Software Development Tools

2020-2021

الدراسات الأولية / المرحلة الرابعة / الفصل الثاني

أستاذ المادة

م.د.فاتن عبدعلي داود

Introduction to Software Development Tools (SDT)

Course Description:

- This course covers formal methods used in the development of tools. It studies the important development tool of object-oriented modeling, such as the **Unified Modeling Language (UML)**.
- This course presents an integrated set of techniques for software analysis and design based on object- oriented concepts and the UML notation.
- This course also covers **Software Development Lifecycle (SDLC)** which including three main stages:

1-Requirements Specification (user and system requirements)
2-Development (design and implementation)
3-Validation (Testing & Debugging)

What is a requirement?

- It may range from a high-level abstract statement of a service that the system should provide or a system constraint to a detailed mathematical functional specification.
- All of the process models for software development include activities aimed at capturing the requirements to understanding what the customer wants and what users expect the system will do.

The requirements may be developing to:
- Replace an existing manual system.
- Enhance or extend an existing software system.
- Develop a new system unrelated to existing systems.

Types of Requirements

User requirements:

1. Statements in natural language (NL) plus diagrams of the services the system provides and its operational constraints.
2. Should describe functional and non-functional requirements so that they are understanding by system users who do not have detailed technical knowledge.
3. User requirements are defined using NL, tables, and diagrams.
4. Written for customers.
5. **Requirements readers** (Client managers, System end-users and System Architects).

System requirements:

1. Structured documents setting out detailed description of the system services.
2. Serve as a basis for designing the system. May be used as part of the system contract.
3. System requirements may also be expressed using system models.
4. Written as a contract between client and contractor.
5. **Requirements readers** (System end-users, System architects and Software developers).

What is Requirements Elicitation?

- The aim of requirements elicitation is to understand the problem clearly.
- **There are three main activities should be used:**
  1: Analyzing the problem
  2: Identifying requirements sources
  3: Eliciting requirements from their sources

1: Analyzing the problem

a - Identifying goals

e.g. Problem : The current information system is costly to maintain.
Goal : Reduce maintenance cost.

b - Identifying constraints

Resources (time, people, budget) are also constraints
Goals and constraints should be measurable.

c- Defining the scope of the problem (the boundary of the system)

- what is external to the problem (i.e. the environment)
- what is internal to the problem (i.e. the core of system)

d- Assessing the risk

(e.g. financial, technical, etc) and its acceptance

e- Estimating an approximate Project Cost

2: Identifying requirements sources

Typical sources of requirements:

- Stakeholders
- The domain

a- the operational environment

How the system will be used at run-time?
e.g. reliability, availability, and performance requirements
Is safety an issue?
e.g. safety requirements

b- the organizational environment

e.g. Who will use the system and for what? (helps identify users and their requirements)

c- the application domain

- Provides knowledge of the general area where the system is applied.
- Can be gathered from various sources such as stakeholders, textbooks, operating manuals and
  in the heads of the people working in that area.

3: Eliciting requirements from their sources

Two main goals :

1- Find out what the stakeholders need
2- Collect information about what it is feasible for the stakeholders to have.

Not all of the Stakeholders requirements will be realistic.
Not all of the information will be useful.
At this stage, we just *collect* it!

Why the eliciting Requirement is a difficult task?

1- Stakeholders may have difficulty describing their tasks and may leave
2- Important information unstated
3- In large systems, no-one knows everything
4- Requirements change
5- Some requirements are implicit

Requirements Elicitation Techniques :

1- **Interviews :** Discussing the system with different stakeholders and building up an understanding of their requirements.
2- **Observation**: Observe how users interact with their systems and each other in the organizational environment.
3- **Facilitated meeting**: encourage work group rather than work individually.
4- **Prototyping:** Shows end-users and system stakeholders what facilities the system can provide.
5- **Scenarios:** Examples of interaction sessions which are concerned with a single type of interaction between an end-user and the system (to perform a task or a function).

What is a scenario?
A scenario is an instance of a use case, that is, a use case describes all possible scenarios for a given piece of functionality.

Why working through a scenario is important?

1- Walking through a scenario with users helps the developer understand what they need from a system.
2- A scenario is useful to elicit and clarify requirements.

Modelling Concepts

What are models?

- A complete description of a system from a particular perspective
- Simplification of reality

Modeling achieves six aims:

1. Provide structure for problem solving
2. Experiment to explore multiple solutions
3. Helps you to **visualize a system** as you want it to be.
4. Permits you to **specify the structure or behavior of a system.**
5. Gives you a **template that guides you in constructing** a system.
6. **Documents the decisions** you have made.

You build models to better understand the system you are developing.

Modeling allows the following business benefits:

- Reduce time-to-market for business problem solutions
- Decrease development costs
- Manage the risk of mistakes

There are two types of Models:

- ***Analysis Model* :** A model of the problem domain (e.g. this is a description of those aspects of the real world system that are relevant to the problem under consideration)

- ***Design Model* :** A model of the solution domain.

***Remember that we are modeling the problem domain, that is, the analysis model using use case diagram and activity diagram of UML* course.**

Unified Modeling Language (UML)

The **UML** is an industry standard for object oriented design notation, supported by the Object Management Group (OMG). **UML** is a complete language that is used to design, visualize, constructs and document systems. It is largely based on the object-oriented paradigm and is an essential tool for developing robust and maintainable software systems.

UML is a language for models

- Technical and graphical specification
- Graphic notation to visualize models
- Not a method or procedure

Why Do We Need UML?

- Standard communication language
- Provides multiple diagrams for capturing different architectural views
- Promotes component reusability

What are the benefits from Using UML Modeling Tool?

1- Visualize in multiple dimensions and levels of detail
2- Use automated layout and visualization tools
3- Generate documentation from modeling environment
4- Analyze traceability through relationships between elements
5- Incremental development and refactoring
6- Teamwork for parallel development of large systems
7- Integration with other development tools

➢ The **UML** is a language for Visualizing, **Specifying**, **Constructing and Documenting** the software-intensive system such as explained as follows:

1- Language for Visualizing

- Communicating conceptual models to others is prone to error unless everyone involved speaks the same language.
- There are things about a software system you can't understand unless you build models.
- An explicit model facilitates communication.

2- Language for Specifying

- The UML builds models that are precise, unambiguous, and complete.

3- Language for Constructing

- UML models can be directly connected to a variety of programming languages.
- Maps to Java, C++, Visual Basic, and so on
- Tables in a RDBMS or persistent store in an OODBMS
- Permits forward engineering
- Permits reverse engineering

4- Language for Documenting

• The UML addresses documentation of system architecture, requirements, tests, project planning, and release management.

Three Ways to Apply UML

1- UML as sketch

- Informal and incomplete diagrams often hand sketched on whiteboards
- created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

2- UML as blueprint

- Relatively detailed design diagrams used either for:
  i. Reverse engineering to visualize and better understanding existing code in UML diagrams
  ii. Code generation (forward engineering).

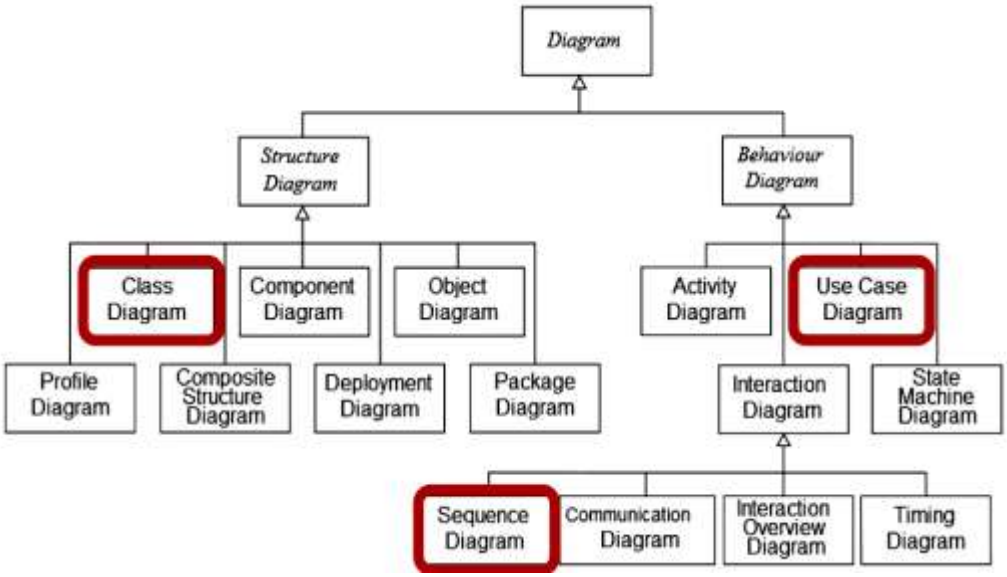
3- UML as programming language

- Complete executable specification of a software system in UML.
- Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language."
- This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams),
- Still under development in terms of theory, tool robustness and usability.

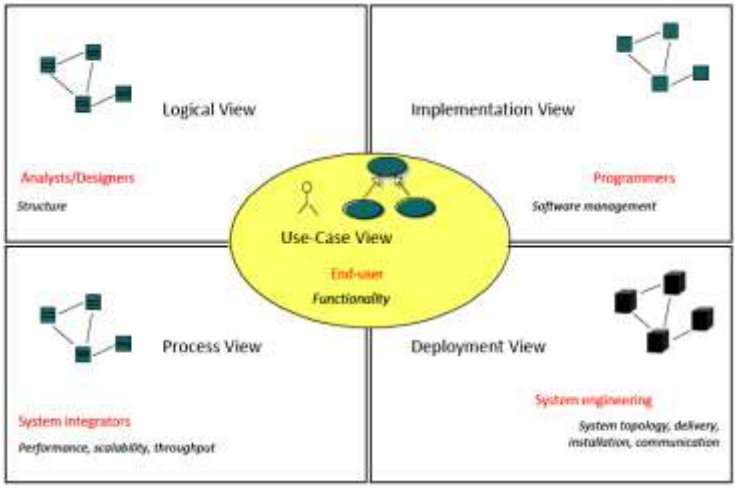
Diagrams are graphically depicting a view of a part of your model. Different diagrams represent different views of the system that you are developing. A model element will appear on one or more diagrams.

There are Two different groups of models:

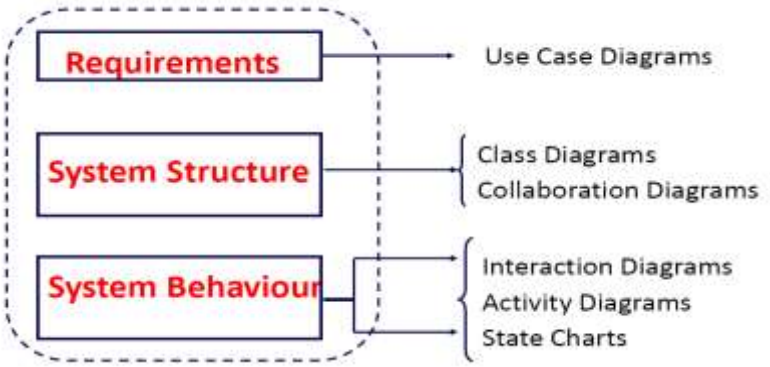
1- Behavior & Interaction models
2- Structural models



Different diagrams of system for different people



Key Diagrams in UML



Visual Case includes **Eight powerful UML Diagrams**. Objects that you've defined can be displayed on multiple diagrams, and many objects can be decomposed into new diagrams.

1- Use Case Diagram: The Use Case Diagram describes the system functionality as a set of Use-cases which represent discrete tasks. Actors interact with the system to complete the tasks.

2- Class Diagram: The Class Diagram describes the structure of the software system. This is the core diagram for object-oriented design.

3- Activity Diagram: The Activity Diagram describes the activities of a class in response to internal events.

4- State Diagram: State Diagrams model the dynamic behavior of a system by showing the various states that an object can get into and the transitions that occur between the states.

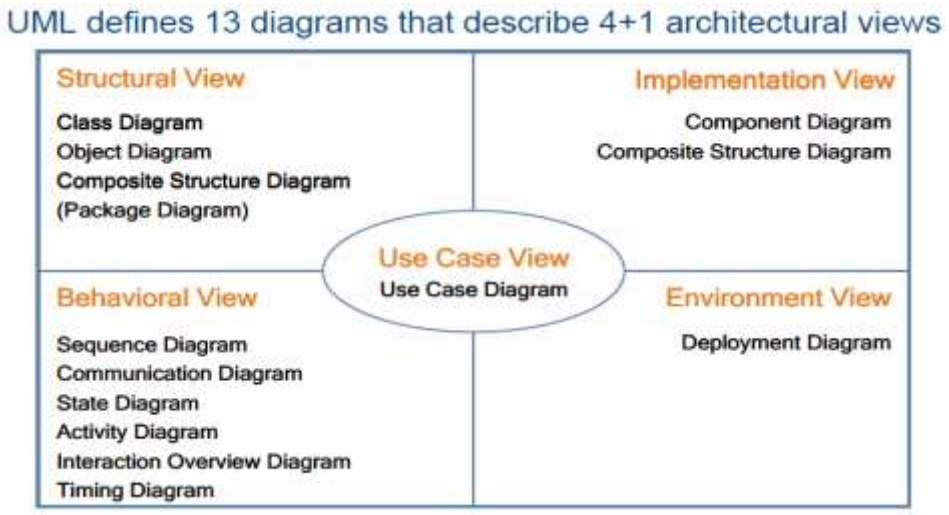
5- Sequence Diagram: The Sequence Diagram describes messages exchanged between classes to accomplish tasks.

3- Collaboration Diagrams: The Collaboration Diagram describes interactions between classes and associations.

7- Component Diagram: The Component Diagram describes the structure and dependencies among software components.

8- Deployment Diagram: The Deployment Diagram describes the physical layout of software components.

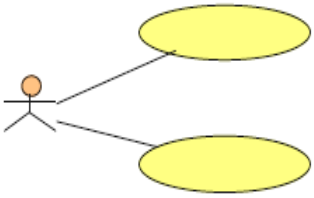
UML Architectural Views & Diagrams



Use Case View

The most important architectural view:

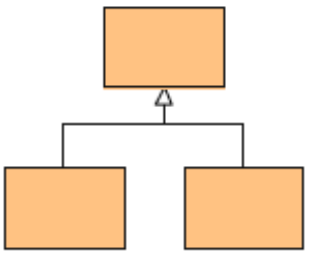
1- Describes use cases that provide value for the users.
2- Essential use cases are used as proof of concept for implementation architecture.
3- Use cases may be visualized in UML use case diagram.
4- Each use case may have multiple possible scenarios.
5- Use case scenarios could be described:
- Using textual descriptions;
- Graphically, using UML activity diagrams.



Structural View

Represents structural elements for implementing solution for defined requirements

- It Defines
  _Object-oriented analysis and design elements;
  _ Domain and solution vocabulary;
  _ System decomposition into layers and subsystems;
  _ Interfaces of the system and its components.

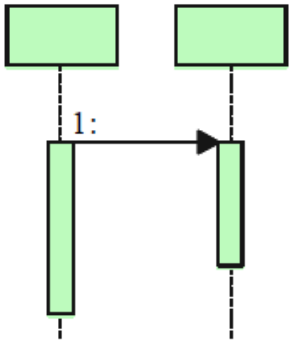


- Is represented by static UML diagrams:
  - Class diagrams in multiple abstraction levels;
  - Package diagrams;
  - Composite structure diagrams.

Behavioral View

Represents dynamic interaction between system components for implementing the requirements

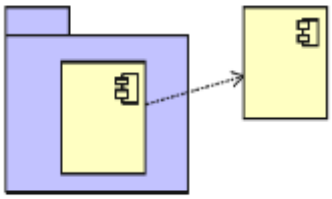
- Shows distribution of responsibilities.
- Allows identifying interaction and coupling bottlenecks.
- A means for discussing non-functional requirements (Performance, maintenance, …)
- Is especially important for distributed systems.
- Is represented by ***dynamic UML diagrams***:
  _ Sequence and/or communication diagrams;
  _ Activity diagrams;
  _ State diagrams;
  _ Interaction overview diagram;
  _ Timing diagrams.



Implementation View

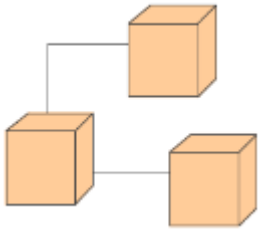
Describes implementation artifacts of logical subsystems defined in structural view

- May include intermediate artifacts used in system construction (code files, libraries, data files, ……..).
- Defines dependencies between implementation components and their connections by required and provided interfaces.
- Is represented by these UML diagrams:
  _ Component diagrams;
  _ Composite structure diagrams.



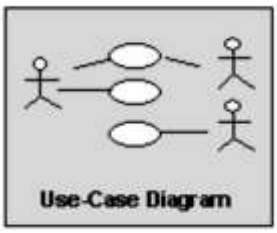
Environment View

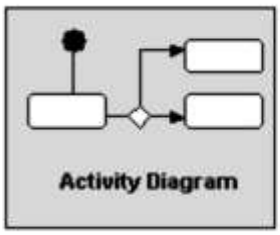
- Represents system's hardware topology.
- Defines how software components are deployed on hardware nodes.
- Useful for analyzing non-functional requirements (Reliability, scalability, security, ……).
- Provides information for system installation and configuration.
- Is represented by
  _ UML deployment diagram.

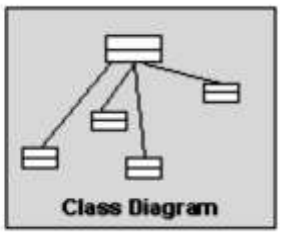


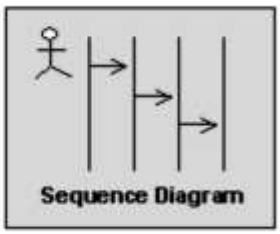
UML Diagrams

1- **Use case diagrams:** Describe the functional behavior of the system as seen by the user.
2- **Class diagrams:** Describe the static structure of the system: Objects, attributes, associations.
3- **Activity diagrams:** Describe the dynamic behavior of a system, in particular the workflow.
4- **Sequence diagrams:** Describe the dynamic behavior between objects of the system
5- **State diagrams:** Describe the dynamic behavior of an individual object.









What is a Scenario?

A scenario is a description of a person's interaction with a system. ***Scenarios*** help focus design efforts on the user's requirements, which are distinct from technical or business requirements. It may be related to 'use cases', which describe interactions at a technical level. Unlike use cases, however, scenarios can be understood by people who do not have any technical background.

They are therefore suitable for use during design activities.

When are scenarios appropriate?

1- Scenarios are appropriate whenever you need to describe a system interaction from the users Perspective.

2- Scenarios are particularly useful when you need to remove focus from the technology in Order to open up design possibilities, or when you need to ensure that technical or budgetary constraints do not override usability constraints without due consideration.

3- Scenarios can help confine complexity to the technology layer (where it belongs), and prevent it from becoming manifest within the user interface.

How do you use scenarios?

Use scenarios during design to ensure that all participants understand and agree to the design parameters, and to specify exactly what interactions the system must support. Translate scenarios into tasks for conducting walkthrough activities and usability tests.

A scenario describes how a user might interact with your system. Since there are several tasks that a user will undertake when interacting with your website or system, it is advisable to list those tasks and then create a scenario for each. In this way the scenarios would be more specific and more manageable.

Example:

Suppose that you have an ***e-commerce website*** from which you sell mobile phones. Ali is a 19-year old student who would like to buy a phone but is on a tight budget etc. In the scenario, you will put Ali in several "stories" based on the objectives that he would like to achieve when using your website.

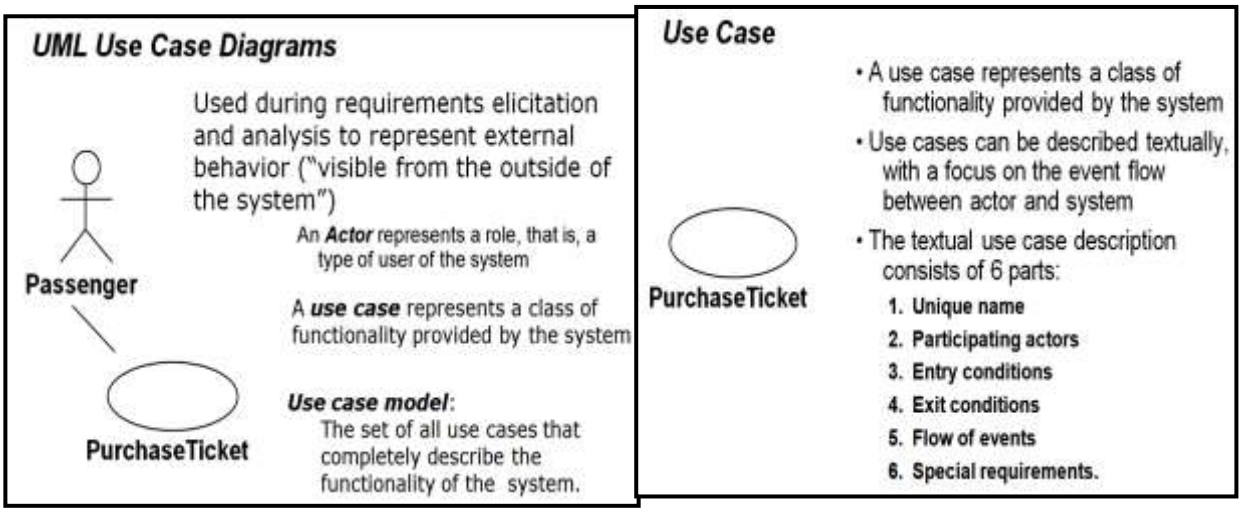
So, you might create a scenario in which Ali wants to achieve these objectives:

- Searching for a mobile phone for his budget
- Comparing the features of two different mobile phones
- Purchasing a mobile phone

1. Use case diagrams- Part 1

Use case diagrams describe what a system does from the standpoint of an external observer.
The emphasis is on ***what*** a system does rather than ***how*** (Describes the functionality provided by system).

- Use case diagrams are used during ***requirements elicitation and analysis*** as a graphical means of representing the functional requirements of the system.
- Use cases are developed during requirements elicitation and are further refined and corrected as they are reviewed (by stakeholders) during analysis.
- Use cases are also very helpful for ***writing acceptance test cases***. The test planner can extract scenarios from the use cases for test cases.



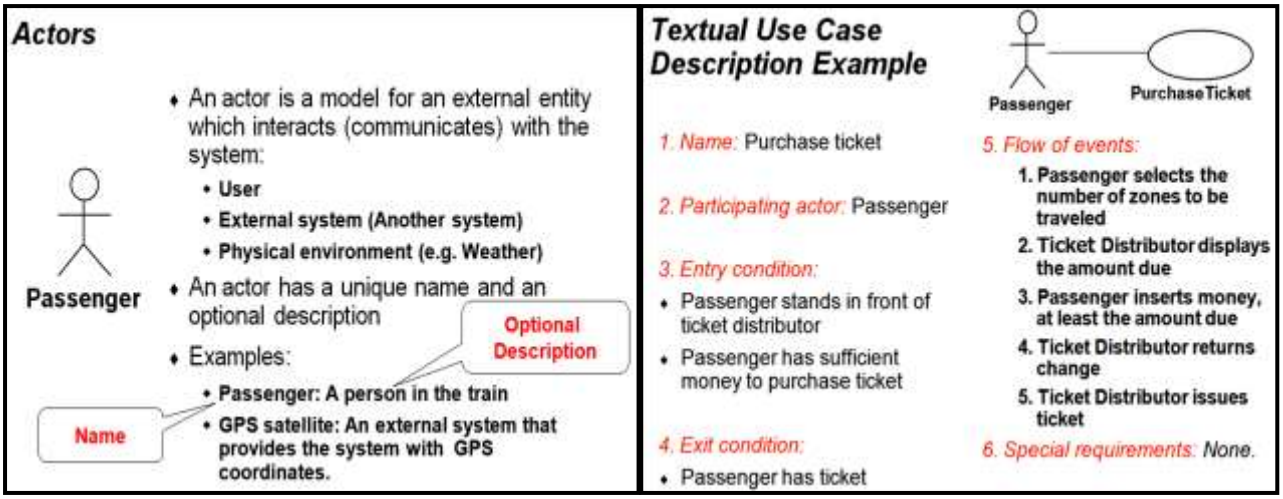
Note: The use case diagram is accompanied by a textual use case flow of events.
Use case diagrams are closely connected to scenarios. A **scenario** is an example of what happens when someone interacts with the system.

- A **use case** is a summary of scenarios for a single task or goal.
- An ***actor*** represents whoever or whatever (person, machine, or other) interacts with the system (i.e. An **actor** is who or what initiates the events involved in that task). Actors are simply roles that people or objects play.

The **actor** is not part of the system itself and represents anyone or anything that must interact with the system to:

- Input information to the system;
- Receive information from the system; or
- Both input information to and receive information from the system.

The total set of actors in a use case model reflects everything that needs to exchange information with the system. In UML, an "Actor" is represented as a *stickman*, "Use cases" are *ovals* and "Communications" are *lines* that link actors to use cases.



There are several different kinds of relationships between ***actors*** and ***use cases***.
The default relationship is the **«communicates»** relationship which indicates that one of these entities initiated invoked a request of the other. An actor communicates with use cases because actors want measurable results. When a line or an arrow is drawn on a diagram and there is no label on the arrow, it is, by default, a «communicates» relationship.

Example **Here is a scenario for a medical clinic:**

"**A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot.** "

The picture below is a **Make Appointment** use case for the medical clinic. The actor is a **Patient**. The connection between actor and use case is a **communication association** (or **communication** for short).

A use case diagram is a collection of actors, use cases, and their communications. In the following diagram, the **Make Appointment** concerned as a part of a diagram with four actors and four use cases.
Notice that a single use case can have multiple actors such as shown in Figure (4.1).

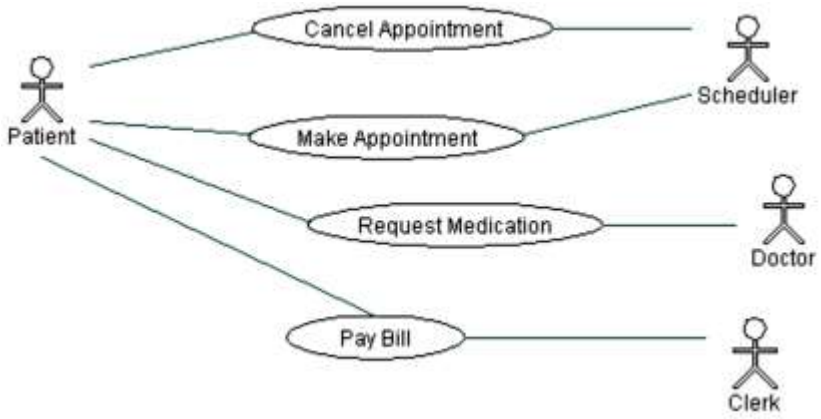


Figure (4.1): An example of Use Case diagram for "Make appointment" in medical clinic.

Use case diagrams are helpful in three areas:

1- **Determining features (requirements)**: new use cases often generate new requirements as the system is analyzed and the design takes shape.
2- **Communicating with clients**: their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
3- **Generating test cases**: the collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

As an example (H.W.):

In a Monopoly game, some use cases are:
- **Enter Player Info**,
- **Buy House**, and
- **Draw Card**.

Give your use case a unique name expressed in a few words (generally no more than five words). These few words must begin with a present-tense verb phrase in active voice, stating the action that must take place (notice: **Enter** Player Info**, Buy** House, **Draw** Card, and **Switch** Turn).

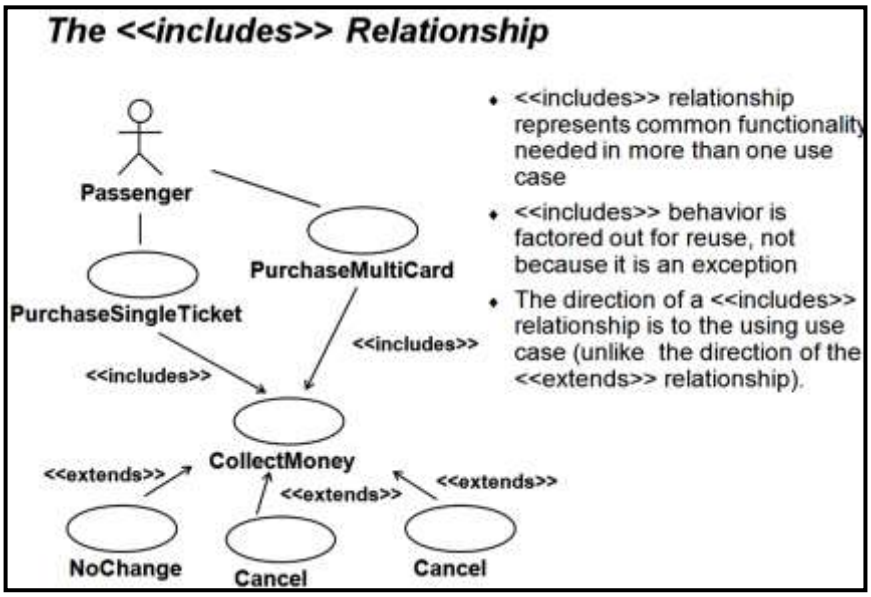
Solution:

1. Use case diagrams- Part 2

There are two other kinds of relationships between use cases (not between actors and use cases) that you might find useful. These are **«include»** and **«extend».**

1- «include» relationship

You use the **«include»** relationship when a chunk of behavior is similar across more than one use case, and you don't want to keep copying the description of that behavior. This is similar to breaking out re-used functionality in a program into its own methods that other methods invoke for the functionality. For example, suppose many actions of a system require the user to login to the system before the functionality can be performed. These use cases would *include* the login use case. The «**include**» relationship is not the default relationship.



Therefore in a use case diagram, the arrow is labeled with «include» when ***one use case makes full use of another use case,*** as shown in the following Figure (5.1). The "Draw Card" and the "Buy House" both use the View Information functionality.

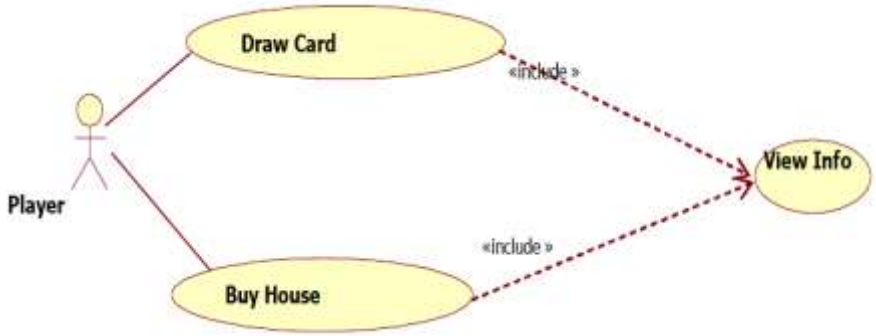
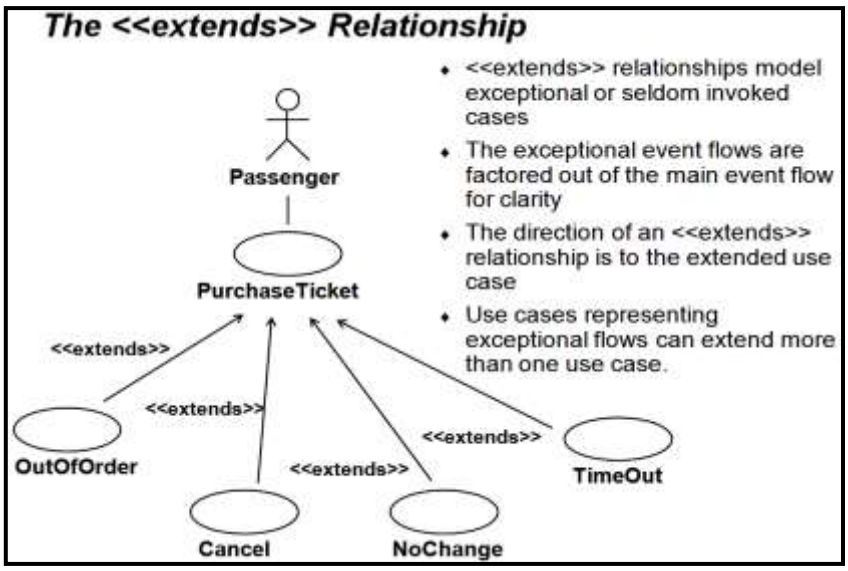


Figure 5.1: Includes Use Case

2- «extend» relationship

You use the **«extend»** relationship when you are describing a variation on normal behavior or behavior that is only executed under certain, stated conditions. The extend relationship is used when the ***alternative flow is fairly complex and/or multi-stepped***, possibly with its ***own sub-flows and alternative flows***.



System Boundary Diagram

Figure (5.2) shows a System Boundary Diagram. The large rectangle is the system boundary. Everything inside the rectangle is part of the system under development. Outside the rectangle we see the ***actors*** that *act* upon the system. Actors are entities outside the system that provide the stimuli for the system. Typically they are human users. They might also be other systems, or even devices such as real-time clocks.

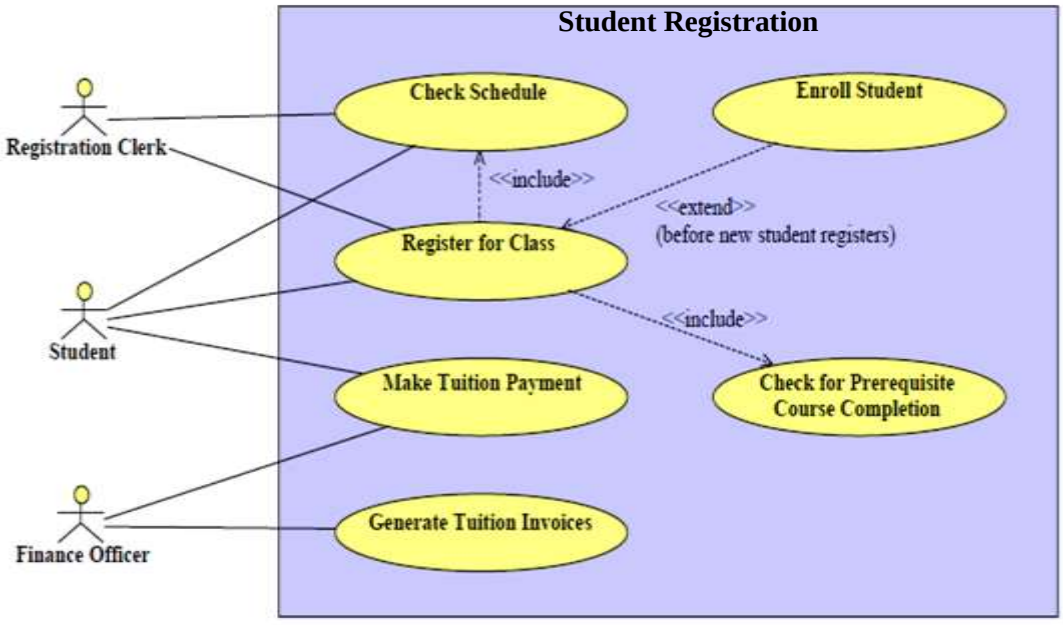


Figure 5.2: System Boundary Diagram

2. Activity diagrams- Part1

- **Activity diagrams** are the object-oriented equivalent of flow charts and data-flow diagrams from structured development. It describes the workflow behavior of a system.

- **Activity diagram** illustrates the dynamic nature of a system by modeling the flow of control from activity to activity.

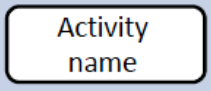
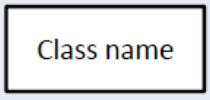
- **Activity diagrams** are used during the design phase of complex methods. It can also be used during analysis to break down the complex flow of a use case. Through an activity diagram, the designer/analyst specifies the essential sequencing rules the method or use case has to follow.

Applications of Activity Diagrams

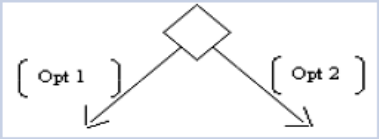
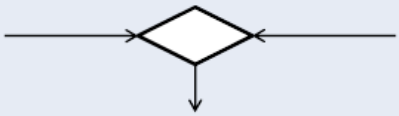
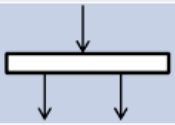
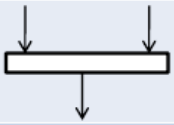
1. Complex operations
2. Business Rule
3. Functions that occur in parallel
4. Single use case
5. Complex chain of multiple use cases
6. Software flows and logic control configurations

UML activity diagrams are an updated and enhanced form of flowcharts; the main enhancement over flowcharts is the ability to handle parallelism. An activity is a single step that needs to be done, whether by a human or a computer.
Incoming transitions (an incoming arrow) trigger the **activity**. If there are several incoming transitions, any of these can trigger the activity independent of the others.

Elements of activity diagram

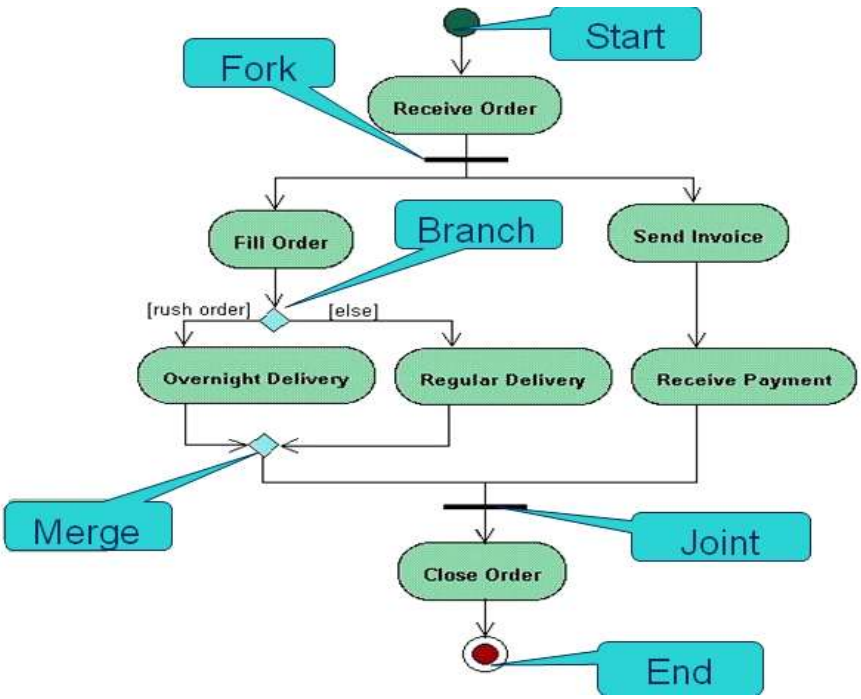
Description	Symbol
Activity : Is used to represent a set of actions	
A Control Flow: Shows the sequence of execution	
An Object Flow: Shows the flow of an object from one activity (or action) to another activity (or action).	
An Initial Node: Portrays the beginning of a set of actions or activities	
A Final-Activity Node: Is used to stop all control flows and object flows in an activity (or action)	
An Object Node: Is used to represent an object that is connected to a set of Object Flows.	

Elements of activity diagram

Description	symbol
A Decision Node: Is used to represent a test condition to ensure that the control flow or object flow only goes down one path	
A Merge Node: Is used to bring back together different decision paths that were created using a decision-node.	
A Fork Node:Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)	
A Join Node: Is used to bring back together a set of parallel or concurrent flows of activities (or actions).	
A Swimlane :A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread	

Example 1: Processing an order

Once the order is **received** the activities **split** into two **parallel** sets of activities. One side **fills** and **sends** the order while the other handles the billing. On the **Fill Order** side, the method of delivery is decided conditionally. Depending on the **condition** either the **Overnight Delivery** activity or the **Regular Delivery** activity is performed. Finally the **parallel** activities **combine** to close the order.



Parallel Activities:

1- It is possible to show that activities can occur in parallel, as seen in Example (1) depicted using two parallel bars:

i- First bar is called a **fork**, it has one transition entering it and two or more transitions leaving it

ii- Second bar is a **join**, with two or more transitions entering it and only one leaving it

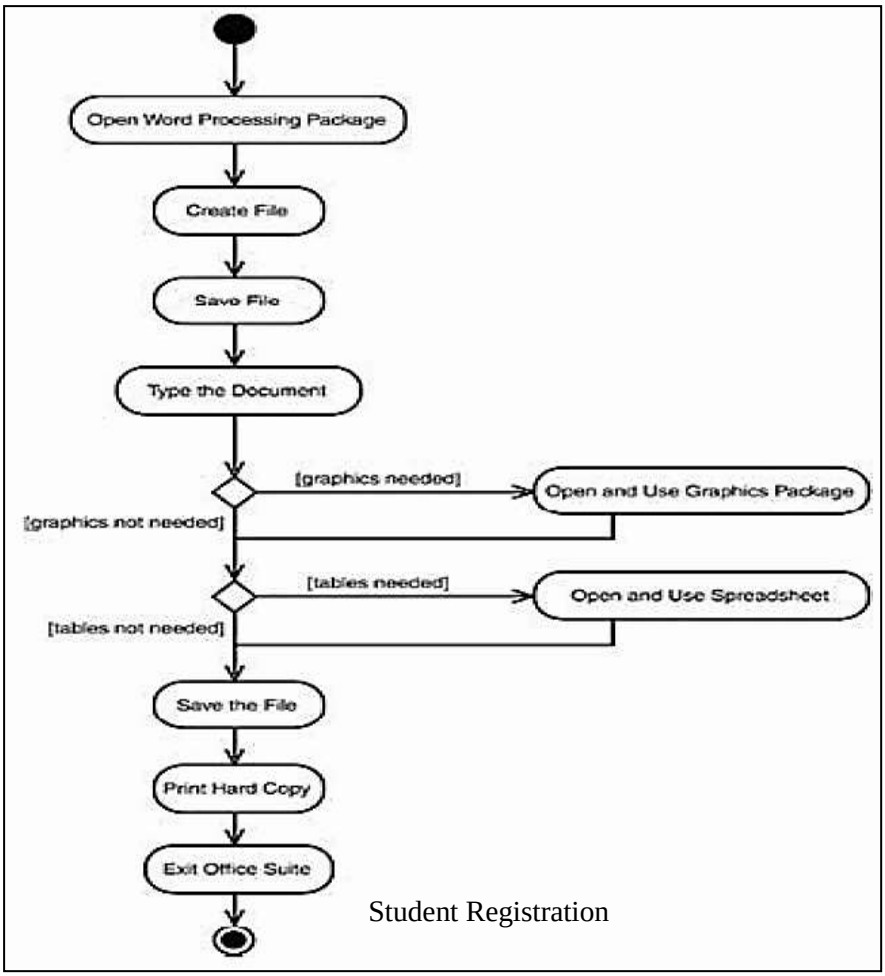
2- A **Fork** Should Have a Corresponding **Join**. In general, for every start (fork) there is an end (join).

3- Forks Have **One Entry Transition**.

4- Joins Have **One Exit Transition**.

Example 2 : Creating document

1. **Open** the word processing package.
2. **Create** a file.
3. **Save** the file under a unique name within its directory.
4. **Type** the document.
5. **If** graphics are necessary, **open** the graphics package, create the graphics, and paste the graphics into the document.
6. **If** a spreadsheet is necessary, **open** the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
7. **Save** the file.
8. **Print** a hard copy of the document.
9. **Exit** the word processing package.

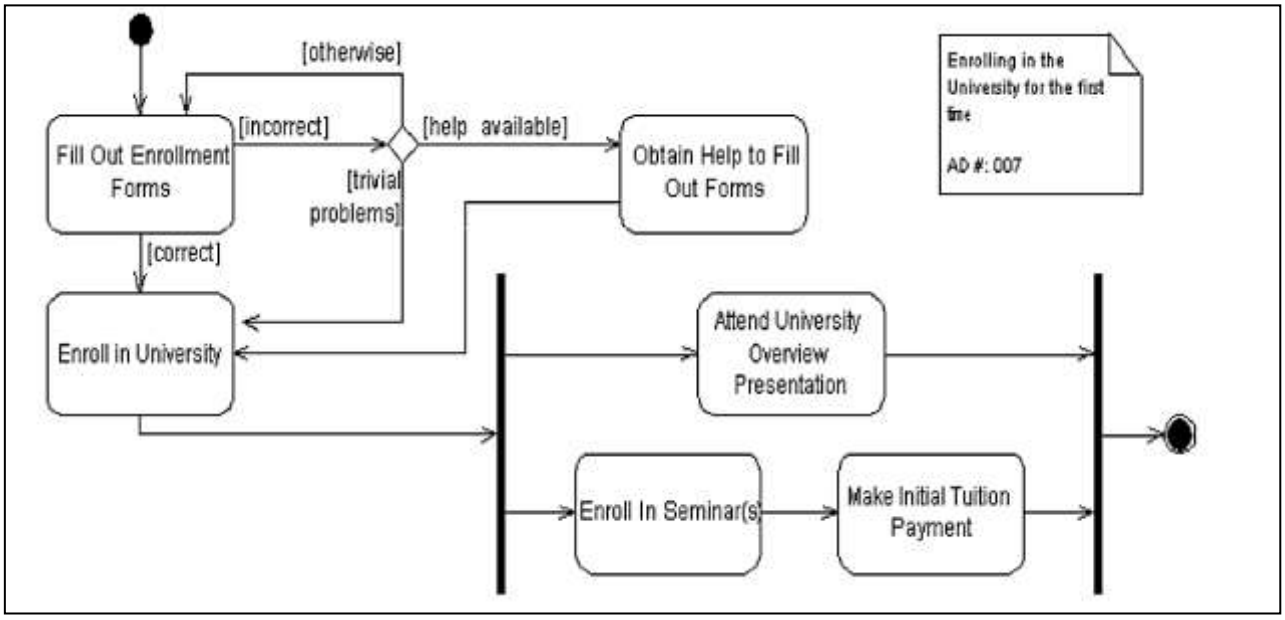


Student Registration

2. Activity diagrams- Part2

Example 3: "Enrollment in university"

1. An applicant wants to **enroll in the university**.
2. The applicant hands **a filled out copy of form** 'University_Application' to the registrar.
3. The registrar inspects the forms.
4. The registrar determines that the forms have been **filled out properly**.
5. The registrar informs student to **attend in university overview presentation**.
6. The registrar helps the student to **enroll in seminars**.
7. The registrar asks the student to **pay the initial tuition fees**.



Swimlane Guidelines

A **swimlane** is a way to group activities performed by the same actor on an Activity diagram or to group activities in a single thread.

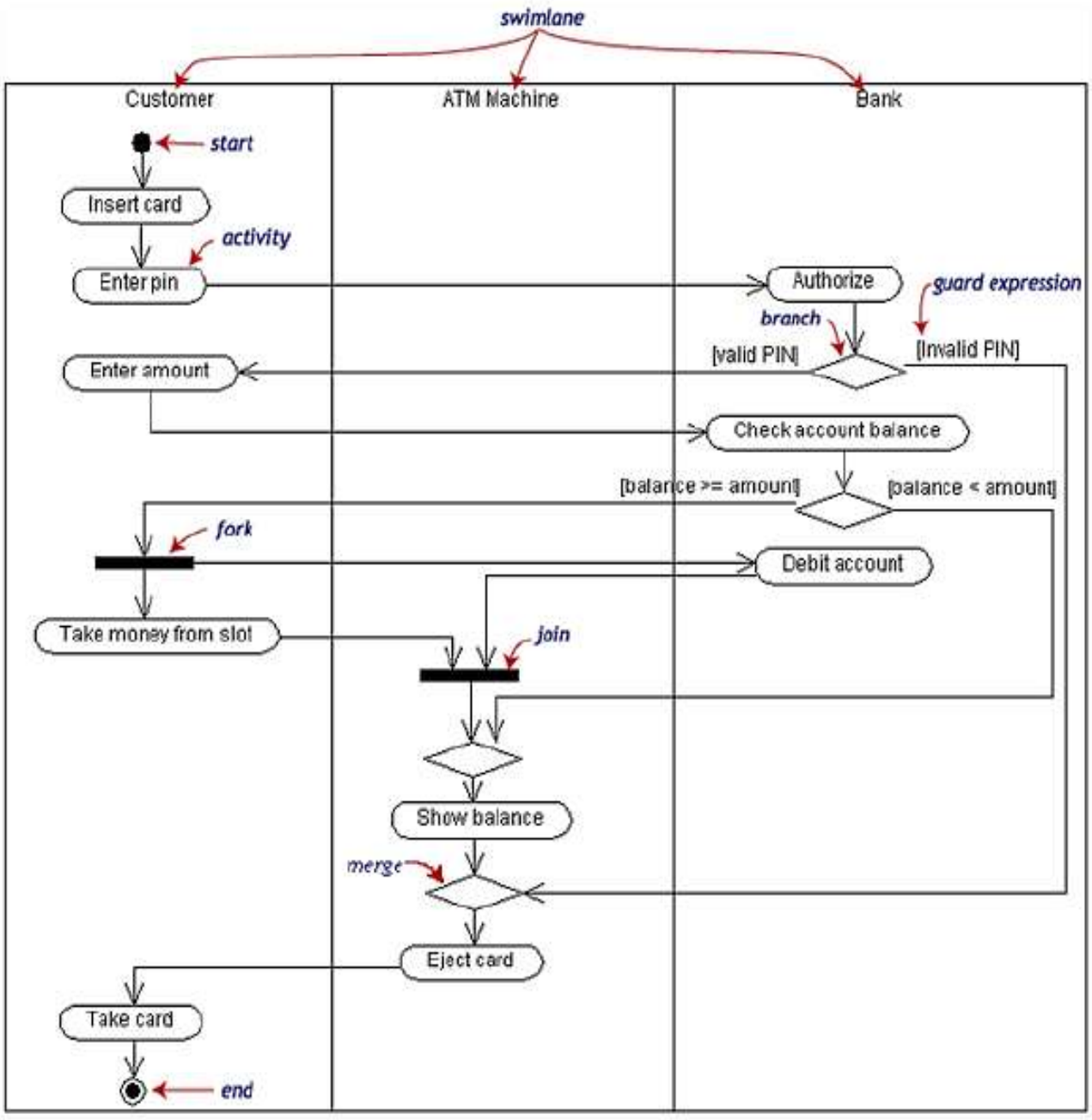
- ***Activity diagrams*** can be divided into object **swimlanes** that determine which object is responsible for which activity. A single **transition** comes out of each activity, connecting it to the next activity. A transition may **branch** into two or more mutually exclusive transitions.

- **Guard expressions** (inside [ ]) label the transitions coming out of a branch. A branch and its subsequent **merge** marking the end of the branch appear in the diagram as hollow diamonds.

- A transition may **fork** into two or more parallel activities. The fork and the subsequent **join** of the threads coming out of the fork appear in the diagram as **solid bars**.

Example: ***"Withdraw money from a bank account through an ATM."***

The three involved classes (people, etc.) of the activity are **Customer**, **ATM**, and **Bank**. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.



When to Use: Activity Diagrams

1- Activity diagrams are used in alignment with other modelling techniques like interaction diagrams and State diagrams.
2- It is used to model the work flow behind the system being designed.
3- Activity diagrams are useful for analysing a use case by describing what actions need to take place and when they should occur, describing a complicated sequential algorithm and modelling applications with parallel processes.

Activity diagrams advantages:

1- Activity diagrams are normally easily comprehensible for both analysts and stakeholders.
2- Activity diagrams allow an analyst to display multiple conditions and actors within a work flow through the use of Swimlanes.
3- It is useful for showing the dependencies between use cases: e.g. workflow of an organization.
4- Can show parallel activities, so make dependencies and non-dependencies explicit.
5- It is the best way to document dependencies between use cases.

Activity diagrams disadvantages:

1- Not automatically clear who/what carries out an activity; can be hard to make the connection with underlying objects.
2- Activity diagrams do not give detail about how objects behave or how objects collaborate.

3. Class Diagrams- Part1

- **Class diagrams are used in both the *analysis* and the *design* phases. During the analysis phase, a very high-level conceptual design is created.** At this time, a class diagram might be created with only the class names shown or possibly some pseudo code-like phrases may be added to describe the responsibilities of the class.

- **The Class diagram created during the *analysis phase* is used to describe the classes and relationships in the problem domain, but it does not suggest how the system is implemented.** By the end of the design phase, class diagrams that describe ***how the system to be implemented should be developed.***

- **The class diagram created after the design phase has detailed implementation information, including the class names, the methods and attributes of the classes, and the relationships among classes.**

- **The class diagram describes the types of objects in a system and the various kinds of static relationships that exist among them.**

In UML, a class is represented by a rectangle with one or more horizontal compartments.
The upper compartment holds the name of the class. The name of the class is the only required field in a class diagram. By convention, the class name starts with a capital letter. The (optional) center compartment of the class rectangle holds the list of the class attributes/data members, and the (optional) lower compartment holds the list of operations/methods.
The complete UML notation for a class is shown in Figure 8.1.

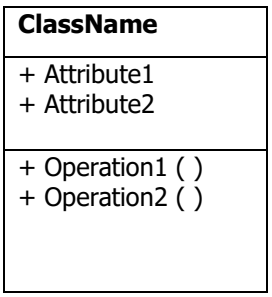
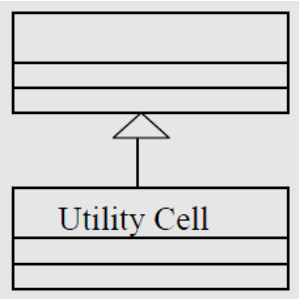


Figure 8.1: UML notation for a class.

Static Relationships:

There are two principle types of static relationships between classes:
inheritance and ***association***. The relationships between classes are drawn on class diagram by various lines and arrows.
Inheritance (termed "***generalization***" for class diagrams) is represented with an empty arrow, pointing from the subclass to the superclass, as shown in this Figure.



UtilityCell inherits from Cell (a.k.a UtilityCell "is-a" specialized version of a Cell). The subclass (UtilityCell) inherits all the methods and attributes of the superclass (Cell) and may override inherited methods.

- An **association** represents a relationship between two instances of classes.
- An association between two classes is shown by a line joining the two classes.
- **Association indicates that one class utilizes an attribute or methods of another class.** If there is no arrow on the line, the association is taken to be bi-directional, that is, both classes hold information about the other class.
- A unidirectional association is indicated by an arrow pointing from the object which holds to the object that is held.

-There are two different specialized types of association relationships:
aggregation, and ***composition***.

If the association conveys the information that one object is part of another object, but their lifetimes are independent (they could exist independently), this relationship is called ***aggregation***.

For example, we may say that "a Department contains a set of Employees," or that "a Faculty contains a set of Teachers." Where generalization can be thought of as an "is-a" relationship, aggregation is often thought of as a "has-a" relationship – "a Department 'has-a' Employee."

- **Aggregation** is implemented by means of one class having an attribute whose type is in included class (the Department class has an attribute whose type is Employee).
- **Aggregation** is stronger than association due to the special nature of the "has-a" relationship. Aggregation is unidirectional: there is a container and one or more contained objects.
- **An aggregation** relationship is indicated by placing a white diamond at the end of the association next to the aggregate class, as shown in Figure 8.2.

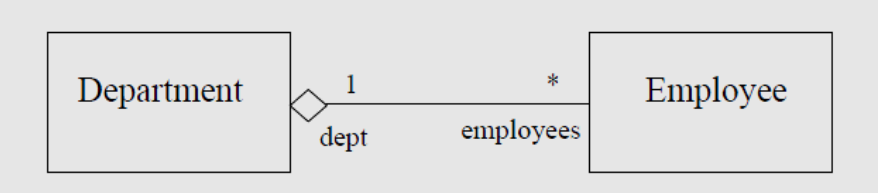


Figure 8.2: Aggregation.

Even stronger than aggregation is ***Composition***. There is composition when an object is contained in another object, and it can exist only as long as the container exists and it only exists for the benefit of the container.
Examples of composition are the relationship Invoice-InvoiceLine, and Drawing-Figure. An invoice line can exist only inside an invoice, and a specific geometric figure only inside a drawing. Any deletion of the whole (Invoice) is considered to cascade to all the parts (the InvoiceLine's are deleted).

Composition is shown by a black diamond on the end of association next to the composite class, as shown in Figure 8.3. In this Figure, we show also the fact that the relationship between a Gameboard and its Cells can be navigated only from Gameboard to Cell (an arrow points from Gameboard to Cell).
Therefore, this relationship is a composition, and not an aggregation.

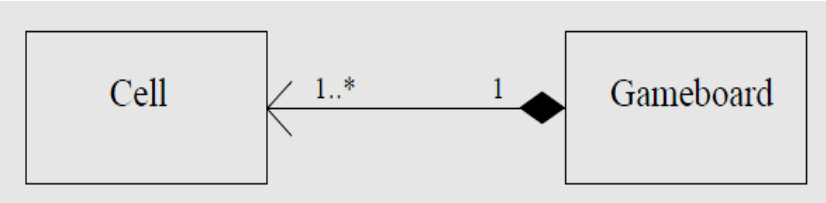


Figure 8.3: Composition.

To summarize – aggregation is a special form of association; composition is a stronger form of aggregation. Both aggregation and composition are a part-whole hierarchy.

Attributes and Operations

Attributes or data members are shown in the middle box of the class diagram. It is optional to show the attributes. When an attribute is included, it is possible to only specify the name of the attribute.
UML notation also allows showing their type (the class of the data type of the attribute), their default value, and their visibility with respect to access from outside the class. **Public attributes** are denoted with a (+) sign, **protected** with a (#) sign, and **private** with a (-) , as shown in Figure 8.4.

The UML syntax for an attribute is:

visibility name : type = defaultValue

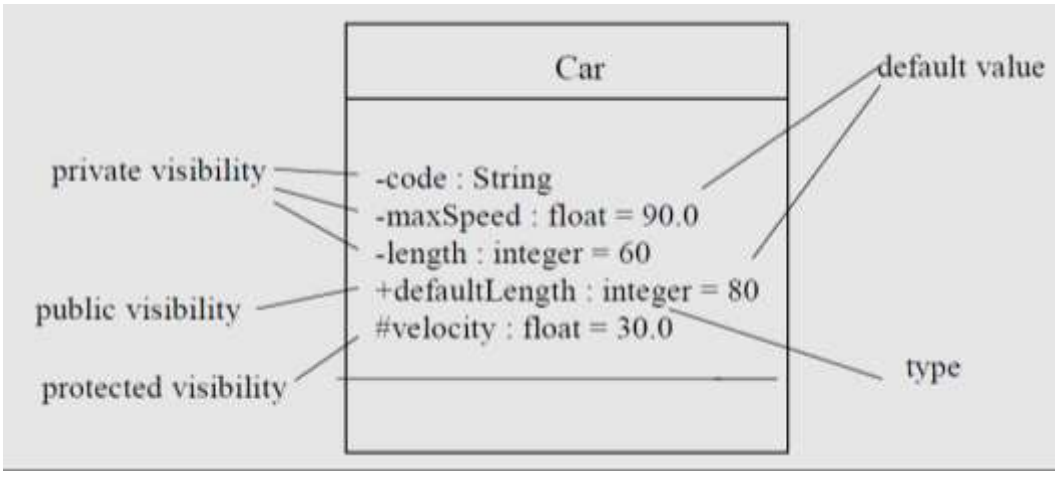


Figure 8.4: Notation for attributes.

The third and bottom compartment of class symbol in UML notation holds a list of class operations or methods.

The operations are the services that a class is responsible for carrying out. They may be specified giving their signature (the names and types of their arguments/parameters), the return type, and their visibility (private, protected, public) may be shown. An optional property string indicates property values that apply to the operation. UML notation for operations/methods is shown in Figure 8.5.

The UML syntax for an operation is:

visibility name(parameter-list) : return-type{property string}

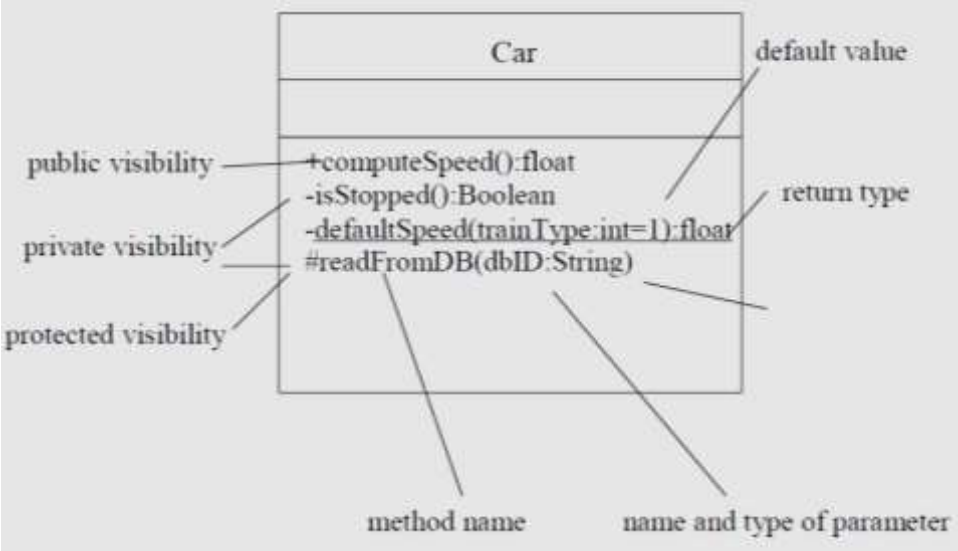


Figure 8.5: UML notation for operations/methods.

Multiplicity

Associations have a multiplicity (sometimes called cardinality) **that indicates how many objects of each class can legitimately be involved in a given relationship.**

Multiplicity is expressed by the "***n..m***" symbol put near to the association line, close to the class whose multiplicity in the association we want to show. Here **"*n*"** refers to the minimum number of class instances that may be involved in the association, and **"*m*"** to the maximum number of such instances.
If ***n = m***, only an "*n*" is shown. An optional relationship is expressed by writing "0" as the minimum number.
Table 8.1 shows the most common cases of multiplicity.

Table 8.1: Multiplicity notation

Cardinality and modality	UML symbol
One-to-one and mandatory	1
One-to-one and optional	0..1
One-to-many and mandatory	1..*
One-to-many and optional	*
With lower bound *l* and upper bound *u*	1..u
With lower bound *l* and no upper bound	1..*

As shown in following Figure 8.6., we demonstrate several of the aspects of association and multiplicity.

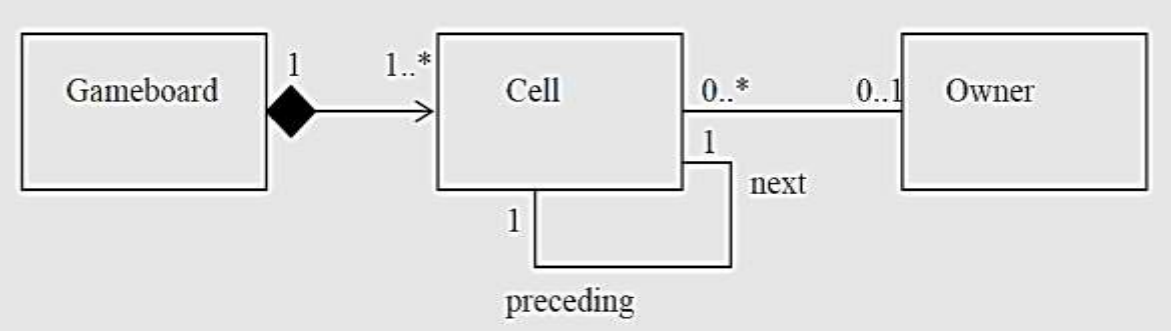


Figure 8.6: An UML class diagram with three classes, their associations, and multiplicity.

Table 8.2 summarizes the associations between these three classes. Notice the "next" and "preceding" labels on the Cells association. These are called "roles." Labeling the end of associations with role names allows us to ***distinguish multiple associations originated from a class and clarify the purpose of the association.***

Table 8.2: Details about the associations

Classes of association	Kind	Information held
Gameboard, Cell	Composition	A gameboard contains one or more cells. A cell is contained in one and only one gameboard. The gameboard can access its sections but the cells do not need to access their gameboard. The cells cannot exist in isolation, but only if contained by a gameboard.
Cell, Cell	Association	Every Cell is associated with, and must be able to access, its *next* Cell and its *preceding* Cell, along the Gameboard.
Cell, Player	Association	A Cell is owned by zero or more Owners. An Owner owns zero or more Cells. The Cell can access its Owner, and the Owner can access the Cells it owns.

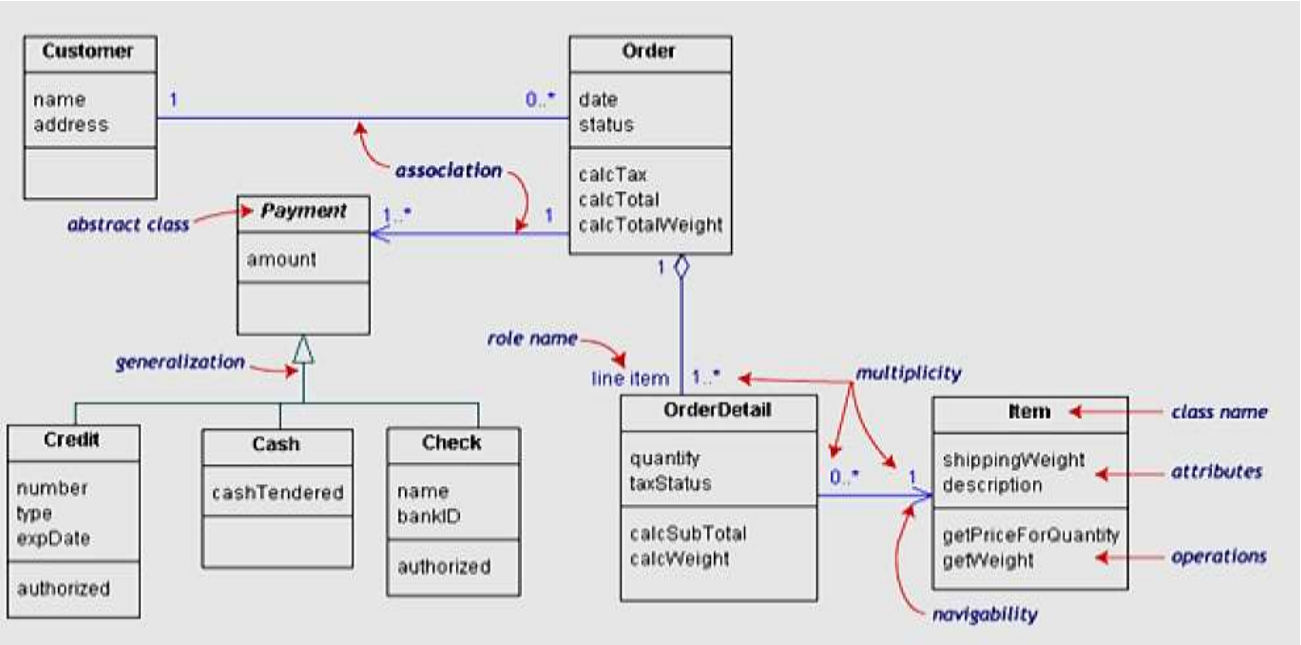
3. Class Diagrams-Part2

Example 1: (Online Shopping)

A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static -- they display what interacts but not what happens when they do interact.

The **class diagrams** below models for problem: "**Online shopping**" which present that a customer order from a retail catalog. The central class is the **Order**. Associated with it is the **Customer** making the purchase and the **Payment**.

A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as ***Payment***, are in italics. Relationships between classes are the connecting links.

The class diagram has *three kinds* of relationships:

1- **Association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

2- **Aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.

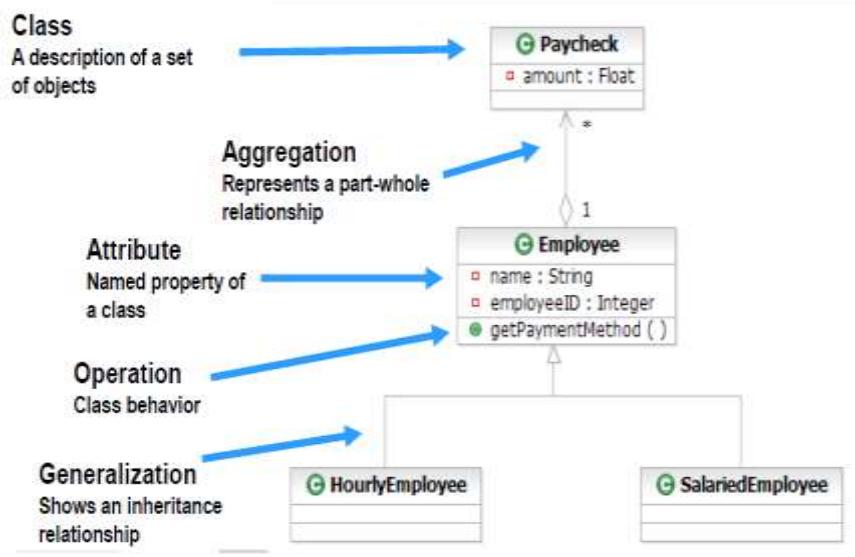
3- **Generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

An association has two ends. An end may have a **role name** to clarify the nature of the association. For example, an **OrderDetail** is a line item of each **Order**.
A **navigability** arrow on an association shows which direction the association can be traversed or queried. An **OrderDetail** can be queried about its **Item**, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, **OrderDetail** has an **Item**. Associations with no navigability arrows are bi-directional.

The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.
This table gives the most common multiplicities.

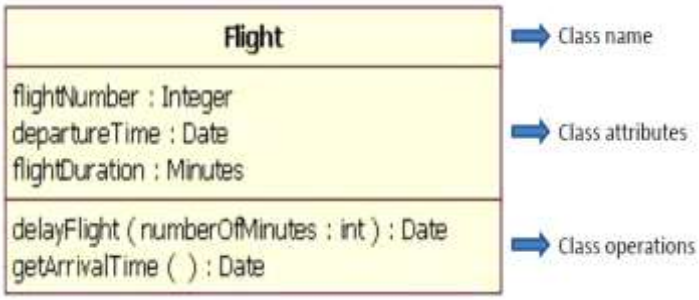
Multiplicities	Meaning
0..1	zero or one instance. The notation *n* . . *m* indicates *n* to *m* instances.
0..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Every class diagram has **classes**, **associations**, and **multiplicities**. Navigability and roles are optional items placed in a diagram to provide clarity.

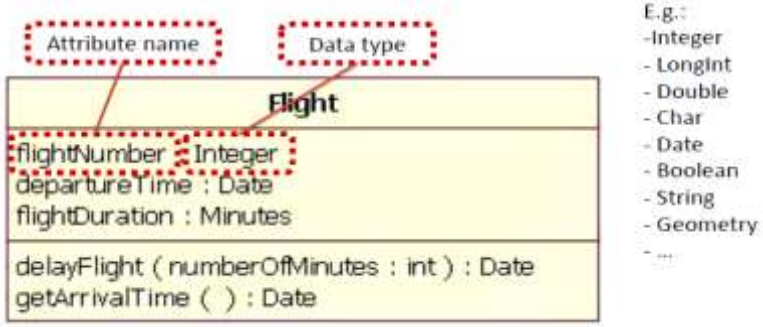


UML – Exercise (Flight- Booking) using Class Diagram

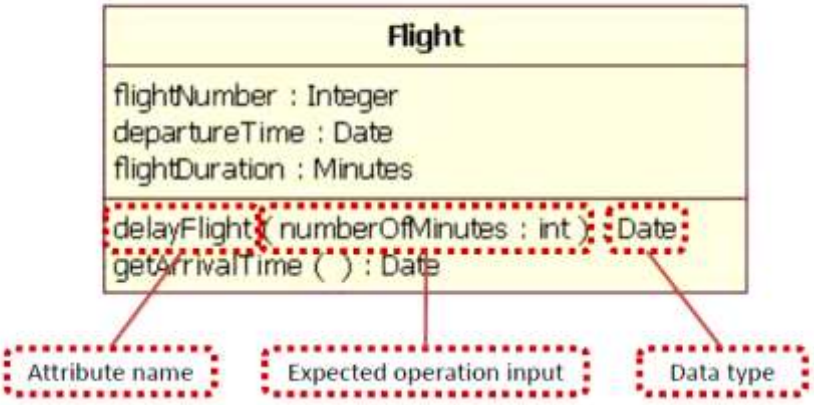
- Summarize a number of objects with the same behavior and semantics



- The class attribute



- The class operations



H.W. (2) : ***Design a Class Diagram of problem above "Flight-Booking"***

4. State Diagram

- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioral** diagram and it represents the behavior using finite state transitions.
- A **state diagram** is used to model the dynamic behavior of a class in response to time and changing external stimuli.

Uses of state diagram :

1. It used to state the events responsible for change in state (we do not show what processes cause those events).
2. It used to model the dynamic behaviour of the system.
3. To understand the reaction of objects/classes to internal or external stimuli.

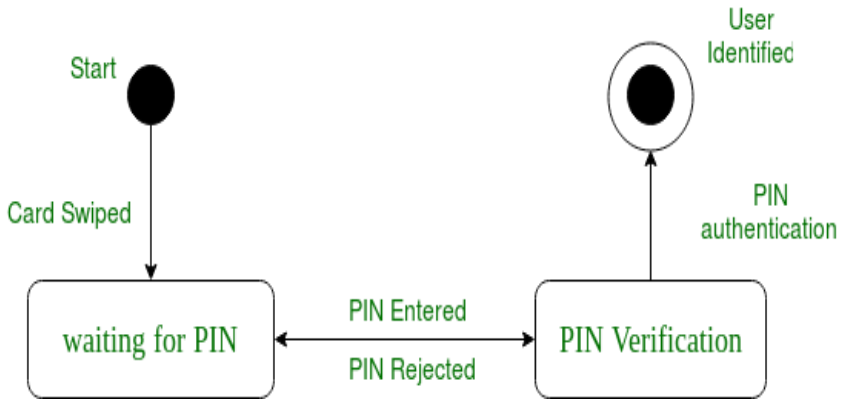


Figure – a state diagram for user verification.

This state diagram shows the different states in which the verification sub-system or class exist for a particular system.

Basic components of a state diagram:

1. **Initial state –** We use a black filled circle represent the initial state of a System or a class.

Figure – initial state notation

2. **Transition –** We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labeled with the event which causes the change in state.

Figure – transition

3. **State –** We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.

Figure – state notation

4. **Fork –** We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.

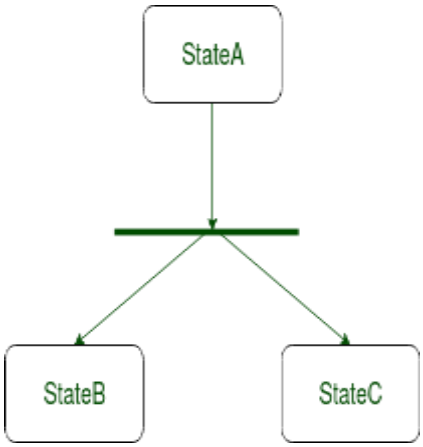


Figure – a diagram using the fork notation

5. **Join –** We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.

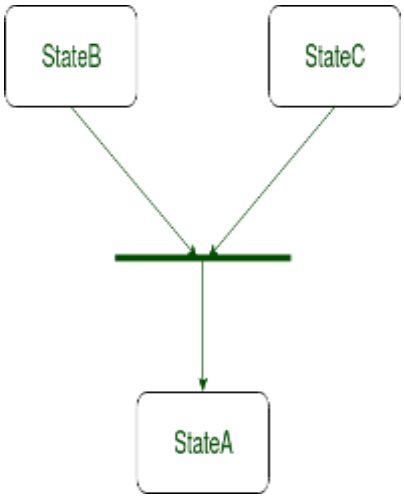


Figure – a diagram using join notation

6. **Self-transition –** We use a solid arrow pointing back to the state itself to represent a self -transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self- transitions to represent such cases.

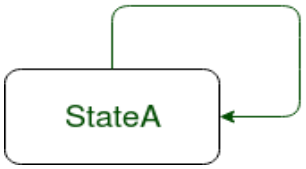


Figure – self transition notation

7. **Composite state –** We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

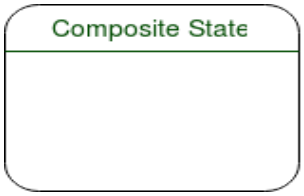


Figure – a state with internal activities

8. **Final state –** We use a filled circle within a circle notation to represent the final state in a state machine diagram.

Figure – final state notation

Steps to draw a state diagram:

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

Example – State diagram for an **"Online ordering"** :

1. On the event of an order being received, we transit from our initial state to unprocessed order state.
2. The unprocessed order is then checked.
3. If the order is rejected, we transit to the Rejected Order state.
4. If the order is accepted and we have the items available we transit to the fulfilled order state.
5. However if the items are not available we transit to the Pending Order state.
6. After the order is fulfilled, we transit to the final state.

In this example, we merge the two states i.e. ***Fulfilled order*** and ***Rejected order*** into one final state.

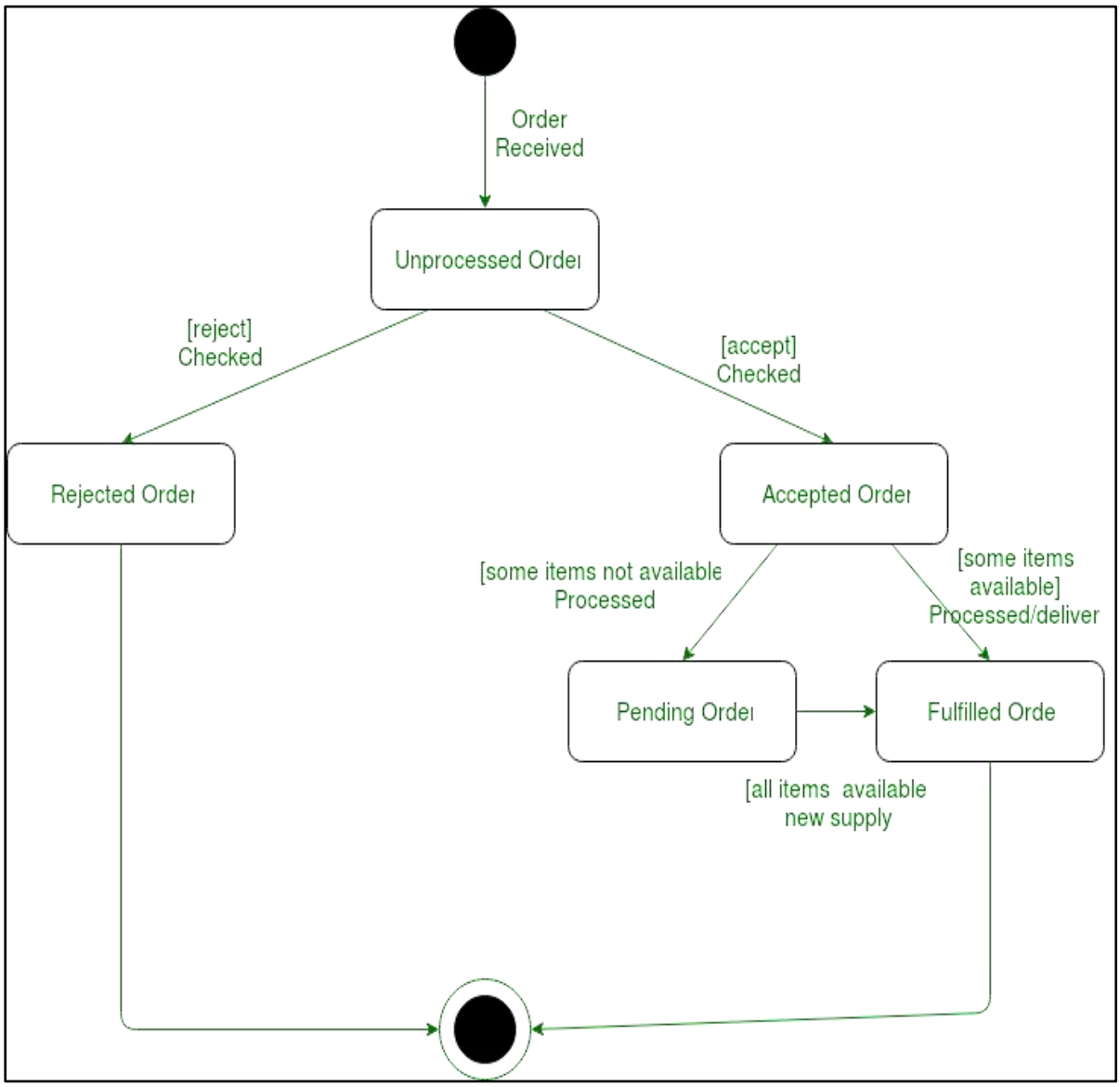


Figure – state diagram for an online order

Note : Here we could have also treated fulfilled order and rejected order as final states separately.