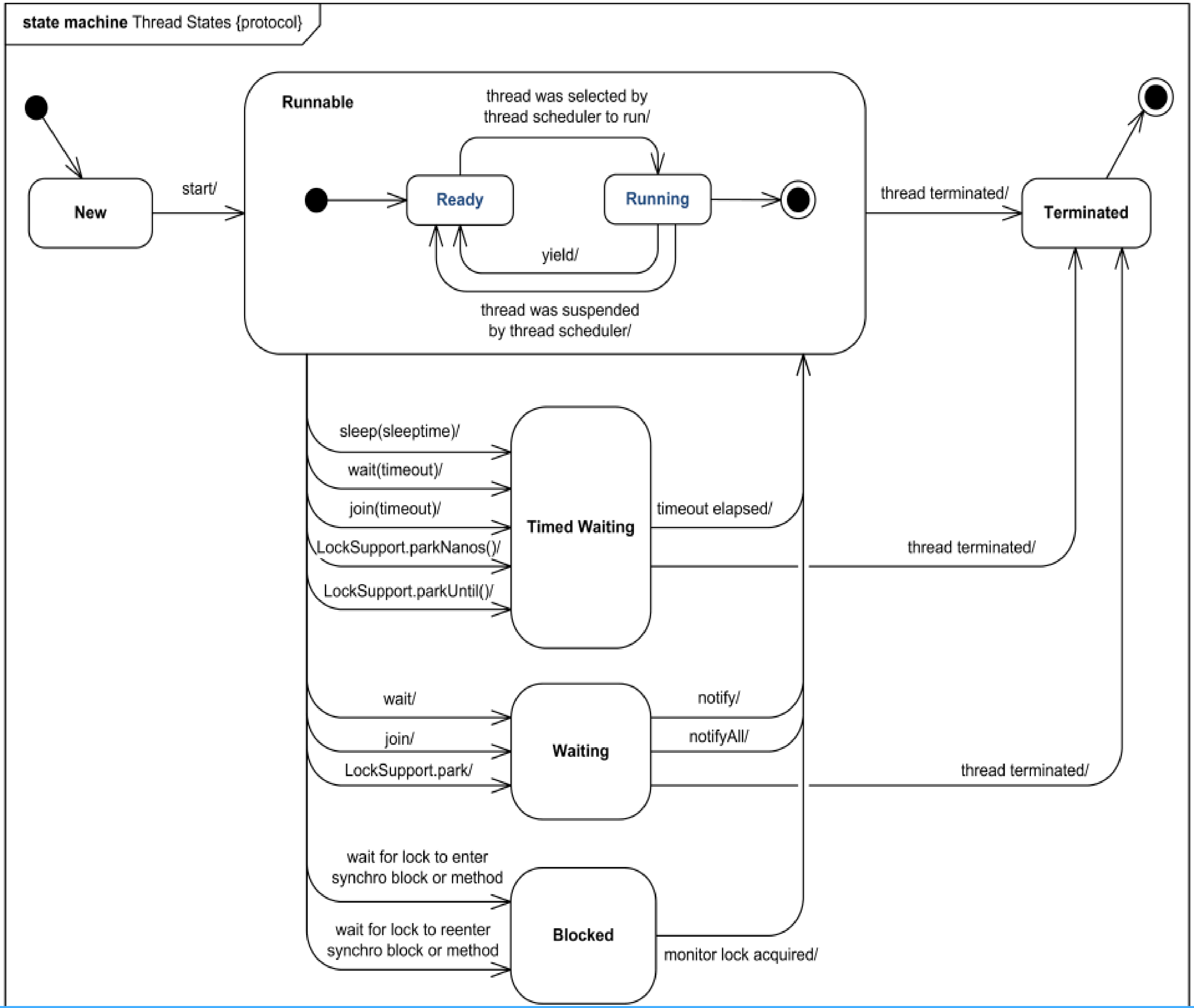
State diagram example

ATM software states at a bank

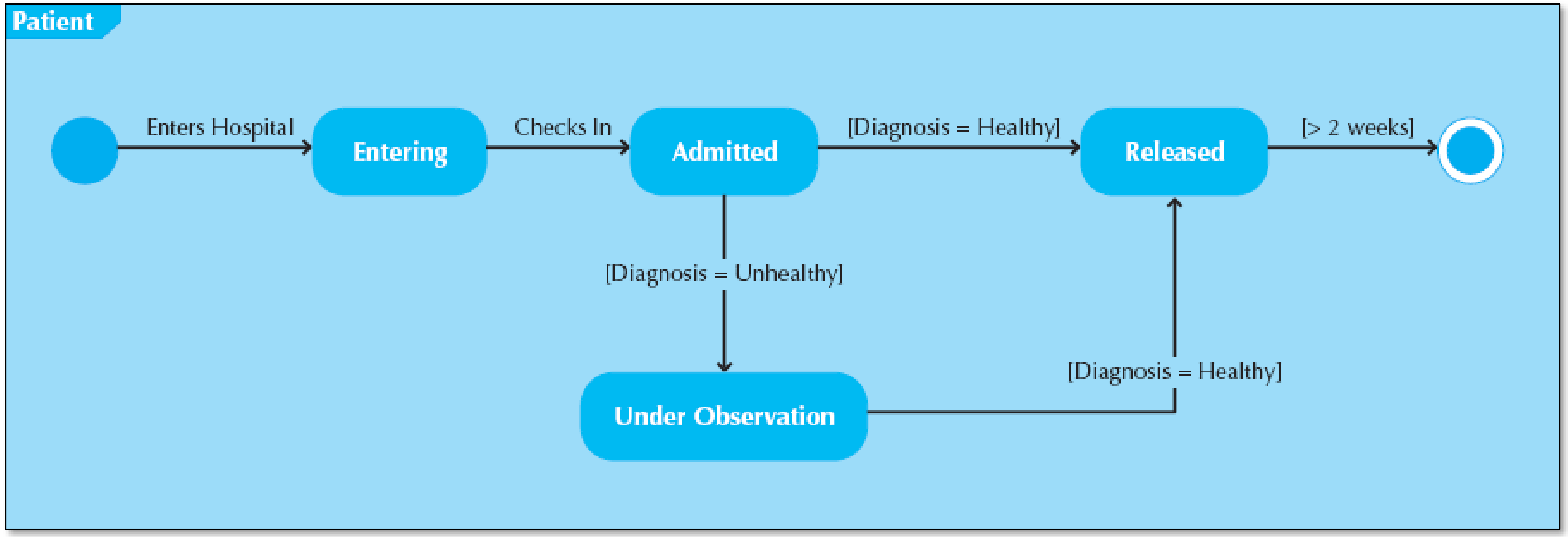


State diagram example

Java thread states



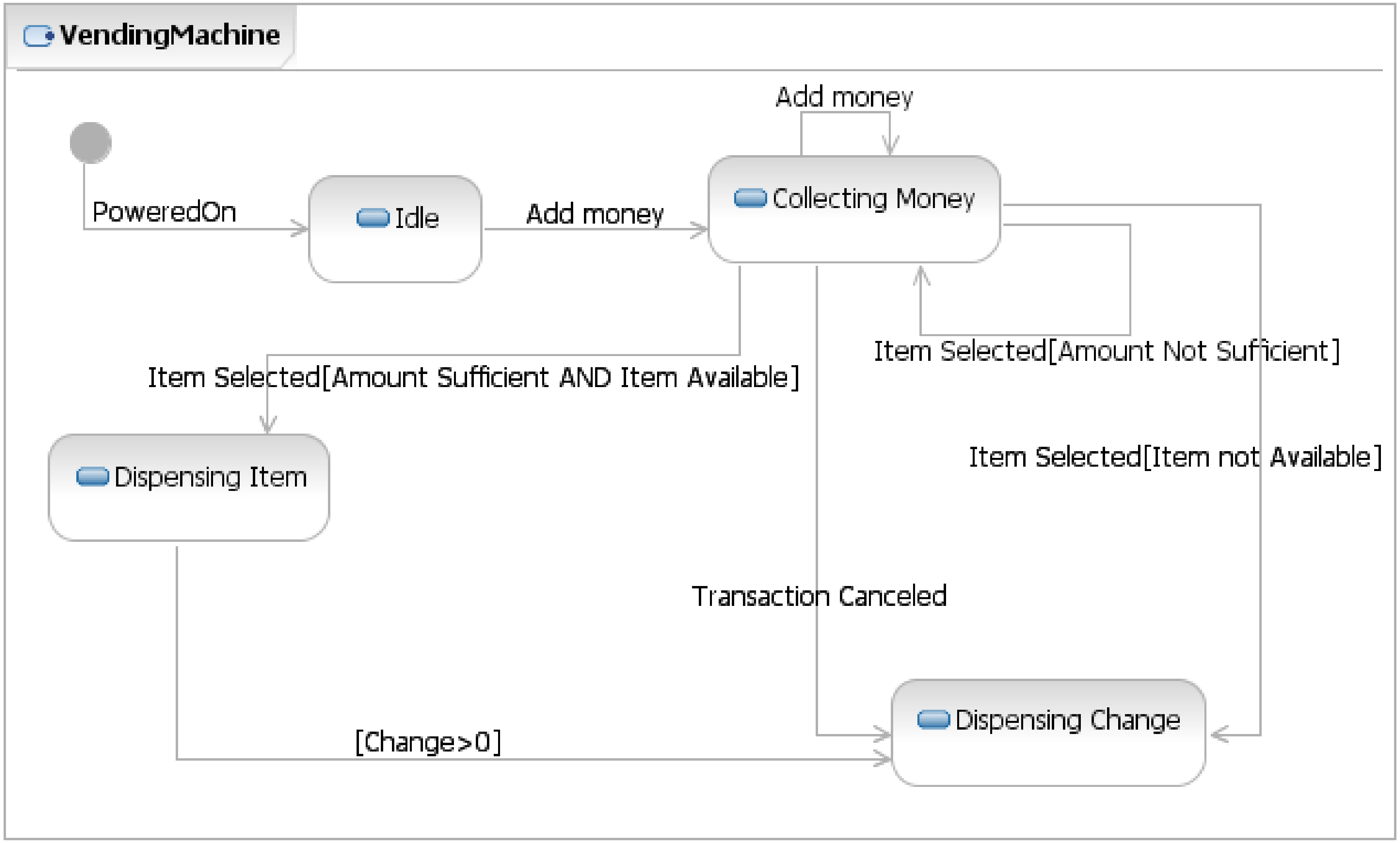
Sample State Diagram for Patient Class



Parts of a State Chart

- Pseudo state – starting point of a statechart
- State – condition that occurs during an object’s life when it satisfies some criteria, performs some action, or waits for an event
- Transition – movement of object from one state to another state
- Message event – trigger for the transition

Vending Machine Example



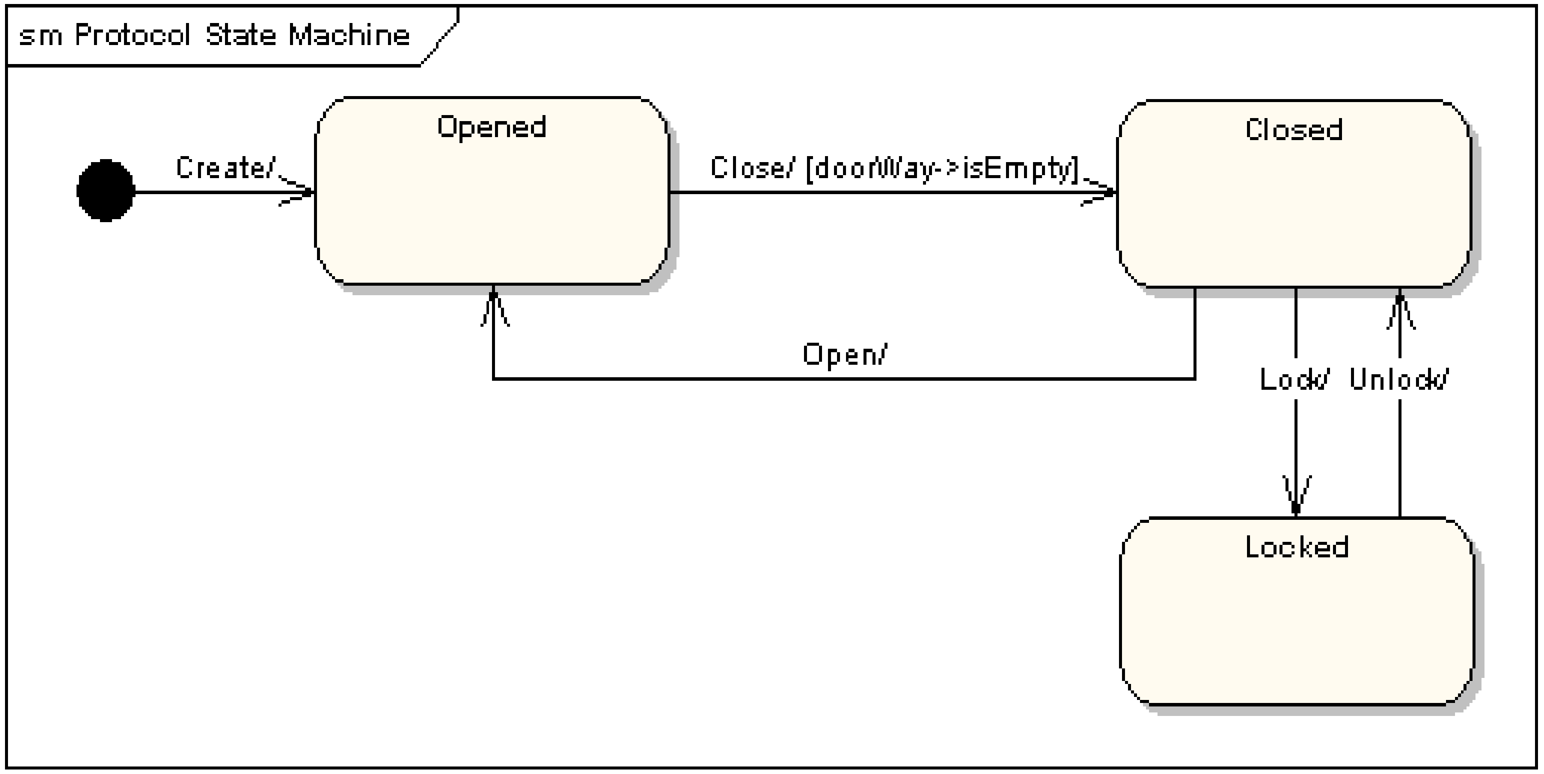
State Machine

•A state machine is a behavior which specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those event.

•A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

•As an example, the following state machine diagram shows the states that a door goes through during its lifetime.

State Machine Diagrams



The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock.

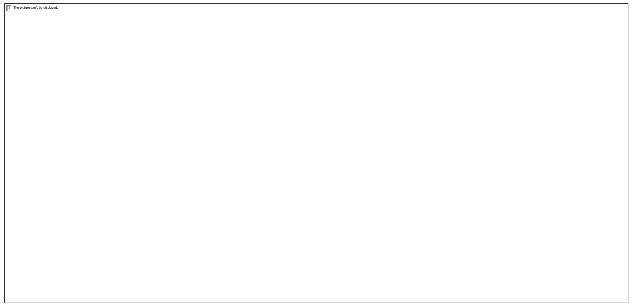
•Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it.

•Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled.

•The syntax and conventions used in state machine diagrams will be discussed in full in the following sections.

State Machine Diagrams

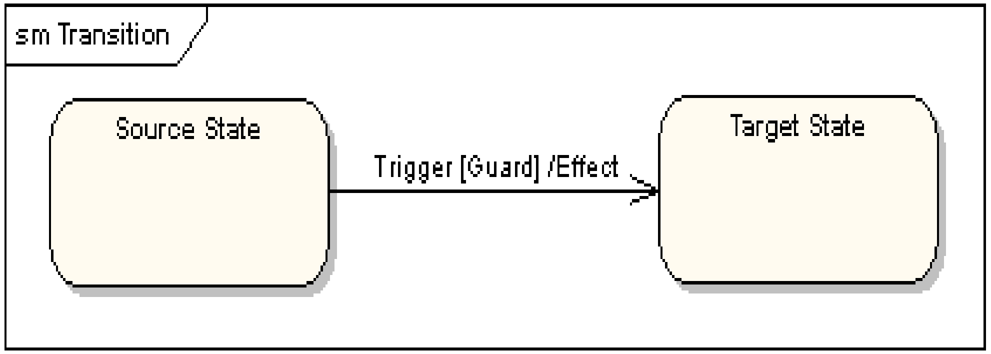
•***States*** - A state is denoted by a round-cornered rectangle with the name of the state written inside it.

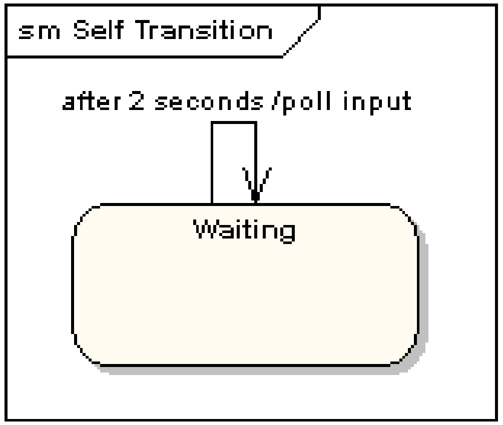


•***Initial and Final States*** **-** The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name.

***Transitions* -** Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.

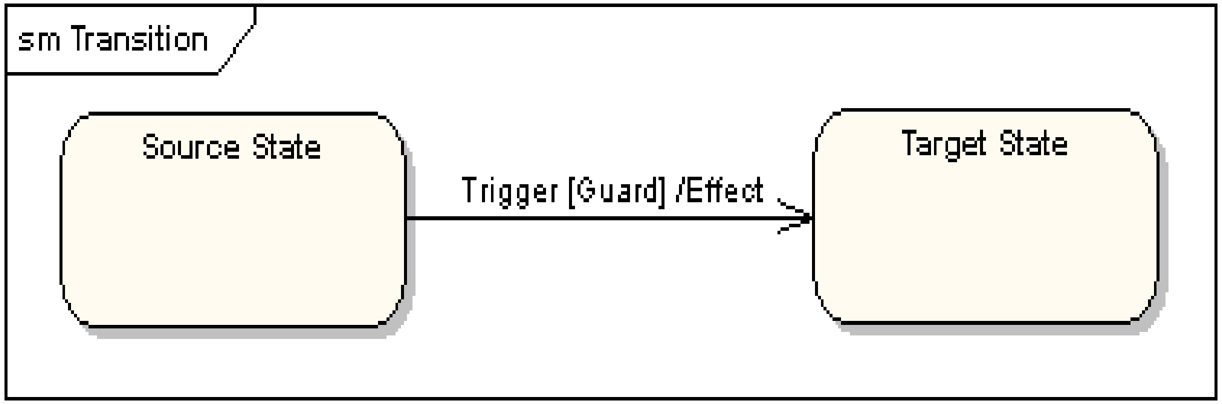
•***Self-Transitions* -** A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.





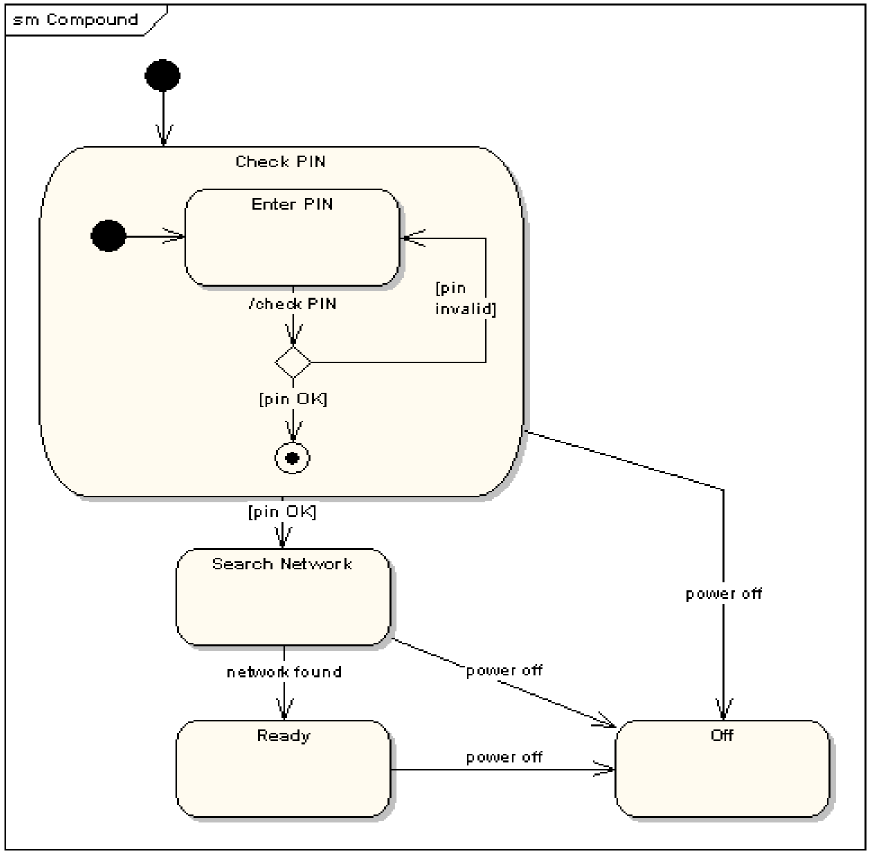
•"***Trigger***" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "***Guard***" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

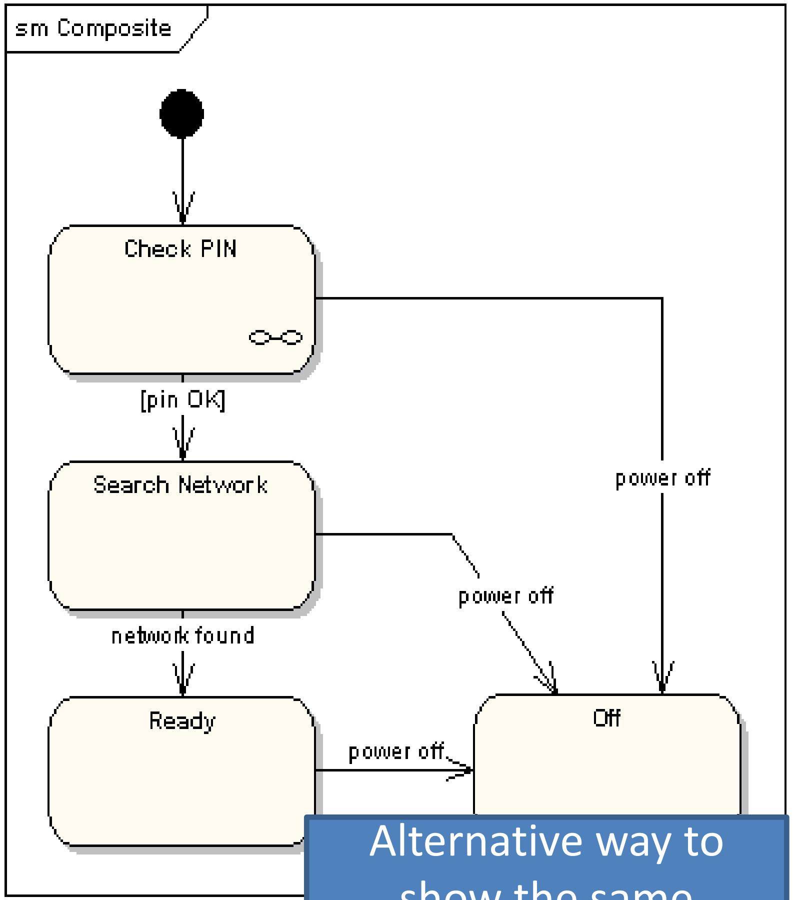
- ***State Actions*** **-** In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



- It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

•*Compound States* - A state machine diagram may include sub-machine diagrams, as in the example below.

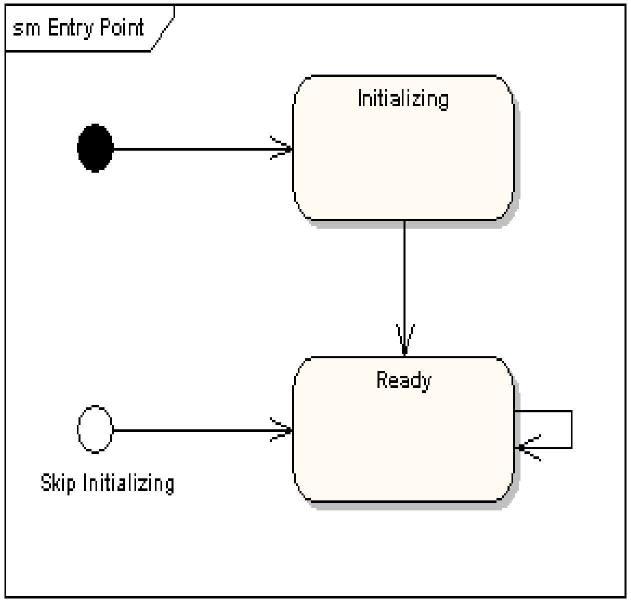


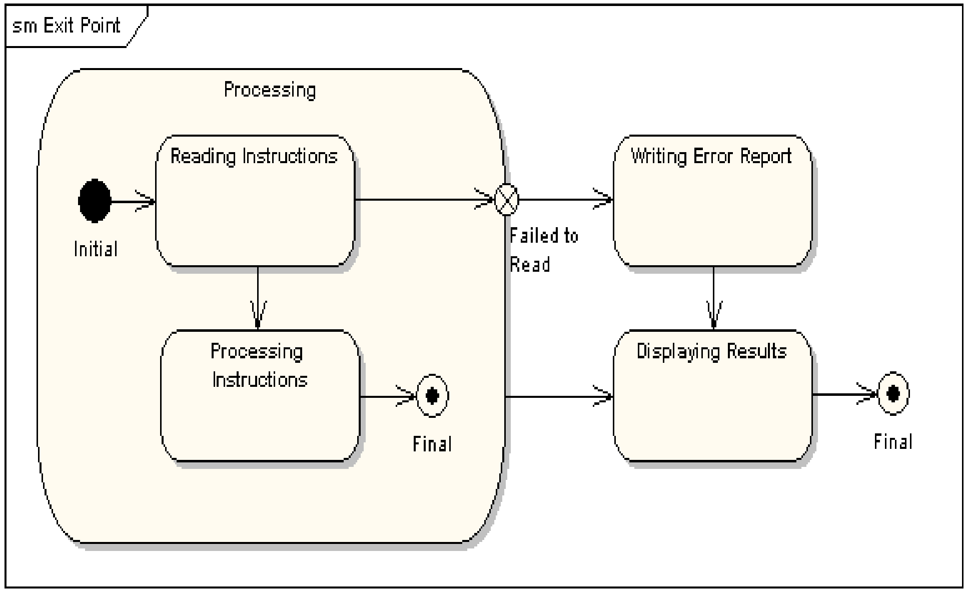


Alternative way to show the same information

•The ∞ symbol indicates that details of the Check PIN sub-machine are shown in a separate diagram.

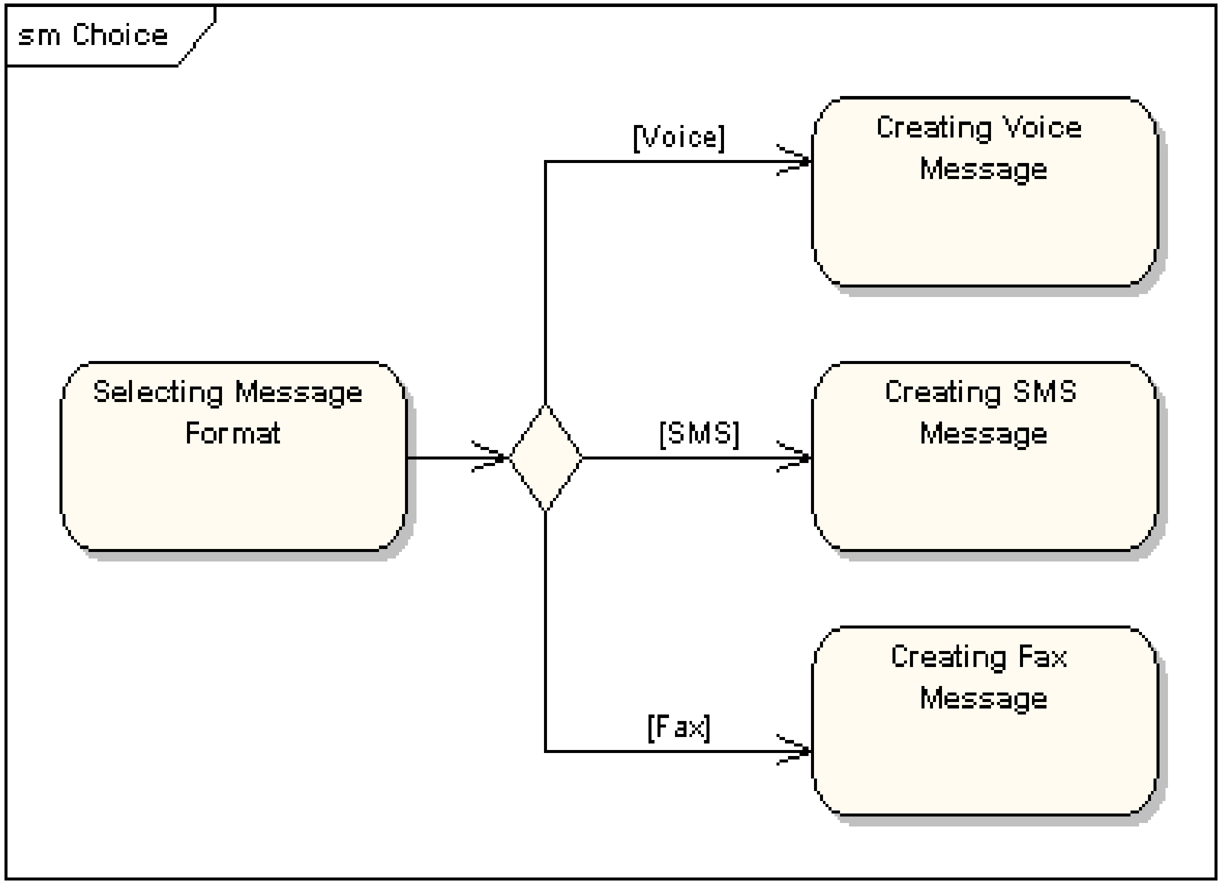
Entry Point - Sometimes you won't want to enter a sub-machine at the normal initial state. For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.



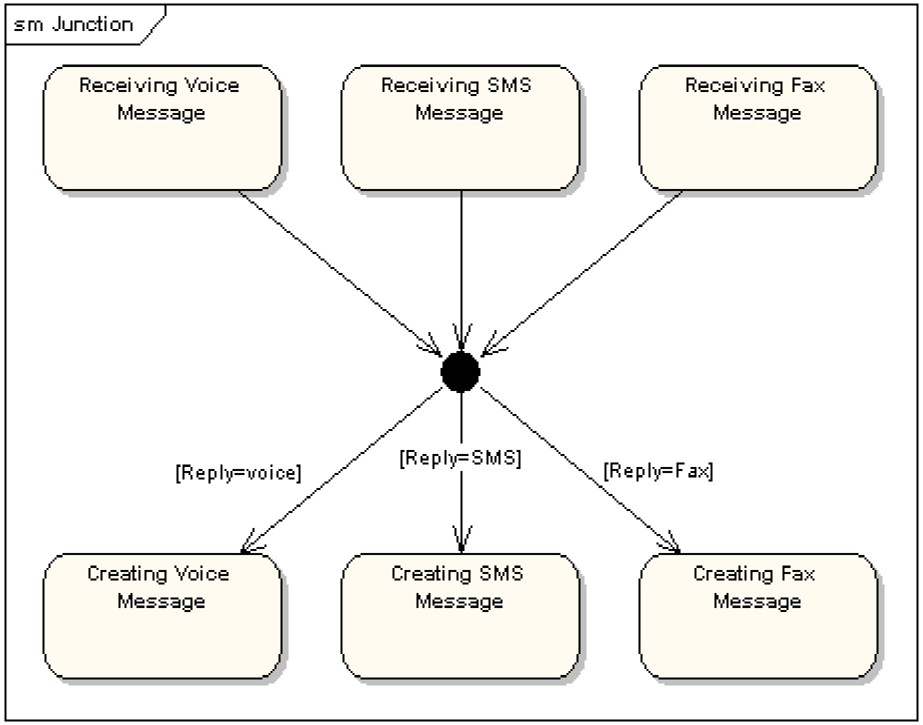


•***Exit Point*** - In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

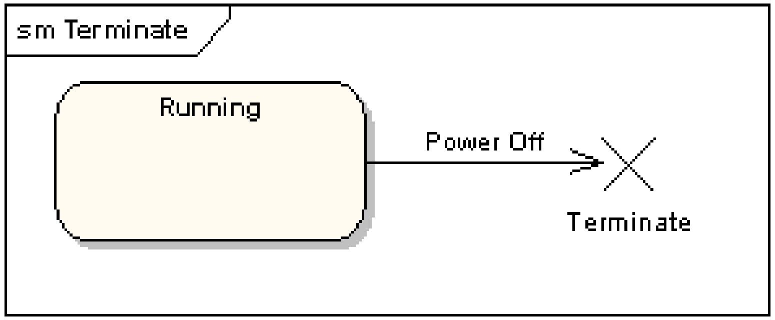
•***Choice Pseudo-State*** - A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



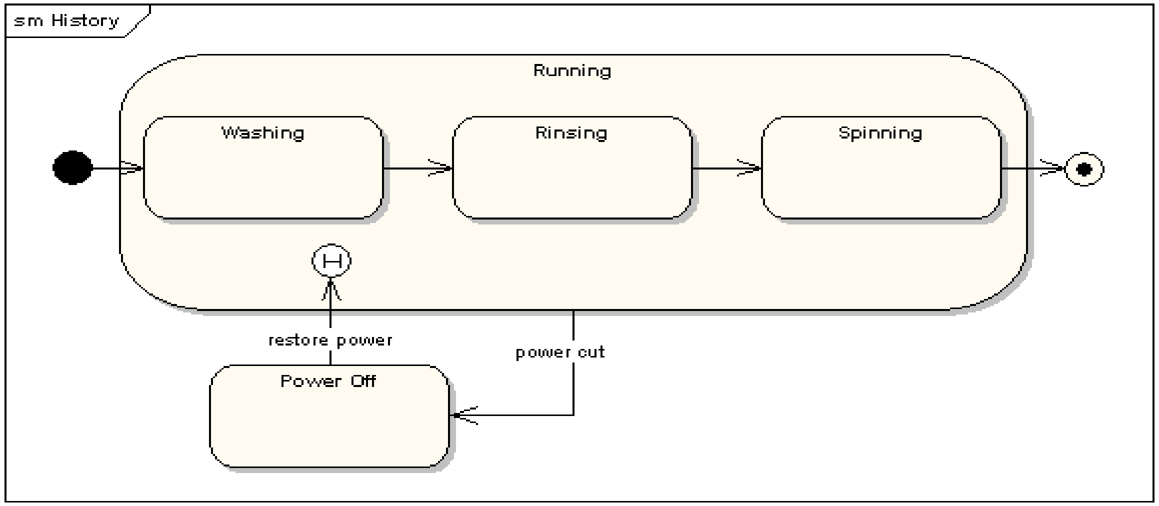
•***Junction Pseudo-State*** - Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free. A junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.



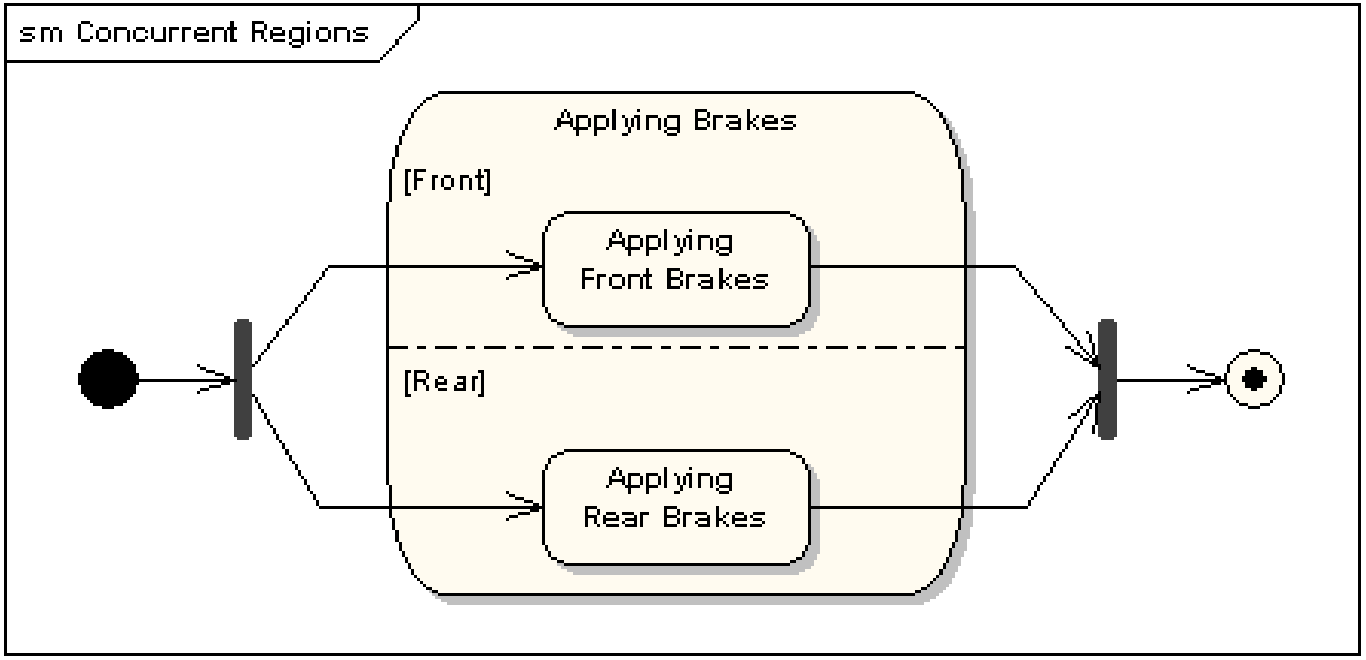
•***Terminate Pseudo-State*** - Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.



•***History States*** - A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



•***Concurrent Regions*** - A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

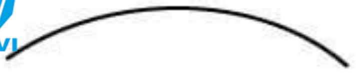
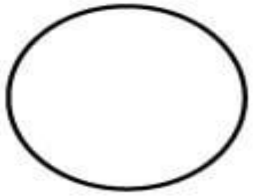
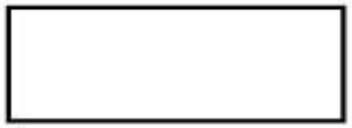
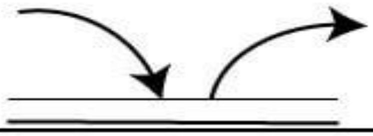


Data Flow Diagrams(DFDs)

- A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It can be manual, automated, or a combination of both.

- It shows how data enters and leaves the system, what changes the information, and where data is stored.

- The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

Symbol	Name	Function
	Data flow	Used to Connect Processes to each , other , to sources or Sinks; te arrow head indicates direction of data flow.
	Process	Perfroms Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

Symbols for Data Flow Diagrams

Circle: A circle (bubble) shows a process that transforms data inputs into data outputs.
Data Flow: A curved line shows the flow of data into or out of a process or data store.
Data Store: A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have an element or group of elements.
Source or Sink: Source or Sink is an external entity and acts as a source of system inputs or sink of system outputs.

➢ Levels in Data Flow Diagrams (DFD)

The DFD may be used to perform a system or software at any level of abstraction. Infact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

➢ 0-level DFDM

It is also known as fundamental system model, or context diagram represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows. Then the system is decomposed and described as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs.

This process may be repeated at as many levels as necessary until the program at hand is well understood. It is essential to preserve the number of inputs and outputs between levels, this concept is called leveling by DeMacro. Thus, if bubble "A" has two inputs x_1 and x_2 and one output y, then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in fig:

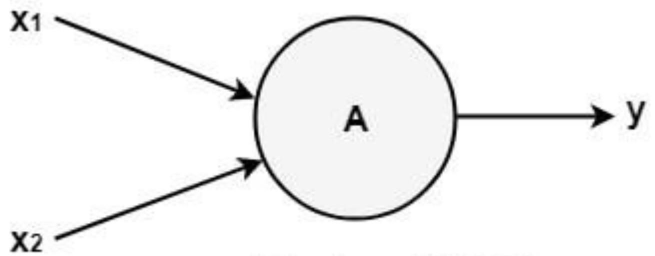


Fig: Level-0 DFD.

The Level-0 DFD, also called context diagram. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flow may also be needed to be decomposed.

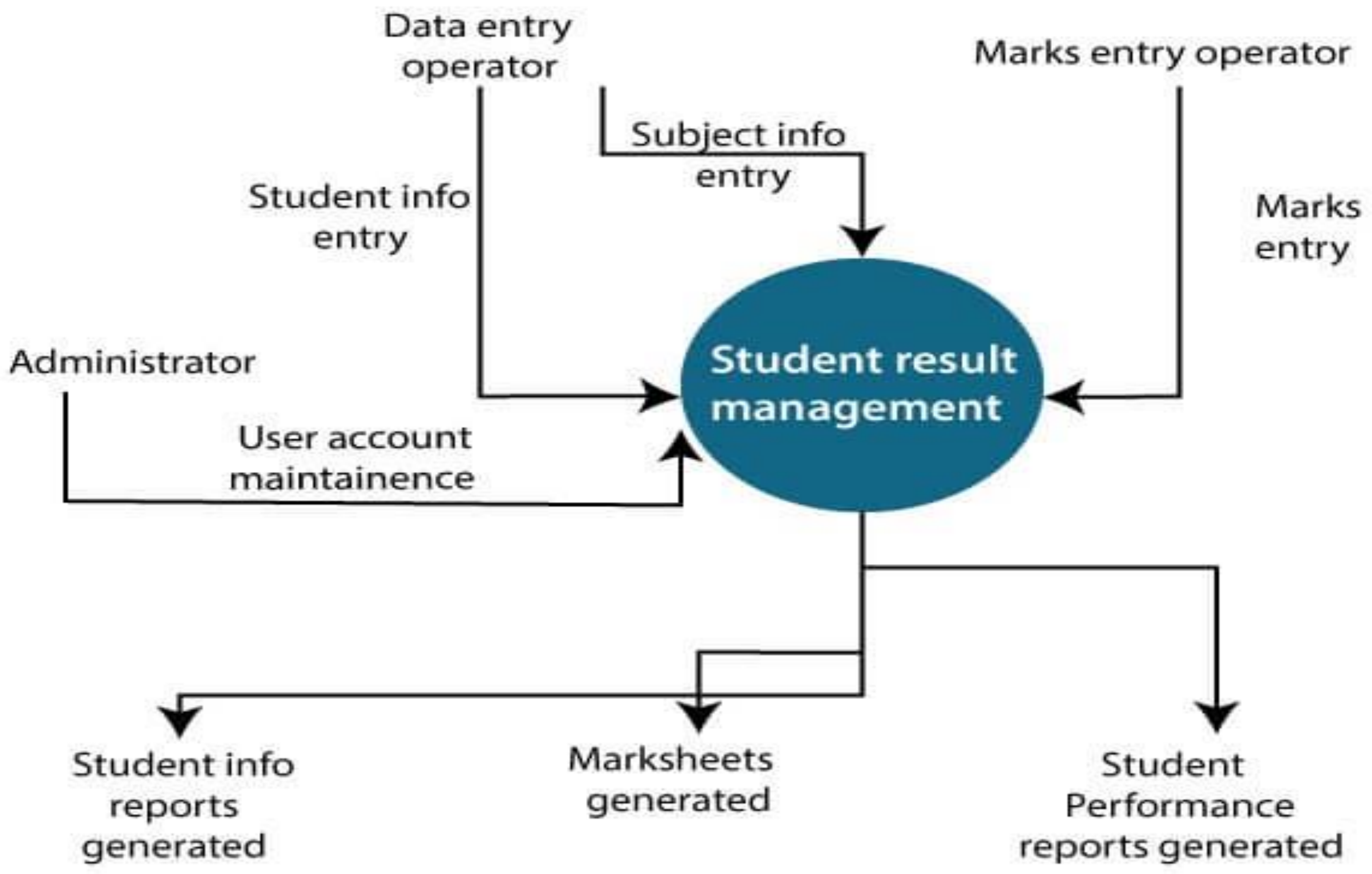


Fig: Level-0 DFD of result management system

1-level DFD

In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, the main objectives of the system and breakdown the high-level process of 0-level DFD into subprocesses are highlighted.

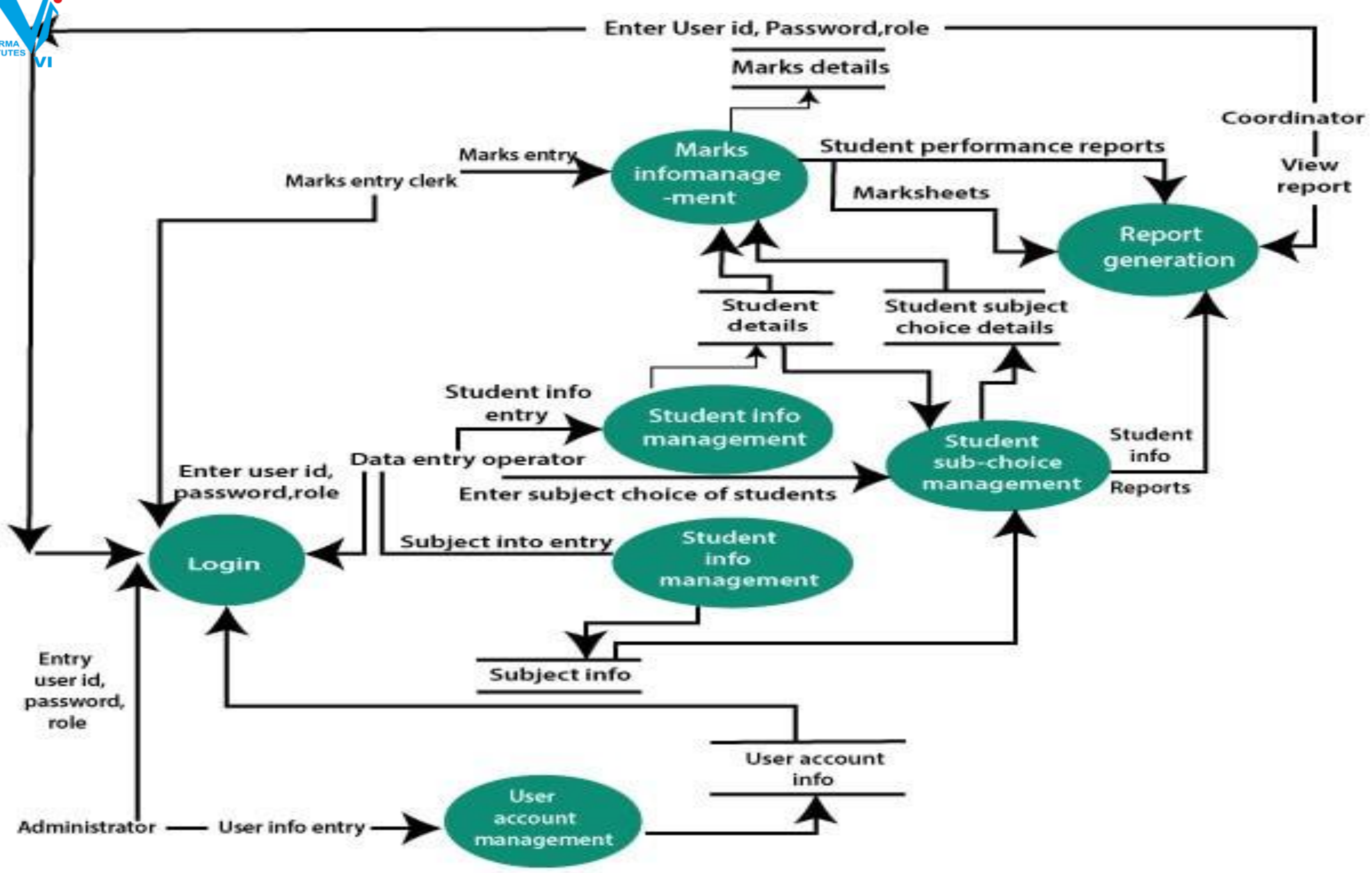
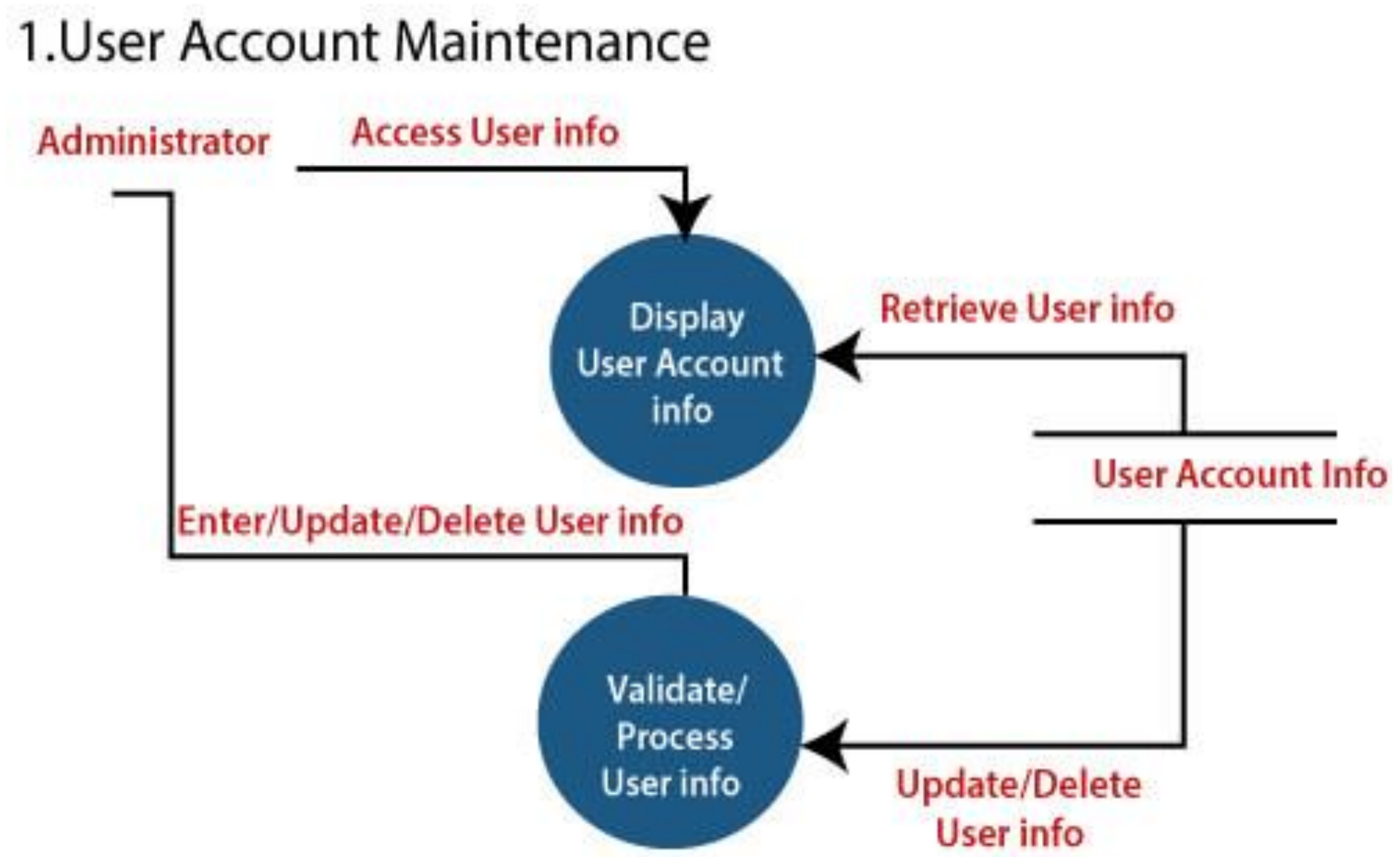


Fig: Level-1 DFD of result management system

2-Level DFD

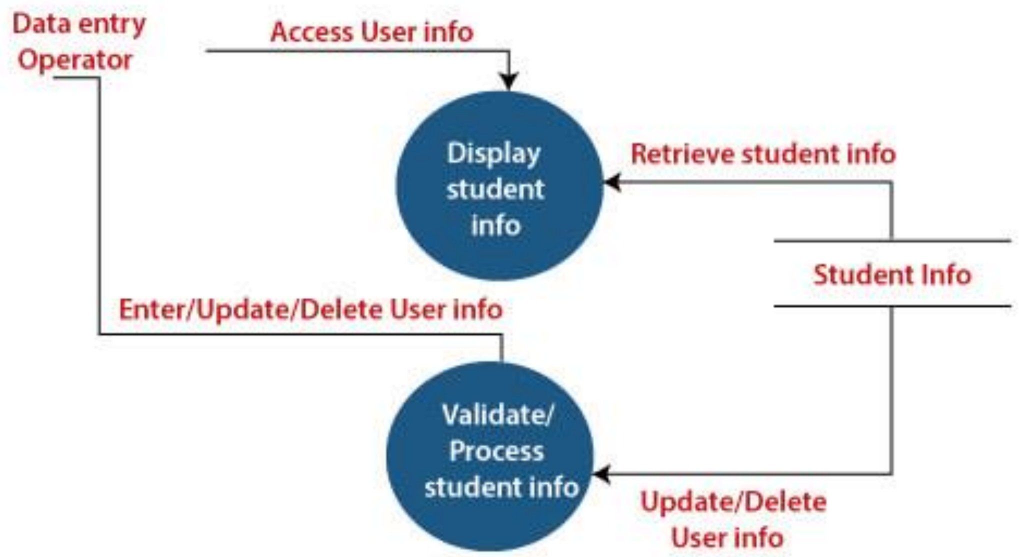
2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.


1.User Account Maintenance
Administrator
Access User info
Display User Account info
Retrieve User info
User Account Info
Enter/Update/Delete User info
Validate/ Process User info
Update/Delete User info

VISHWAKARMA INSTITUTES VI

The picture can't be displayed.

3. Student Information Management



Control Flow Diagrams

- A control-flow diagram can consist of a subdivision to show sequential steps, with if-then-else conditions, repetition, and/or case conditions.
- There are several types of control-flow diagrams, for example:

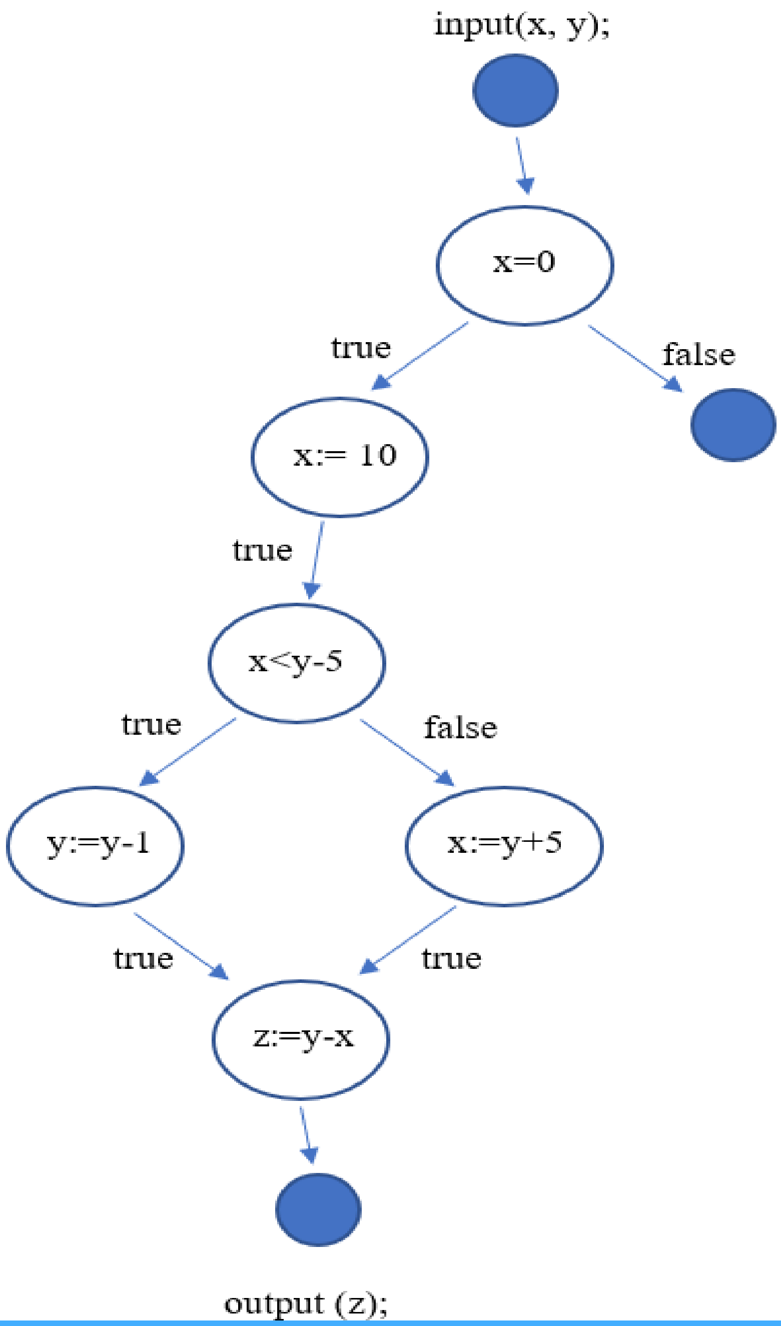
✓ Change-control-flow diagram, used in project management

✓ Configuration-decision control-flow diagram, used in configuration management

✓ Process-control-flow diagram, used in process management

✓ Quality-control-flow diagram, used in quality control.

- In software and systems development, control-flow diagrams can be used in control-flow analysis, data-flow analysis, algorithm analysis, and simulation.



input(x, y);
x=0
true
false
x:= 10
true
x<y-5
true
false
y:=y-1
x:=y+5
true
true
z:=y-x
output (z);

Software Requirement Specifications (SRS)

- A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

- A complete specification of what the proposed system should do.

Purpose of SRS document?

- SRS establishes basis of agreement between the user and the supplier.
  - Users needs to be satisfied, but user may not understand software
  - Developers will develop the system, but may not know about problem domain

- SRS is
  - the medium to bridge the communications gap, and
  - specifies user needs in a manner both can understand

Need for SRS…

- Helps user understand his needs.
  - users do not always know their needs
  - must analyze and understand the potential
  - The requirement process helps clarify needs

- SRS provides a reference for validation of the final product
  - Clear understanding about what is expected.
  - Validation - “ SW satisfies the SRS “

Need for SRS…

- High quality SRS essential for high Quality SW
  - Requirement errors get manifested in final sw
  - To satisfy the quality objective, must begin with high quality SRS

  - Requirements defects cause later problems

Need for SRS…

- Good SRS reduces the development cost
  - SRS errors are expensive to fix later
  - Req. changes can cost a lot (up to 40%)
  - Good SRS can minimize changes and errors
  - Substantial savings; extra effort spent during req. saves multiple times that effort

Systems Requirements Specification

Table of Contents

Systems Requirements Specification

I. Introduction

   A Purpose

   B Scope

   C Definition, Acronyms, or Abbreviations

   D References

   E Overview

Systems Requirements Specification

II. General Description

A Product Perspective

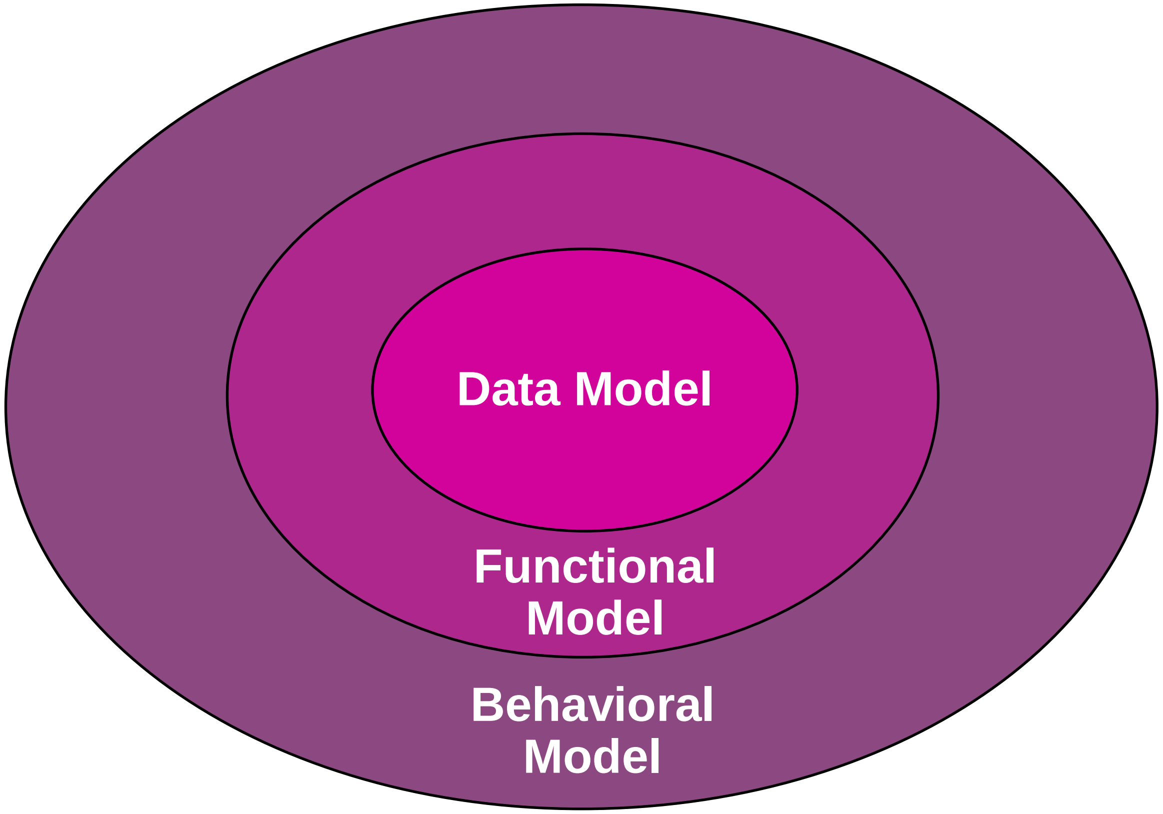
B Product Functions

C User Characteristics

D General Constraints

E Assumptions

Systems Requirements Specification



The SRS is composed of the outer layer of the behavioral model, the functional model, then the data model.

Systems Requirements Specification

Correct	Complete
Precise	Organized
Unambiguous	Verifiable
Consistent	Understandable
Modifiable	Traceable
Design Independent	Annotated
Concise	

Systems Requirements Specification

Correct -

specifies every true requirement known at that time and no incorrect specifications - no wrong data

Precise -

remember this must eventually turn to executable code, fuzzy words in requirements are not acceptable

Unambiguous

each requirement has only one interpretation - English interpretation

Complete -

everything included behavior (methods, use cases, systems, subsystems, business rules) and data (objects, attributes)

Systems Requirements Specification

Verifiable

is the software built what was specified in the SRS

Consistent

conflicting terms, characteristics

Understandable

question: are formal specifications understandable, are informal specifications understandable

Systems Requirements Specification

Modifiable

changing requirements easily modified when specifying, designing, coding, implementing

Traceable

can I locate the SRS origin of software components.

Design Independent

SRS should not specify a particular design

Systems Requirements Specification

- Section One
  - Overview document for executives describing the system from a management perspective
- Section Two
  - General Description describing the system from a user and system perspective in general terms.
- Section Three
  - Detailed document for users and developers describing the system in detailed terms.

Systems Requirements Specification

I. Introduction

A Purpose

B Scope

C Definition, Acronyms, or Abbreviations

D References

E Overview

Systems Requirements Specification

I. Introduction

A Purpose

The purpose of this Software Requirements Specification document

Intended audience of this document

Systems Requirements Specification

I. Introduction

A Purpose

The purpose of the Software Requirements Specification document is to clearly define the system under development, namely the Video Rental System (VRS). The intended audience of this document includes the owner of the video store, the clerks of the video store, and the end users of the VRS. Other intended audience includes the development team such as the requirements team, requirements analyst, design team, and other members of the developing organization.

Systems Requirements Specification

I. Introduction

B. Scope

Origin of need
High-level description of the system functionality
Goals of proposed system

Systems Requirements Specification

I. Introduction

B. Scope

Origin of the need

• who and what triggered the request for this software development activity

• gives developers an understanding of the goals for the proposed system

Systems Requirements Specification

I. Introduction

B. Scope

High-level functionality

- defined for the system
- usually in list separated by commas

Systems Requirements Specification

I. Introduction

B. Scope

Goals are general purposes of a system. They are fuzzy and non measurable.

A typical goal would be things like

- Increase customer satisfaction
- Make xyz easier for the customer
- Improve customer relationships

Systems Requirements Specification

I. Introduction

B. Scope

The owner of a local video store wanted to create a new business plan where everything about renting a video (except the picking up and returning of videos) was done online. Therefore, the new VRS will allow the following functionality online: to search for videos, to become members, to rent videos, to modify membership information, and to pay overdue fees. The store personnel may use the VRS to process the rented or returned videos, to add or remove videos to/from his store's video inventory and to update video information. The VRS is intended to increase the owner's profit margin by increasing video sales with this unique business approach and by allowing him to reduce the staffing needed in his stores.

Systems Requirements Specification

I. **Introduction**

C. Definitions, Acronyms..

As you begin to define a system, you will encounter words which need definition and general usage acronyms. These should be documented for new personnel and for clarity of all concerned parties.

Systems Requirements Specification

I. Introduction C Definitions, Acronyms..

FSU – Florida State University

CS - Computer Science

MSES - Masters in Software Engineering Science

DOE - Department of Education

….

Systems Requirements Specification

I. Introduction

D. References

Many references may be used to define existing systems, procedures (both new and old), documents and their requirements, or previous system activities. These references are listed here for others.

If any of these references are provided in the appendices, it should be noted here.

Systems Requirements Specification

I. Introduction D References

Clerk - Personnel staff who is working in a video store

Customer - Anyone who interacts with the VRS with becoming a member

Functional requirement - A service provided by the software system

Member - Anyone who registers with the VRS to acquire membership in the video store

Section I of SRS

I.A Purpose	Paragraph form
I.B Scope of the System Specified	Paragraph form
I.C Definitions, Acronyms, and Abbreviations	Table form or bulleted list
I.D References to Supporting Documents	Bulleted list
I.E Overview of rest of SRS	Paragraph form

I. Introduction

E. Overview

This section defines the organization of the entire document. It will lay the framework for reading the document.

I. Introduction E Overview

Section 2 of the SRS describes the product in more detail. Section 3 provides a complete list of the functional requirements of the intended system. Section 4 provides the non-functional requirements. Section 5 shows the class diagram, and Section 6 the use case diagram. The appendices appear next.

Systems Requirements Specification

II. General Description

- A Product Perspective
- B Product Functions
- C User Characteristics
- D General Constraints
- E Assumptions

II General Description

A Product Perspective

This defines the relationship this product has in the entire spectrum of products.

It defines who will be responsible for the product and what business purpose it serves.

It also defines what interfaces it may have to other systems.

II General Description

A Product Perspective

The VRS is a web-based system. The system interfaces with two other systems, the owner's email system, the video distributor's video system, and the browsers used by VRS customers. The system provides a secure environment for all financial transactions and for the storing and retrieving of confidential member information.

II. General Description B Product Functions

This section lists the major functions of the system.

It provides a summary of all the functions of the software. The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.

This section should be consistent with the functional requirements defined in Section III.

II. General Description – B Product Functions

The VRS allows customers to search the video inventory provided by this video store. To rent videos through the VRS, one must register as a member using the VRS. Upon becoming a member and logging into the VRS, the VRS provides the functionality for renting videos, modifying membership information, and paying overdue fines.

The clerks of the video store use VRS to process the return of rented videos. The owner of the video store uses VRS to add new videos into the system, remove videos from the system, and modify video information.

The VRS sends emails to members concerning video rentals. One day before a rented video is due to be returned, VRS emails the member a reminder of the due date for the video(s). For any overdue videos, VRS emails the member every 3rd day with overdue notices. At the 60-day limit for outstanding videos, VRS debits the member's credit card with the appropriate charge and notifies the member of this charge.

II. General Description – C User Characteristics

List the users involved with the proposed system including the general characteristics of eventual users (for example, educational background, amount of product training).

List the responsibility of each type of user involved, if needed.

II. General Description – C User Characteristics

The three main groups of VRS users are customers, members, and store personnel. A customer is anyone who is not a member. The customer can only search through the video inventory. The amount of product training needed for a customer is none since the level of technical expertise and educational background is unknown. The only skill needed by a customer is the ability to browse a website.

Member is someone who has registered with VRS. A member can rent videos and pay fees online. As with a customer, these activities require no product training since the level of technical expertise and educational background of a member is unknown. The only skill needed by a member is the ability to browse a website.

The store personnel are divided into two groups: the clerk-level personnel and owner-level personnel. Their educational level is unknown and both group needs little to no training.

II. General Description – D General Constraints

In this section, the constraints of the system are listed. They include hardware, network, system software, and software constraints. It also includes user constraints, processing constraints, timing constraints, and control limits.

II. General Description – D General Constraints

This system provides web access for all customer and member functions. The user interface will be spontaneous enough so that no training is required by customers, members, or store personnel. All online financial transactions and the storage of confidential member information will be done in a secure environment. Persistent storage for membership, rental, and video inventory information will be maintained.

II General Description – D Assumptions and Dependencies

This includes assumptions made at the beginning of the development effort as well as those made during the development.

List and describe each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but any changes to them can affect the requirements in the SRS. For example, an assumption might be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change.

Section II of SRS

II.A Product Perspective	Paragraph form
II.B Product Functions	Paragraph form
II.C User Characteristics	Paragraph form
II.D General Constraints	Paragraph form
II.E Assumptions and Dependencies	Paragraph form

III Functional Requirements

Functional requirements are those business functions which are included in this software under development. It describes the features of the product and the needed behavior.

The functional requirements are going to be written in narrative form identified with numbers. Each requirement is something that the system SHALL do. Thus, it has a common name of a *shall* list. You may provide a brief design rationale for any requirement which you feel requires explanation for how and/or why the requirement was derived.

IV Non Functional Requirements

Non functional requirements are properties that the system must have such as performance, reusability, usability, user friendliness, etc.

The same format as the functional requirements is to be used for the non-functional requirements. You may provide a brief design rationale for any requirement which you feel requires explanation for how and/or why the requirement was derived.

V System Architecture

This section presents a high-level overview of the anticipated system architecture using a class diagram. It shows the fundamental objects/classes that must be modeled with the system to satisfy its requirements. Each class on the diagram must include the attributes related to the class. All the relationships between classes and their multiplicity must be shown on the class diagram. The classes specified in this document <u>only</u> are those directly derived from the application domain.

VI System Models

This section presents the use case diagram for the system under development. The use case diagram should be a complete version containing all the use cases needed to describe the functionality to be developed.

VII Appendixes

Appendix A. Data dictionary

Appendix B. Raw use case point analysis

Appendix C. Screens and reports with navigation matrix.

Appendix D. Scenario analysis tables

Appendix E. Screens/reports list

Appendix F and following. Other items needed

SRS Table of Contents

3. Specific Requirements

3.1 Functional Requirements

- 3.1.1 Func Req 1
  - 3.1.1.1 Introduction
  - 3.1.1.2 Inputs
  - 3.1.1.3 Processing
  - 3.1.1.4 Outputs
- 3.1.2 Func Req 2
  - **…**

3.2 External Interface Requirements

- 3.2.1 User Interface
- 3.2.2 Hardware Interfaces
- 3.2.3 Software Interfaces
- 3.2.4 Communication Interfaces

3.3 Performance Requirements

3.4 Design Constraints

- 3.4.1 Standards Compliance
- 3.4.2 Hardware Limitations

3.5 Attributes

- 3.5.1 Security
- 3.5.2 Maintainability

3.6 Other Requirements

- 3.6.1 Database

Summary

- Having a good quality SRS is essential.
- The req. phase has 3 major sub phases
  – analysis , specification and validation
- Analysis
  – for problem understanding and modeling
- Key properties of an SRS: correctness, completeness, consistency, traceability, unambiguousness.
- Specification
  – must contain functionality, performance , interfaces and design constraints
  – Mostly natural languages used
- Validation - through reviews

References

- SOFTWARE REQUIREMENT ELICITATION TECHNIQUES ...epgp.inflibnet.ac.in › epgpdata › uploads › epgp_content
- https://www.tutorialspoint.com/software_engineering/software_requirements.htm
- https://web.cs.dal.ca › ~hawkey › May17_SRS
- https://www.javatpoint.com/software-engineering-data-flow-diagrams
- SRS-www.cs.fsu.edu › ~baker › swe1 › restricted › notes › ppt › SRS
-

Behavioral Modeling: State Diagrams

CIS 4800
Kannan Mohan
Department of CIS
Zicklin School of Business, Baruch College

VISHWAKARMA INSTITUTES VI

Thank You

Object-Oriented Software Engineering
Conquering Complex and Changing Systems

Chapter 4, Requirements Elicitation