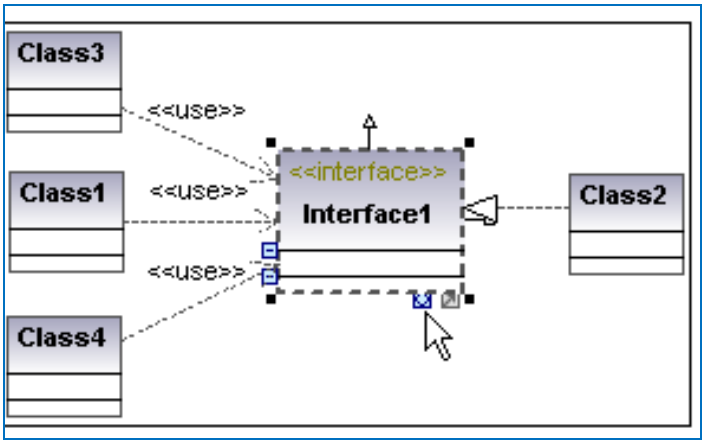
UNESCO-NIGERIA TECHNICAL &
VOCATIONAL EDUCATION
REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



Unified Modelling Language (UML)

(COM213)

YEAR 2-SEMESTER 1

PRACTICAL

Version 1: December 2008

Table of Contents

WEEK One (practical)

Objectives

At the end of the practical week the students should

- Understand the concepts of OOP
- Model the concepts to real life scenario

Lab1

Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world.

What Is an Object?

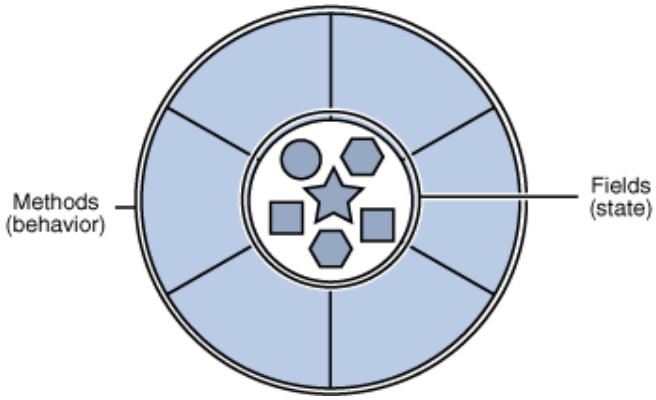
An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. Real life objects are anything that have attribute (state) and behaviour (method). Such real world objects are Cats, Car, Ball, table etc.

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn

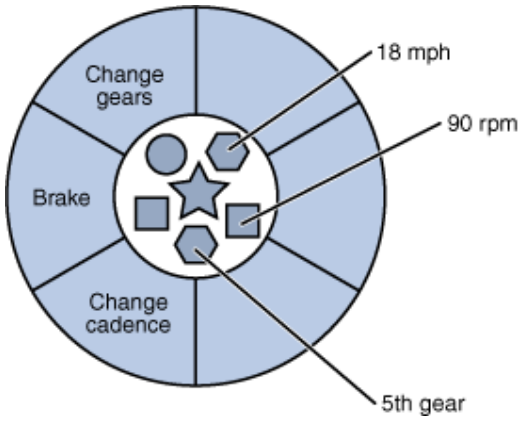
on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



A bicycle modeled as a software object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

What Is a Class?

A class is a blueprint or prototype from which objects are created. In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

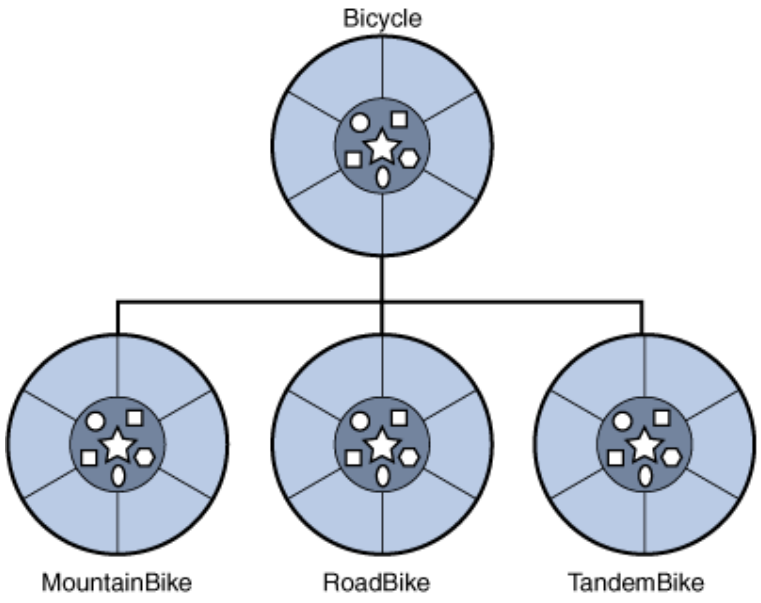
What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of

bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike.



1. A hierarchy of bicycle classes.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

Exercise

1. Real-world objects contain ___ and ___.
2. A software object's state is stored in ___.
3. A software object's behavior is exposed through ___.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data ___.
5. A blueprint for a software object is called a ___.
6. Common behavior can be defined in a ___ and inherited into a ___ using the ___ keyword.

7. A collection of methods with no implementation is called an ___.
8. A namespace that organizes classes and interfaces by functionality is called a ___.
9. The term API stands for ___?

Answers to Exercises: Object-Oriented Programming Concepts

1. Real-world objects contain **state** and **behavior**.
2. A software object's state is stored in **fields**.
3. A software object's behavior is exposed through **methods**.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data **encapsulation**.
5. A blueprint for a software object is called a **class**.
6. Common behavior can be defined in a **superclass** and inherited into a **subclass** using the **extends** keyword.
7. A collection of methods with no implementation is called an **interface**.
8. A namespace that organizes classes and interfaces by functionality is called a **package**.
9. The term API stands for **Application Programming Interface**.

WEEK Two (practical)

Objectives

At the end of the practical week the students should

- Be introduced and understand the concept of UML and its Importance to software development
- Identify and load applications such as MS-Word that aids in drawing symbols
- Mention and draw notations that are accepted as models in other fields of specialization

Concept of UML

UML stands for Unified Modelling Language whose purposes is to provide the development community with a stable and common design language that could be used to develop and build computer applications. UML brought forth a unified standard modelling notation that IT professionals had been wanting for years. The idea is to create standardization for the building and defining software engineering just like every other field of specialization. Various Symbols are generally accepted as notations to represent major events.

Standardization

UML is officially defined by the Object Management Group (OMG) as the UML metamodel, a Meta-Object Facility metamodel (MOF). Like other MOF-based specifications, UML has allowed software developers to concentrate more on design and architecture.
UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages, supported by the OMG.

UML diagrams

UML recognizes 13 different symbols which are subgroup to three of structure, behaviour and interaction models;

Structure Model

- Class diagram
- Composite structure diagram:
- Component diagram:
- Deployment diagram:
- Object diagram:
- Package diagram:

Behaviour Mode

- Activity diagram:
- State diagram:
- Use case diagram:

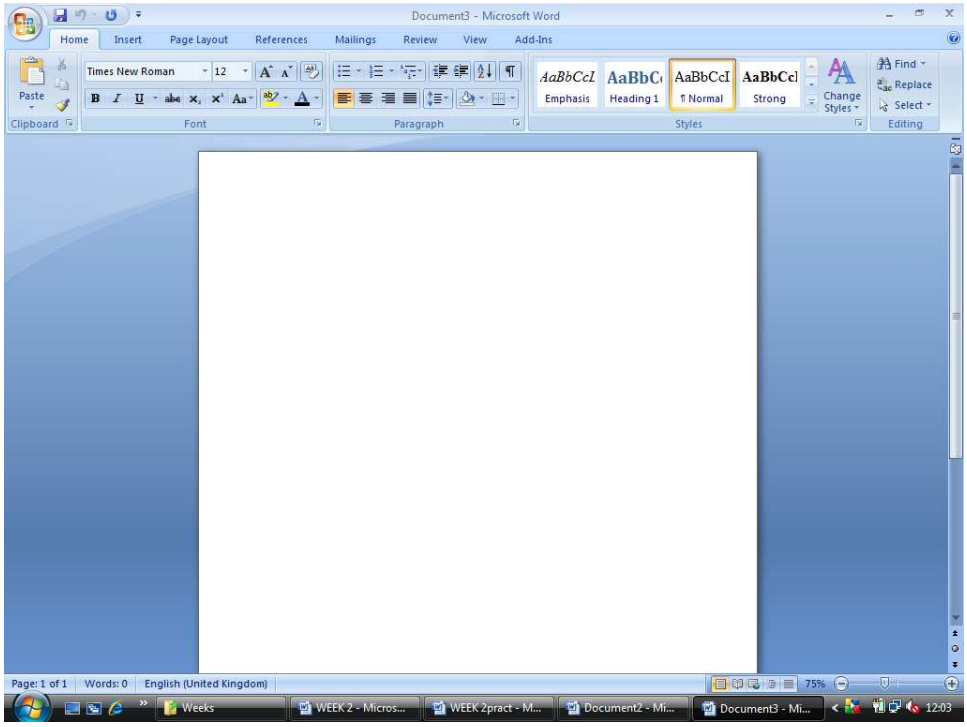
Interaction diagrams

- Communication diagram:
- Interaction overview diagram:
- Sequence diagram:
- Timing diagrams:

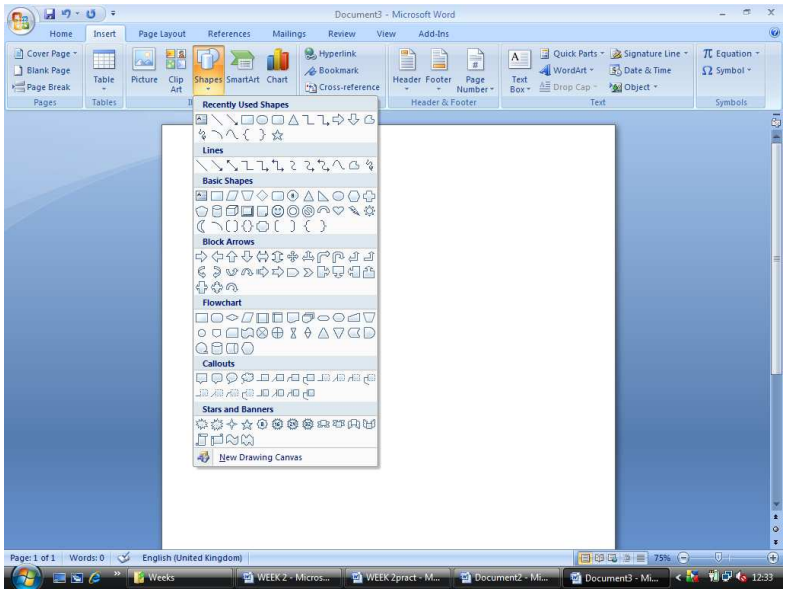
DRAWING OF UML SYMBOLS/NOTATIONS USING WORD PROCESSING APPLICATION (MS WORD)

Procedure

Step 1: Insert Object



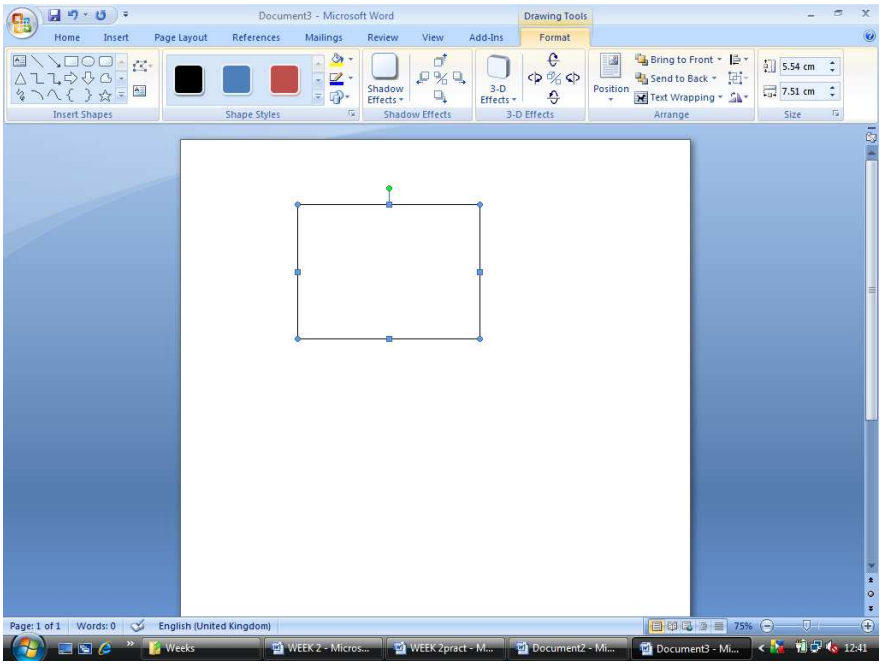
(ii)Click on the Insert Menu to get the basic shapes



(iii) Pick any of the objects and draw using your mouse on the text area of your document.

Step 2: Edit Object
Once an object is inserted it can be edited through the format menu.

Use the step 1 above to insert a rectangular object as below



The Format menu is displayed for all the edit required. For other common operations follow below;

Insert Text

- Right-click the object
- Click add text
- Type the text

Group Object

Objects can be bounded together as one using Grouping properties

- Select all the object using select tool or Shift + mouse click
- Right-Click selection
- Click Grouping
- Click Group

For more on drawing tools with Word 2007, get help from the MS-word 2007 application or from online (www.Microsoft.com)

PRACTICAL EXERCISE

1. Using MS-Word as a tool, draw objects that are related to at least five (5) different fields of specialization.
2. Using the appropriate mathematical notations, derive a formula that can be modelled to a live picture or diagram.

WEEK Three (practical)

Objectives

At the end of the practical week the students should

- Be able to identify the class symbol
- Design a sample class with sample name, attribute and method

Class Diagram

The Class diagram describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes. In dealing with objects and classes the followings are important,

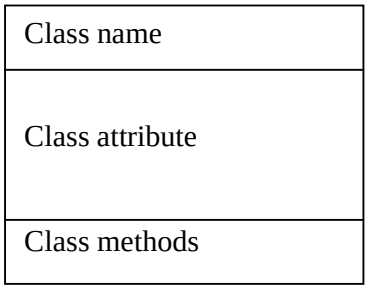
(1) **object and object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).

(2) **attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.

(3) **class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.

Lab 1: Present a sample class diagram symbol

Result:



A sample class diagram

Lab 2: Automobile is a class of Transportation object which can posses several attributes and behaviours. Among those attributes are;

- Model
- Engine No.#
- Color
- Weight
- Registration No
- Etc

The Automobile class can posses many behaviours as well; such as Speeding, Stopping, ignition, etc.

Develop a sample Automobile class named Motor with appropriate attributes and values as well as the automobile methods.

Steps to obtain Results

1. Identify the class name
   Name: Motor

2. Identify the attributes required with data types
   Model No: String
   Engine No: String
   Color: string
   Weight: Real
   Date manufactured: Date
   Chassis No: String
   Amount: Currency

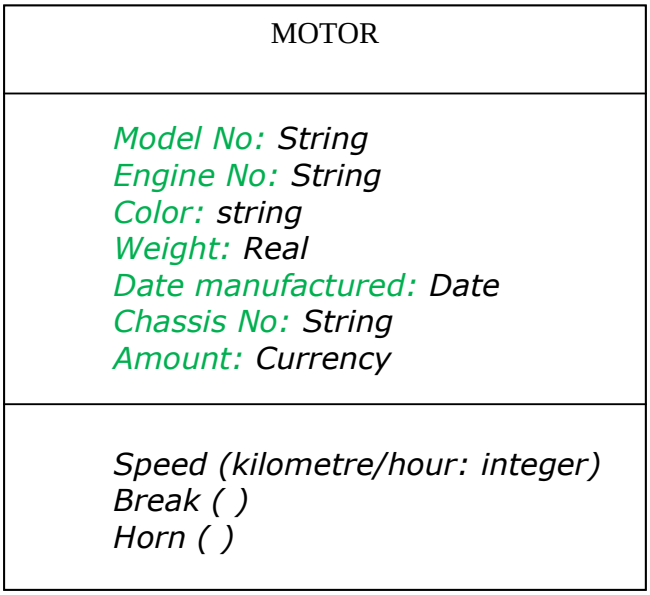
3. Identify the necessary behaviour that it is expected to exhibit
   Speed (kilometre/hour: integer)
   Break ( )
   Horn ( )

4. Use the UML tool to implement the class by feeding in the above information appropriately or use any of the drawing tools such as the MS-word drawing objects as earlier discussed to draw the diagram following the steps below.

- Open MS-word
- Click insert menu
- Click on the object tools and pick the rectangular object and draw appropriately on drawing area.

- Pick line object too to demarcate the rectangular object to form the three compartment required of the class diagram. (In order to duplicate, click on the drawn object, hold the CRTL key and drag the mouse. When you reach the required place release and a duplicate of the selected object is made). Use this to actually make the two lines needed from the first one drawn.
- Add the appropriate text in the rectangular object taking cognisance of the three compartments.
- Group the object together as one

The case is as displayed below.



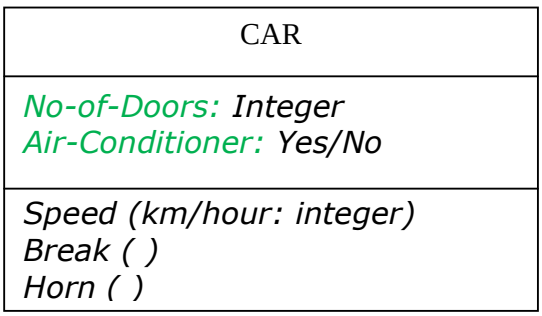
A sample Auto class

INHERITANCE

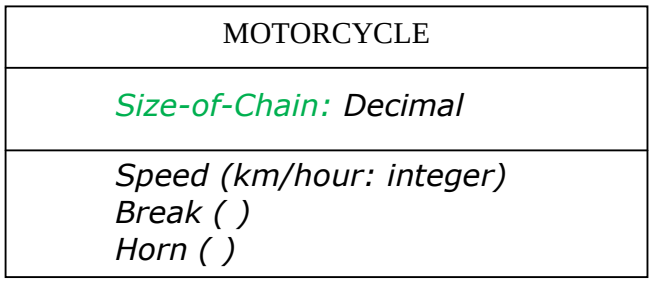
Lab 3: Using the motor class above, create two subclasses (the two subclasses could be CAR subclass and MOTORCYCLE subclass) that can be inherited from the motor class and allow each to add one or more attributes to the general attributes of the Superclass (motor).

Steps to obtain result

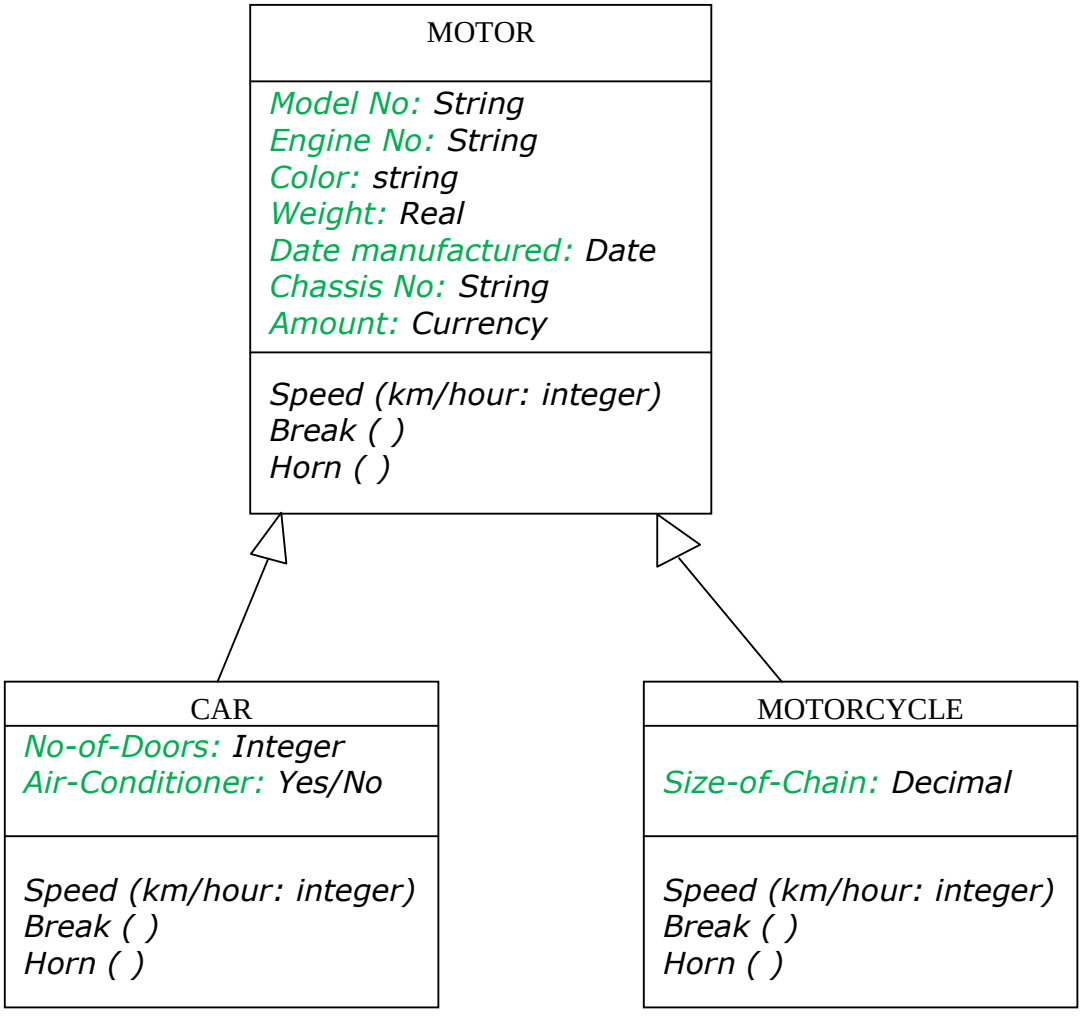
1. Create elements of Car Subclass mentioning any additional functionality such as; No-of-Doors; Air-Conditioner; as below



2. Create elements of MortorCycle subclass mentioning any additional functionality such as; Size-of-Chain as shown below.



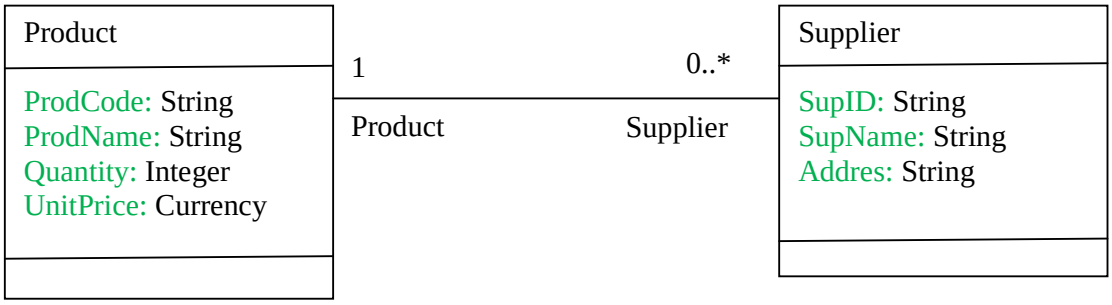
3. Link the two subclasses to their mother class (Motor)



Association

Lab 4: Create a Bi-directional association diagram between products and supplier of products to a company given that for every product there is only one supplier but a supplier can supply more than one product.

Result



Practical Exercises

1. Create a unidirectional association diagram for any real life event

2. Create a basic aggregation relationship diagram to depict any named real life subclass and superclass.

3. Create a composition aggregation relationship diagram to depict any named real life subclass and superclass.

4. Draw a sample reflexive association diagram of a given class

WEEK Four (practical)

Objectives

At the end of the practical week the students should

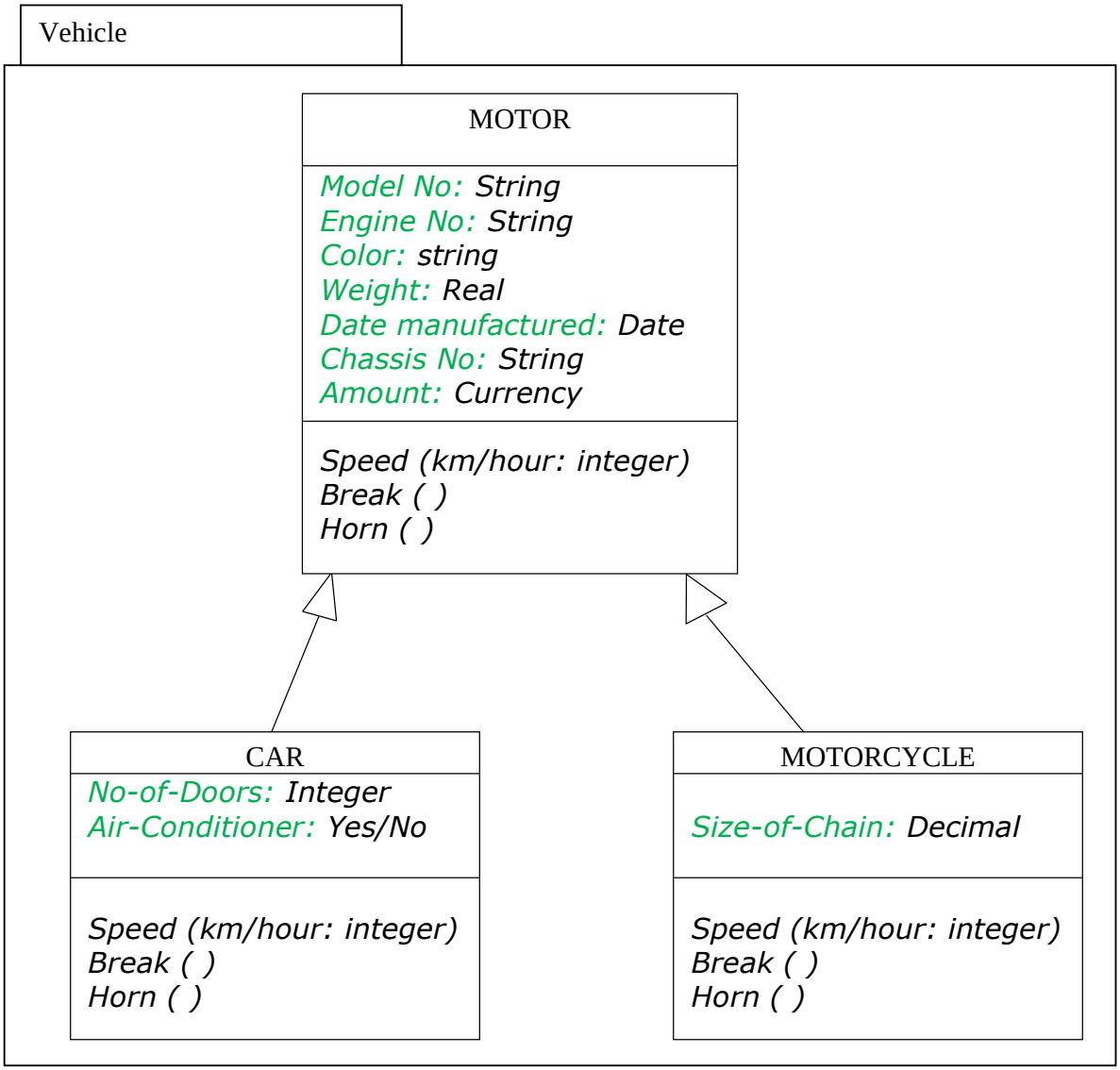
- Be able to model a package using any two methods for a real life application or system
- Design an interface model for at least two classes
- Understand visibility and be able to identify all its marks
- Understand Instance and design instance diagram for a real world scenario
- Model a class role diagrammatically
- Understand class internal structure representation in UML

Package

Think of a package as compartment for grouping classes or similar classes. In a structured format it is in form of a module that house codes capable of maintaining a unit of a project. In a filing system, it can be taught of as a folder that contains similar files and the combination of these folders make up the file structure.

Lab1: Develop a package using any two methods for a given system.

Result



A package element that shows its members inside the package's rectangle boundaries

Interface

An interface is a contract between a class and the outside world.

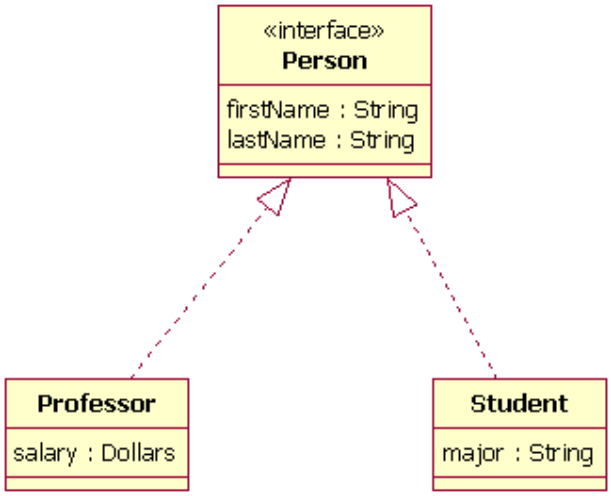
Methods form the object's *interface* with the outside world; for example the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. In the real programming sense the

command button object is interfacing between the user and the actual code which is encapsulated within it. Note that an interface must have at least one class to implement it.

Lab2: Develop an interface for a real life class.

Result



A class diagram in which the Professor and Student classes implement the Person interface

Visibility

In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

Instances

When modelling a system's structure it is sometimes useful to show example instances of the classes. To model this, UML 2 provides the *instance specification* element, which shows interesting information using example (or real) instances in the system.
The notation of an instance is the same as a class, but instead of the top compartment merely having the class's name, the name is an underlined concatenation of Instance Name and Class Name.

Practical Exercises

1. Develop a package element showing its membership via connected lines
2. Develop a class diagram where the four visibility type is implemented for a real life object.
3. Develop a class diagram of Honda car depicting the instance of the particular model.
4. Design a model that display the internal structure of a real life system such as a named Polytechnic.

WEEK Five (practical)

Objectives

At the end of the practical week the students should

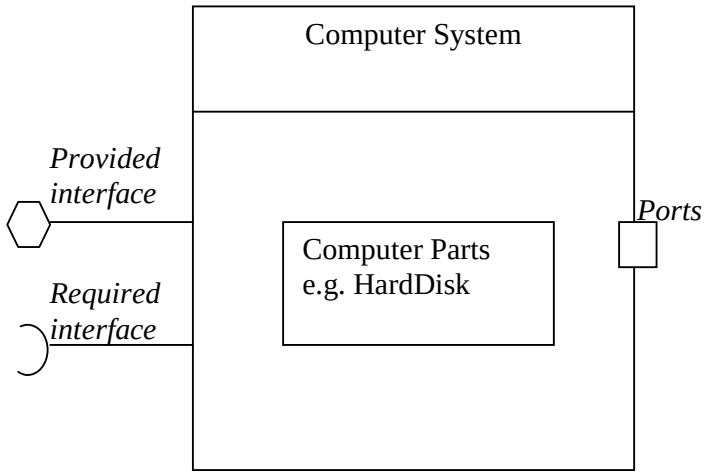
- Be able to design a composite structure of a named classifier.
- Understand the functional component of a composite diagram

COMPOSITE DIAGRAM

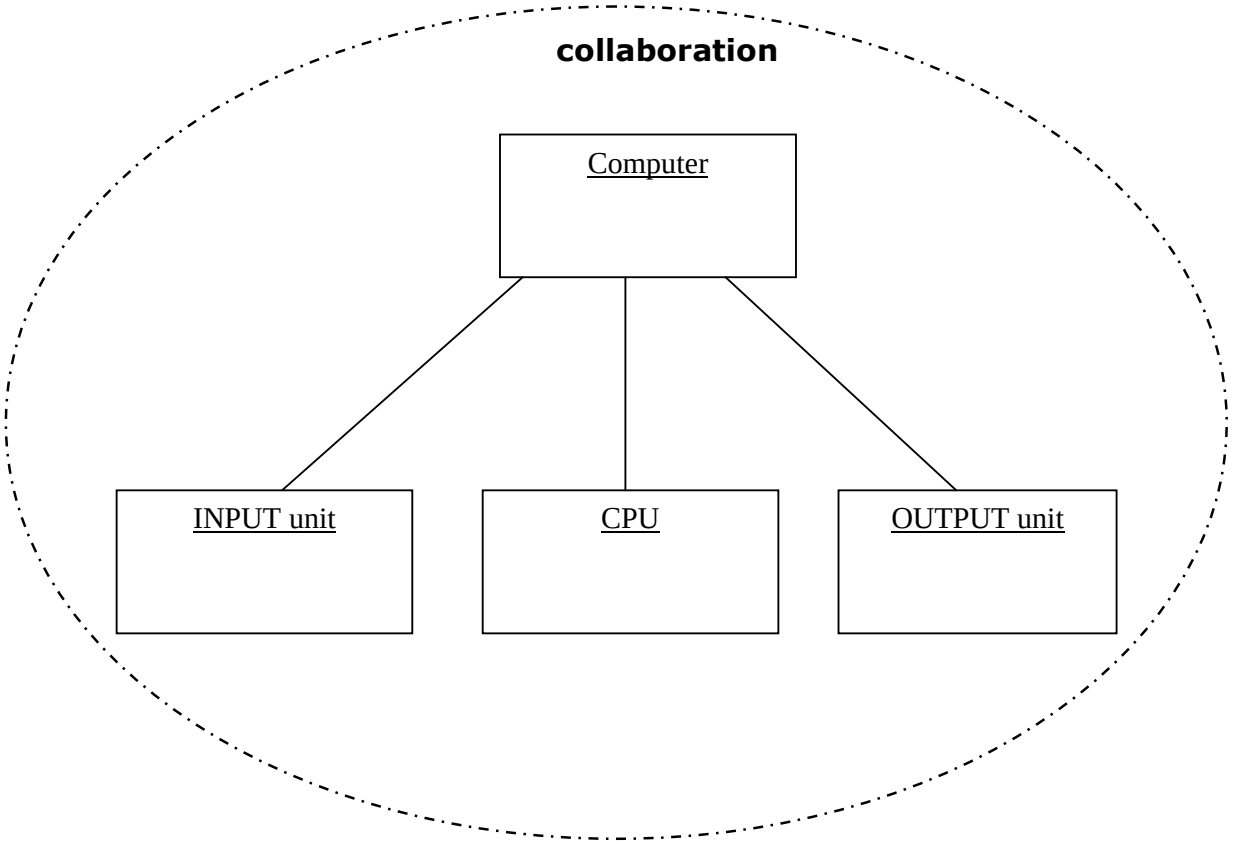
A composite structure diagram is a diagram that shows the internal structure of a classifier, including its interaction points to other parts of the system. It shows the configuration and relationship of parts, that together, perform the behavior of the containing classifier.

Lab1: Develop a sample composite diagram of computer system class.

Results



Lab2: Design a collaboration system composing a Computer and some mentioned units such as Input, CPU and Output units.



Practical Exercises

Create a Role Binding, Represents, and Occurrence connector scenario of the collaboration model above.

WEEK Six (practical)

Objectives

At the end of the practical week the students should

- Understand the notation for a component diagram
- Use the notations to model a real life system
  Be able to design a composite structure of a named classifier.
- Understand the functional component of a composite diagram

COMPONENT DIAGRAM

The component diagram's main purpose is to show the structural relationships between the components of a system. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building, which helps developers begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed skill enhancements. System administrators find component diagrams useful because they get an early view of the logical software components that will be running on their systems. Although system administrators will not be able to identify the physical machines or the physical executables from the diagram, a component diagram will nevertheless be welcomed because it provides early information about the components and their relationships (which allows sys-admins to loosely plan ahead).

COMPONENT NOTATION

In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modelled as just a rectangle with the component's name and the component stereotype text and/or icon. The component stereotype's text is «component»" and the component stereotype icon is a rectangle with two smaller rectangles protruding on its left side (the UML 1.4 notation element for a component). Figure 5.2 shows three different ways a component can be drawn using the UML 2 specification.

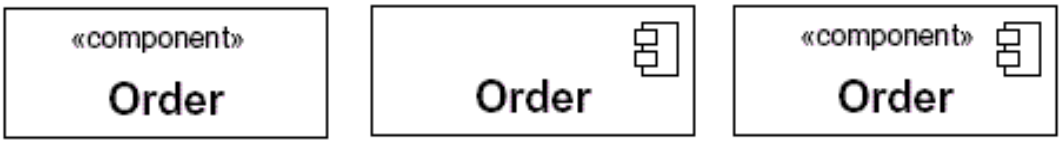
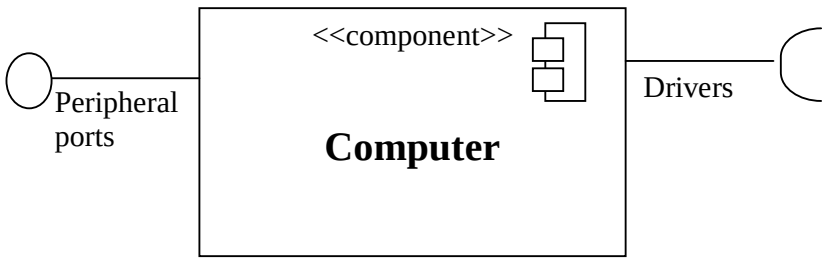


Figure 2: The different ways to draw a component's name compartment

When drawing a component on a diagram, it is important that you always include the component stereotype text (the word "component" inside double angle brackets, as shown in Figure 5.2) and/or icon. The reason? In UML, a rectangle without any stereotype classifier is interpreted as a class element. The component stereotype and/or icon distinguishes this rectangle as a component element.

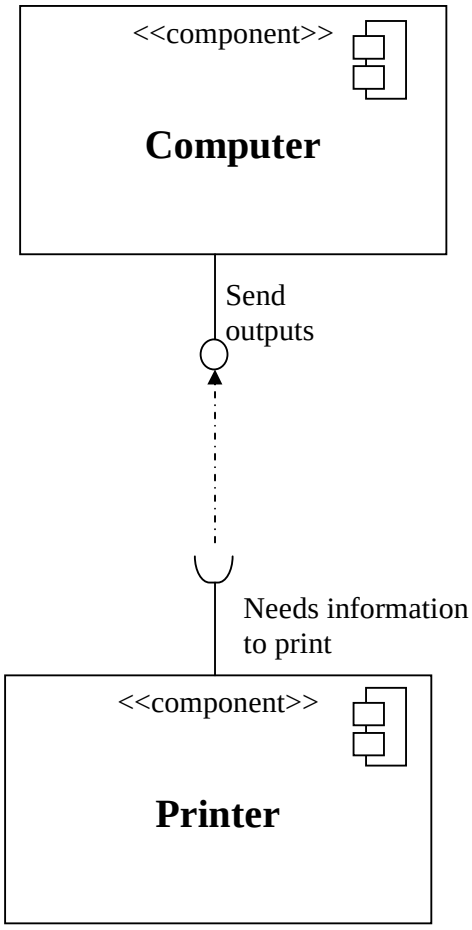
Lab1:Design a component model of a computer system showing the interfaces that the object provides and required.

Result



Lab2: Model a component relationship diagram of a printer requiring data from a computer system. Design a component model of a computer system showing the interfaces that the object provides and required.

Result



Practical Exercises

1. Develop a subsystem classifier for any subunit of the computer system.
2. Model a component relationship of a real life scenario

WEEK Seven (practical)

Objectives

At the end of the practical week the students should

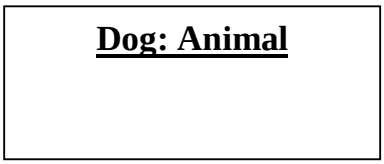
- Be able to identify Object Diagram and differentiate it from class diagram
- Be able to identify and design a deployment model

OBJECT DIAGRAM

An object diagram may be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. Object elements don't have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.

Lab1:Design a sample object diagram of a named object.

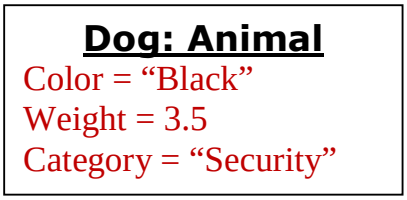
Result



A typical dog object diagram. Note that the Dog object is an instance of an animal Class

Lab1: Develop the dog object in lab1 above with some run time property values

Result



PACKAGE DIAGRAMS

Package diagrams are used to reflect the organization of packages and their elements. Elements contained in a package share the same namespace. Therefore, the elements contained in a specific namespace must have unique names.

Practical Exercises

3. Develop an object diagram for any chosen object
4. Create an object from a class and develop both the class diagram and the object diagram for them
5. Create a package model for a complete system

WEEK Eight (practical)

Objectives

At the end of the practical week the students should

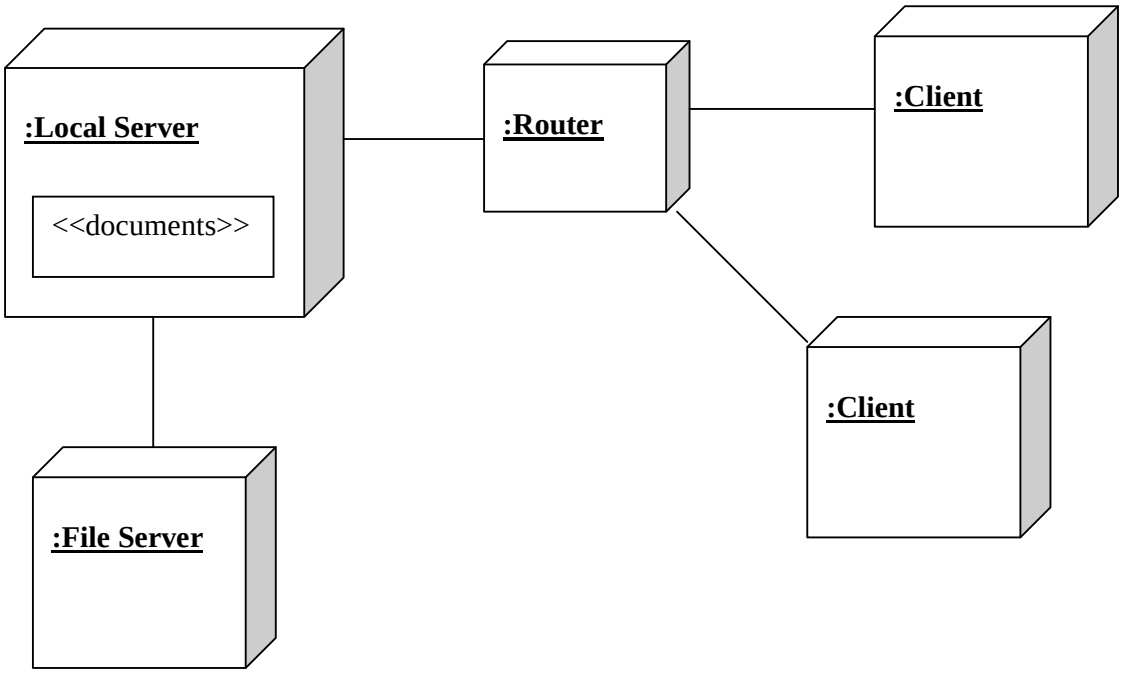
- Understand and be able to identify Nodes in a deployment model
- Create association amongst nodes
- Model a typical life projects depicting the existing nodes and show the contained elements.

Deployment Diagrams

A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

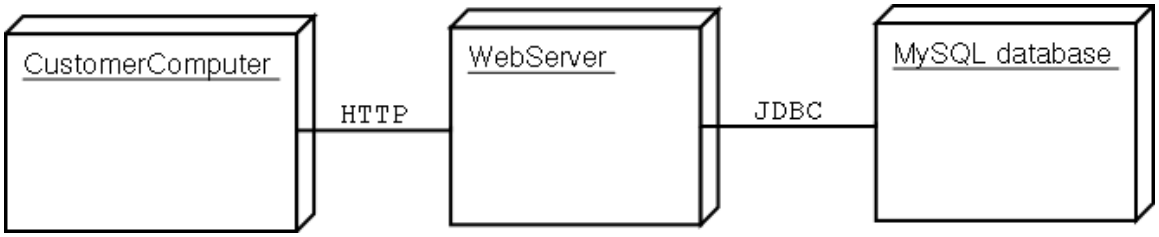
Lab1: Model a deployment diagram for a typical small office Local Area Network (LAN) setup via a Router.

Result



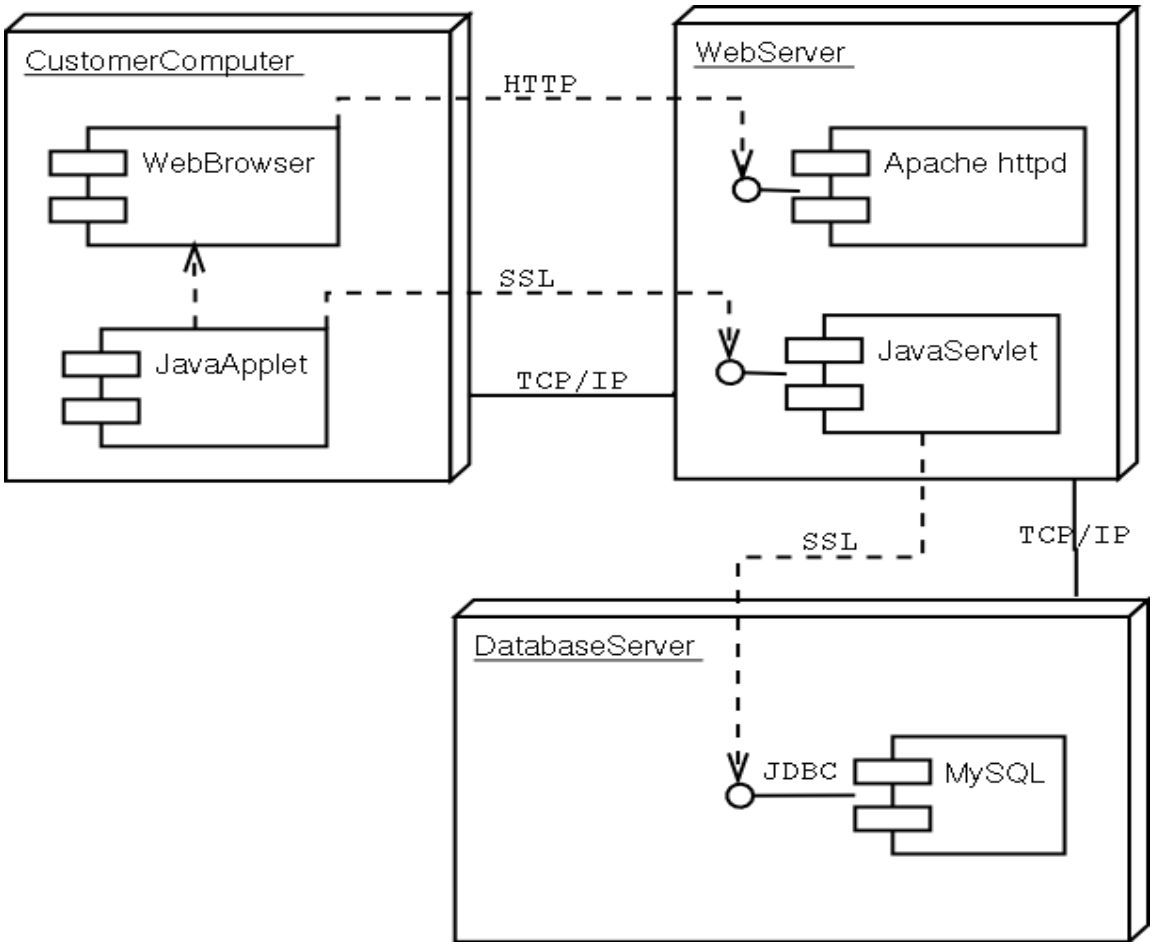
Lab2: Design a deployment diagram that captures the distinct number of computers required for a typical computer communication setup.

Result



Lab3: Embed a component diagram into the deployment model above

Result



WEEK Nine (practical)

Objectives

At the end of the practical week the students should

- Understand the concept of activity and be able to identify activity diagram notations
- Create a simple activity diagram for a real life programming algorithm
- Model a typical life projects or system using the activity diagram.

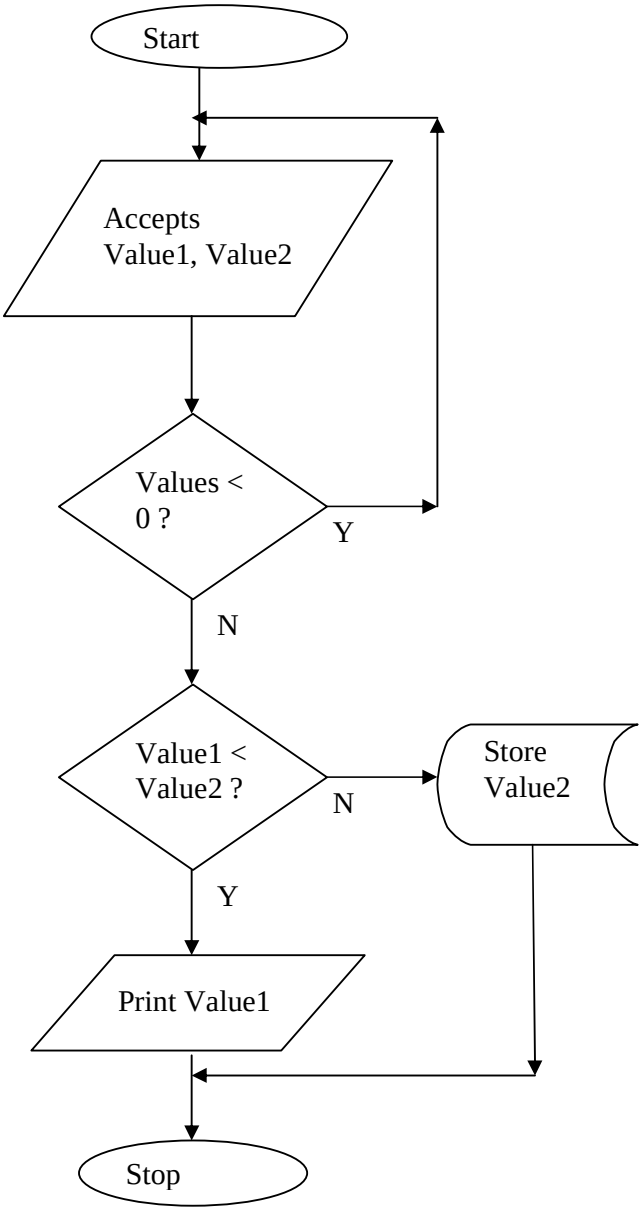
Activity Diagrams

Activity is defined as an ongoing non-atomic execution within a state machine. A structure that has a duration with possible interruption points. Activities can be modelled by nested states, by a submachine reference, or by an activity expression.
Activity state represents the execution of a pure computation with substructure. The normal use of an activity state is to model a step in the execution of a procedure or algorithm.

An activity can represent a fairly large procedure (with some substructure), as well as something relatively small.

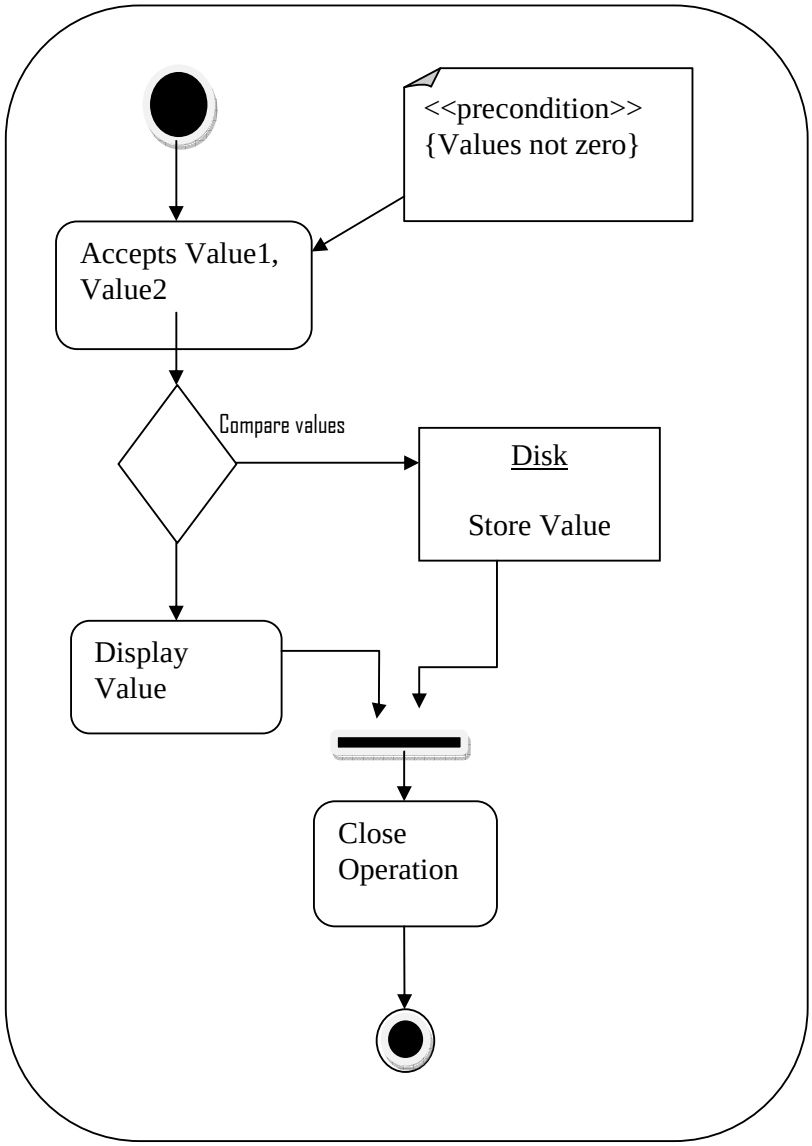
Lab1: Draw a flowchart for a program that accepts two none negative values, compare them, display the smaller and store the bigger.

Result:


Start
Accepts
Value1, Value2
Values <
0 ?
Y
N
Value1 <
Value2 ?
N
Store
Value2
Y
Print Value1
Stop

Lab2: Draw an activity diagram for the program represented in the above flowchart.

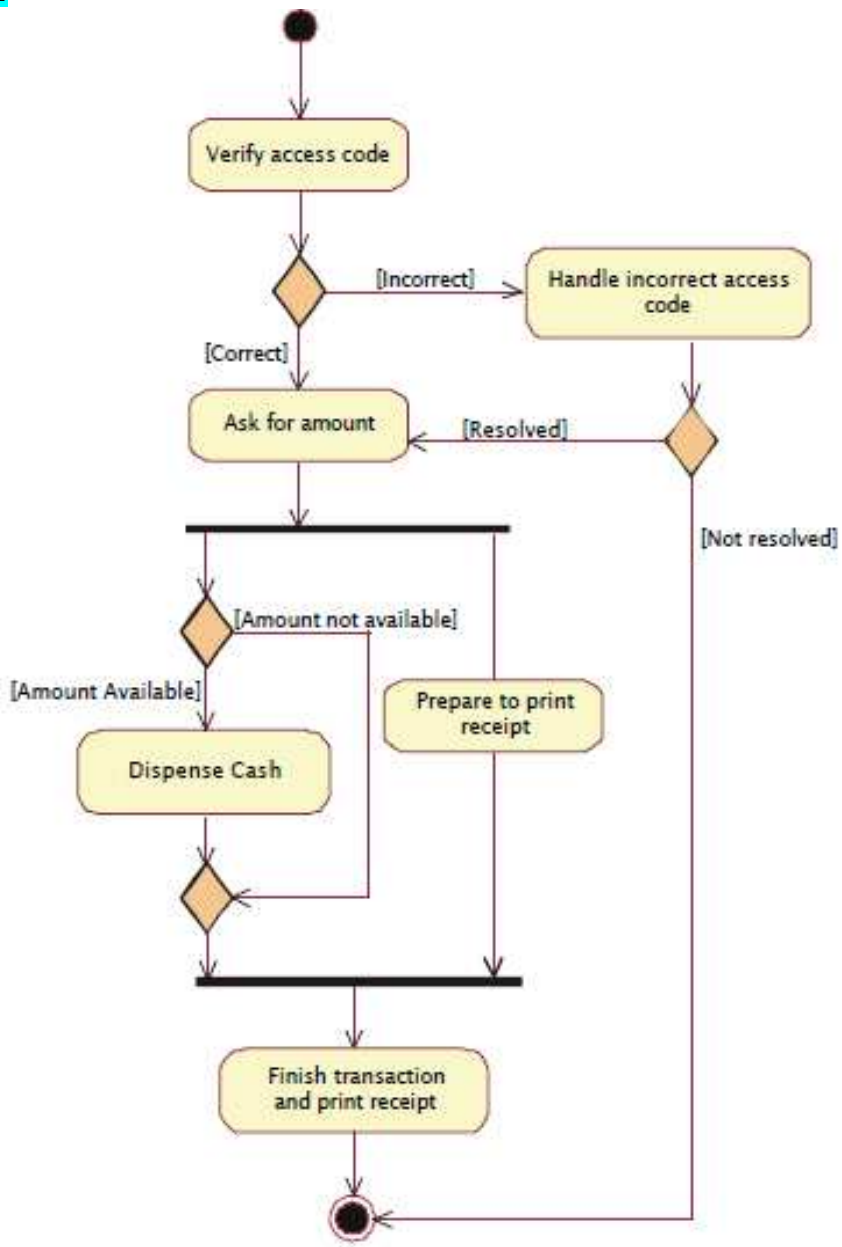
Result:



An activity diagram for a program flow

Lab2: Develop an activity diagram for an operation of ATM machine.

Result:



EXERCISE

1. Develop and activity diagram for a simple program flow
2. Develop and activity diagram for any real life event

WEEK Ten (practical)

Objectives

At the end of the practical week the students should

- Understand the meaning of use case and its notations
- Model a real life activity with a Use case diagram

Use Cases

A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system. UML Use Case Diagrams can be used to describe the functionality of a system in a horizontal way. That is, rather than merely representing the details of individual features of your system, UCDs can be used to show all of its available functionality. It is important to note, though, that UCDs are fundamentally different from sequence diagrams or flow charts because they do not make any attempt to represent the order or number of times that the systems actions and sub-actions should be executed.

UCDs have only 4 major elements: The **actors** that the system you are describing interacts with, the **system** itself, the **use cases**, or services, that the system knows how to perform, and the lines that represent **relationships** between these elements.

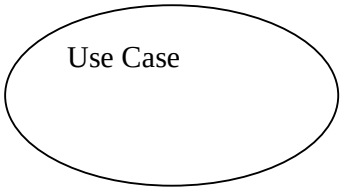
You should use UCDs to represent the functionality of your system from a top-down perspective (that is, at a glance the system's functionality is obvious, but all descriptions are at a very high level. Further detail can later be added to the diagram to elucidate interesting points in the system's behavior.)

Example: A UCD is well suited to the task of describing all of the things that can be done with a database system, by all of the people who might use it (administrators, developers, data entry personnel.)

You should NOT use UCDs to represent exception behavior (when errors happen) or to try to illustrate the sequence of steps that must be performed in order to complete a task. Use Sequence diagrams to show these design features.

Example: A UCD would be poorly suited to describing the TCP/IP network protocol, because there are many exception cases, branching behaviors, and conditional functionality (what happens when a packet is lost or late, what about when the connection

dies?) The notation for a use case is an ellipse. Use cases may contain the functionality of another use case as part of their normal processing. In addition one use case may be used to extend the behavior of another; this is typically used in exceptional circumstances.



How do you know who the actors are in a UCD?

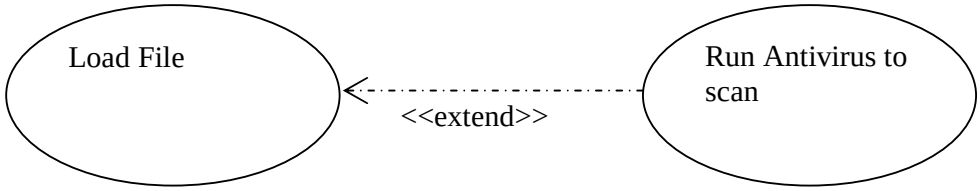
When working from an Action/Response table, itentifying the actors is easy: entities whose behavior appears in the "Actor's Actions" column are the actors, and entities whose behavior appears in the "System's Response" column are components in the system.

If you are working from an informal narrative, a sequence diagram, or a scenario description, the actors are typically those entities whose behavior cannot control or change (i.e., agents that are not part of the system that you are building or describing.) The most obvious candidates for actors are the humans in the system; except in rare cases when the system you are describing is actually a human process (such as a specific method of dealing with customers that employees should follow) the humans that you must interact with will all be actors. If your system interacts with other systems (databases, servers maintained by other people, legacy systems) you will be best to treat these as actors, also, since it is not their behavior that you are interested in describing.

Example: When adding a new database system to manage a company's finances, your system will probably have to interface with their existing inventory management software. Since you didn't write this software, don't intend to replace it, and only use the services that it provides, it makes sense for that system to be an **actor**.

Lab1: Draw an extended use case diagram for Loading a document from a flash disk that require scanning for virus.

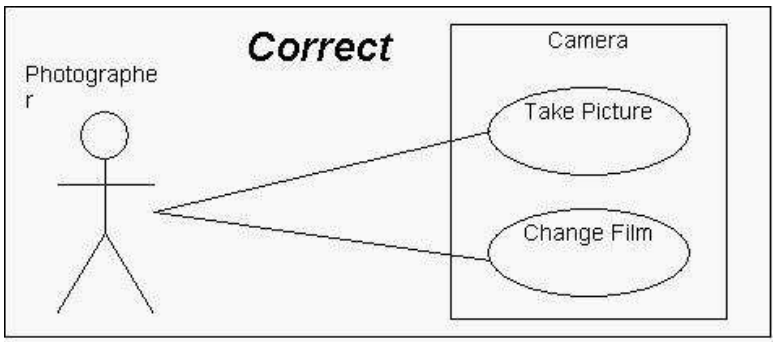
Result:

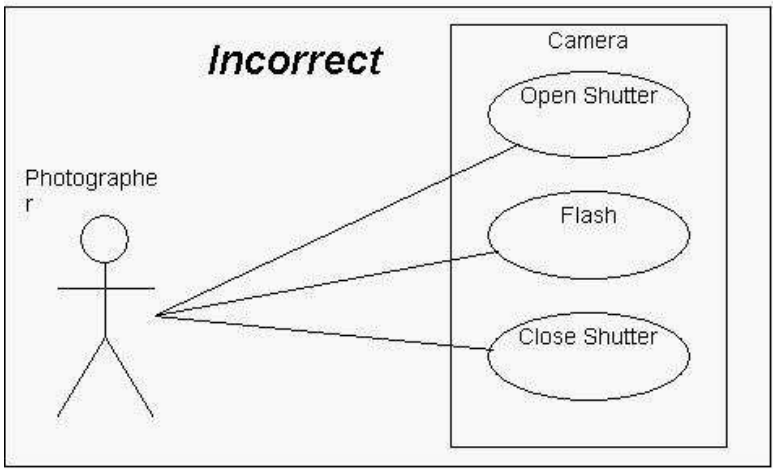


Lab2: Present the use cases for a camera. Suppose we choose "Open Shutter", "Flash", and "Close Shutter" as the top-level use cases.

Note: Certainly these are all behaviors that a camera has, but no photographer would ever pick up their camera, open the shutter, and then put it down, satisfied with their photographic session for the day. The crucial thing to realize is that these behaviors are not done in isolation, but are rather a **part** of a more high-level use case, "Take Picture". (Note as well that it does make sense for a photographer to "Take Picture" just once during a session with their camera.)

Results





EXERCISE

1. Develop an included use case diagram for the lab 1 above
2. Design a comprehensive use case that embed both include and extend functionalities.

WEEK Eleven (practical)

Objectives

At the end of the practical week the students should

- Understand the meaning of state machines
- Model a real life machine operation using a state machine diagram

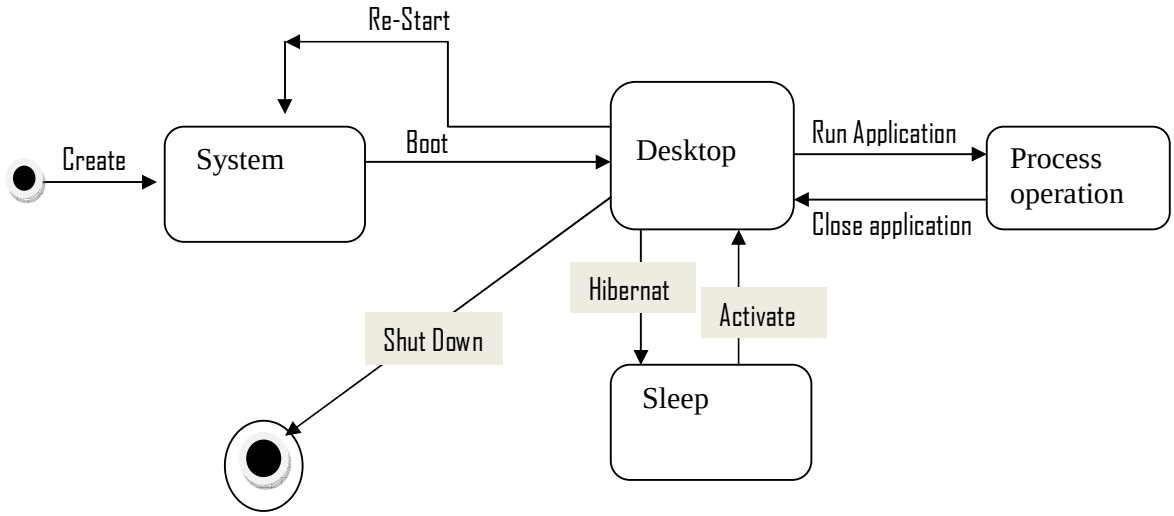
STATE DIAGRAM

State diagrams are used to describe the behavior of a system. State diagrams can describe the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

State diagram can be used to graphically represent finite state machines.

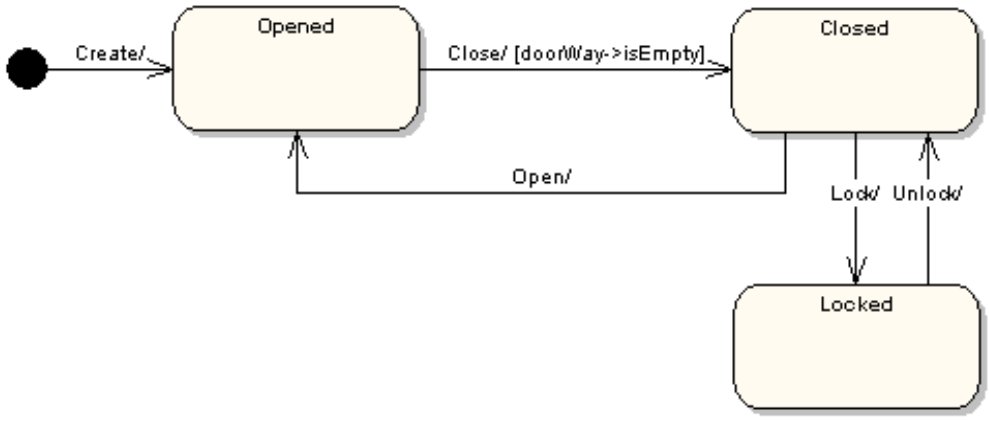
Lab1: Create a state machine diagram for a computer system from rest to processing state and shutting down.

Result:



Lab2: Create a state machine diagram showing the states that a door goes through during its lifetime.

Result:



EXERCISE

1. Model a real life named machine operation using a state machine diagram
2. Create a compound state machine diagram to model a complex system containing sub systems.

WEEK Twelve (practical)

Objectives

At the end of the practical week the students should

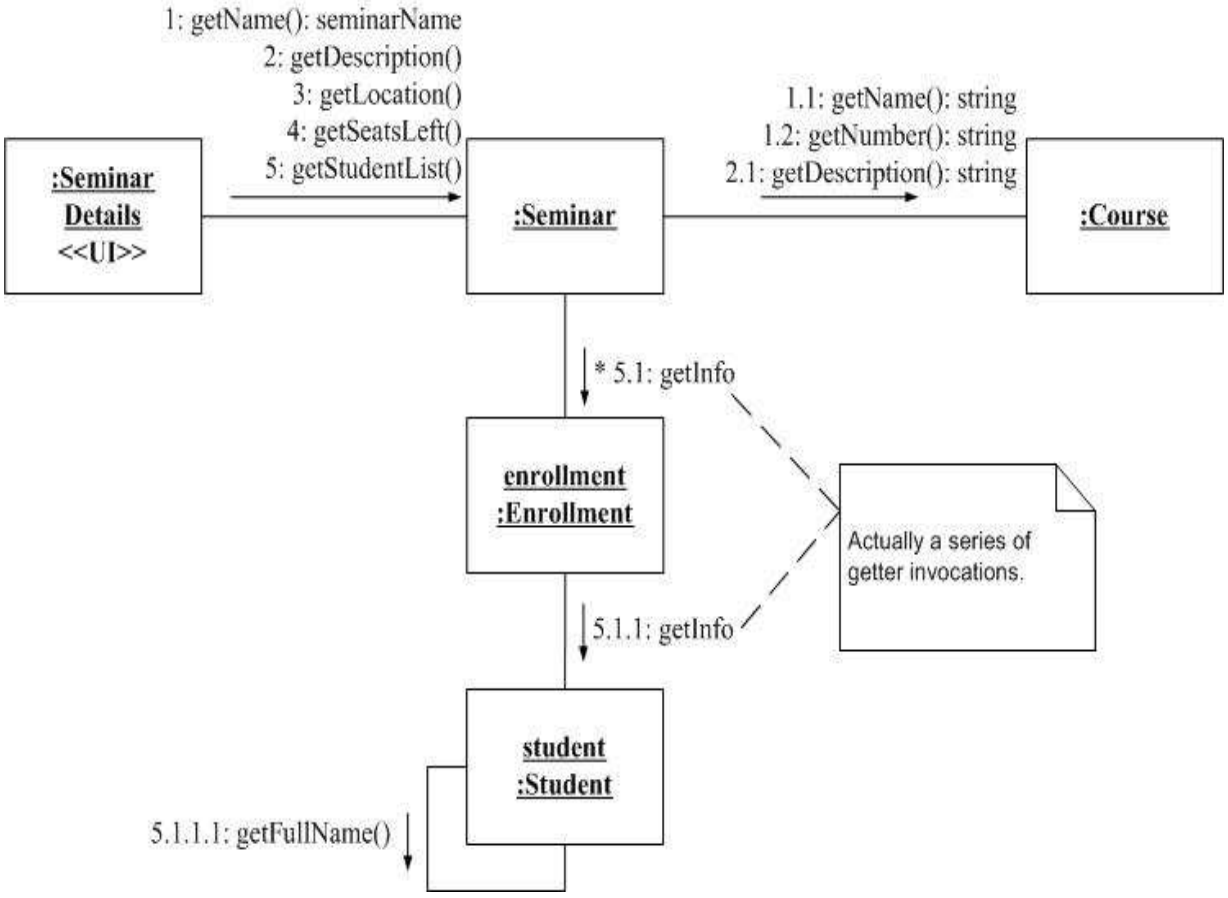
- Be able to create communication link between events and objects
- Understand the uses of sequence diagram and be able to apply to a real life system.

COMMUNICATION MODEL

Communication diagrams show the message flow between objects in an OO application and also imply the basic associations (relationships) between classes.

Lab 1: Develop a simplified collaboration diagram for displaying a seminar details screen or page.

Result:



Explanation

The rectangles represent the various objects involved that make up the application. The lines between the classes represent the relationships (associations, composition, dependencies, or inheritance) between them. The same notation for classes and objects used on UML sequence diagrams are used on UML communication diagrams, another example of the consistency of the UML. The details of your associations, such as their multiplicities, are not modelled because this information is contained on your UML class diagrams: remember, each UML diagram has its own specific purpose and no single diagram is sufficient on its own. Messages are depicted as a labelled arrow that indicates the direction of the message, using a notation similar to that used on sequence diagrams.

SEQUENCE DIAGRAM

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. Much like the class diagram, developers typically think sequence diagrams were meant exclusively for them. However, an organization's business staff can find sequence diagrams useful to communicate how the business currently works by showing how various business objects interact. Besides documenting an organization's current affairs, a business-level sequence diagram can be used as a requirements document to communicate requirements for a future system implementation. During the requirements phase of a project, analysts can take use cases to the next level by providing a more formal level of refinement. When that occurs, use cases are often refined into one or more sequence diagrams.

An organization's technical staff can find sequence diagrams useful in documenting how a future system should behave. During the design phase, architects and developers can use the diagram to force out the system's object interactions, thus fleshing out overall system design.

One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or

more sequence diagrams. In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an existing (call it "legacy") system currently interact. This documentation is very useful when transitioning a system to another person or organization.

EXERCISE

1. Model a typical communication system for a local area network using a router.
2. Create a sequence diagram for the above.

WEEK Thirteen (practical)

Objectives

At the end of the practical week the students should

- Understand and be able to create an overview diagram for a life event.
- Understand the uses of Timing diagram.

INTERACTION OVERVIEW DIAGRAM

TIMING DIAGRAM

UML timing diagrams are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.

Timing diagrams depict the changes in state, or condition, of one or more interacting objects over a given period of time. States, or conditions, are displayed as timelines responding to message events, where a lifeline represents a Classifier Instance or Classifier Role.

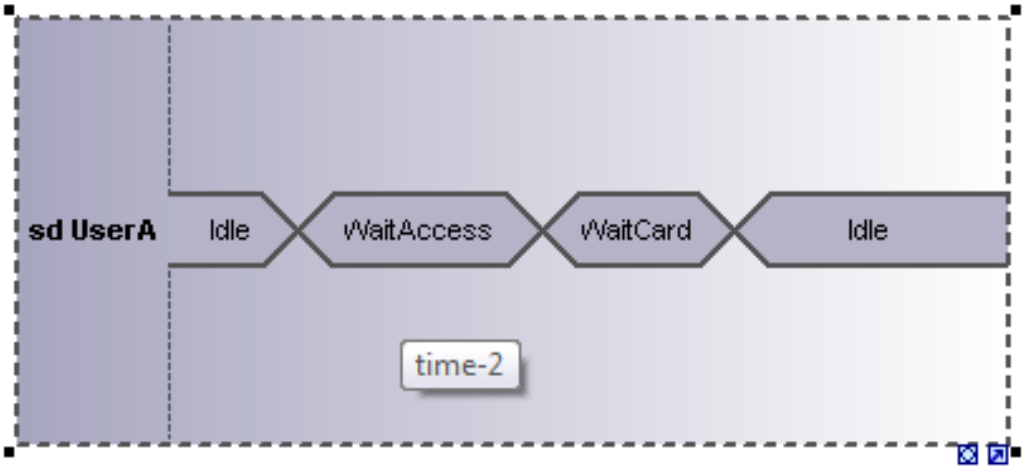
A Timing diagram is a special form of a sequence diagram. The difference is that the axes are reversed i.e. time increases from left to right, and lifelines are shown in separate vertically stacked compartments.

Timing diagrams are generally used when designing embedded software or real-time systems.

TYPES OF TIMING DIAGRAM

There are two different types of timing diagram: one containing the State/Condition timeline as shown above, and the other, the

General value lifeline, shown below


sd UserA
Idle
WaitAccess
WaitCard
Idle
time-2

WEEK Fourteen (practical)

Objectives

At the end of the practical week the students should

- Understand the basis of using software tools for UML.
- Understand the basic requirement of any standard UML tools.

The UML Tool

A UML tool or UML modeling tool is a software application that supports some or all of the notation and semantics associated with the Unified Modeling Language (UML), which is the industry standard general purpose modeling language for software engineering.

UML tool is used broadly here to include application programs which are not exclusively focused on UML, but which support some functions of the Unified Modeling Language, either as an *add-on*, as a *component* or as a *part* of their overall functionality.

Practical exercise

1. Research on at least two UML modelling tool and compare their functionalities, interface, simplicity, acquisition, and their requirements.
2. Adopt an UML tool and state it relevance to the computer society

WEEK Fifteen (practical)

Objectives

At the end of the practical week the students should

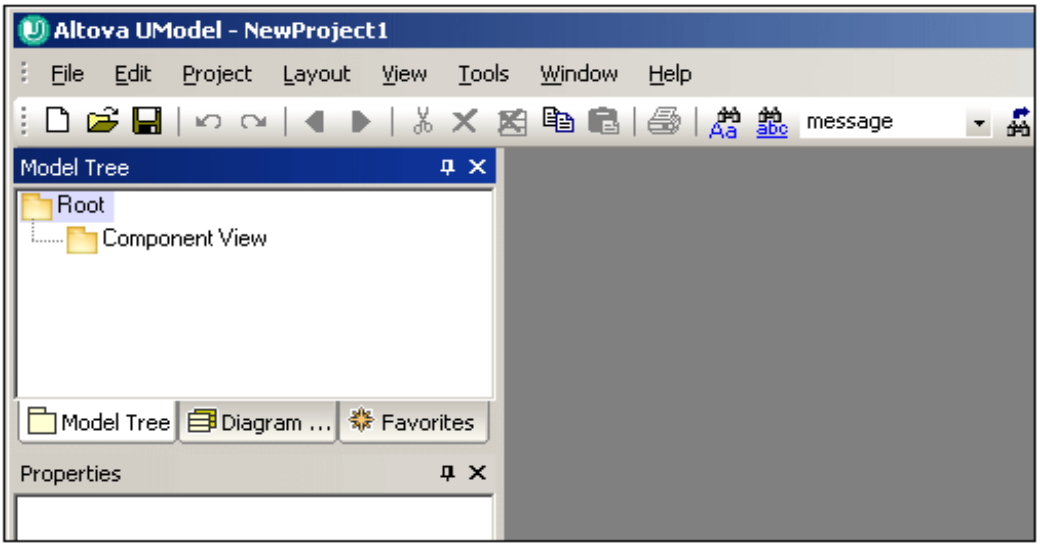
- Understand the installation and usage of Altova UModel 2008 development tool.

Starting UModel

Having installed UModel on your computer:

Start UModel by double-clicking the UModel icon on your desktop, or use the **Start | All Programs** menu to access the UModel program.

UModel is started with a default project "NewProject1" visible in the interface.

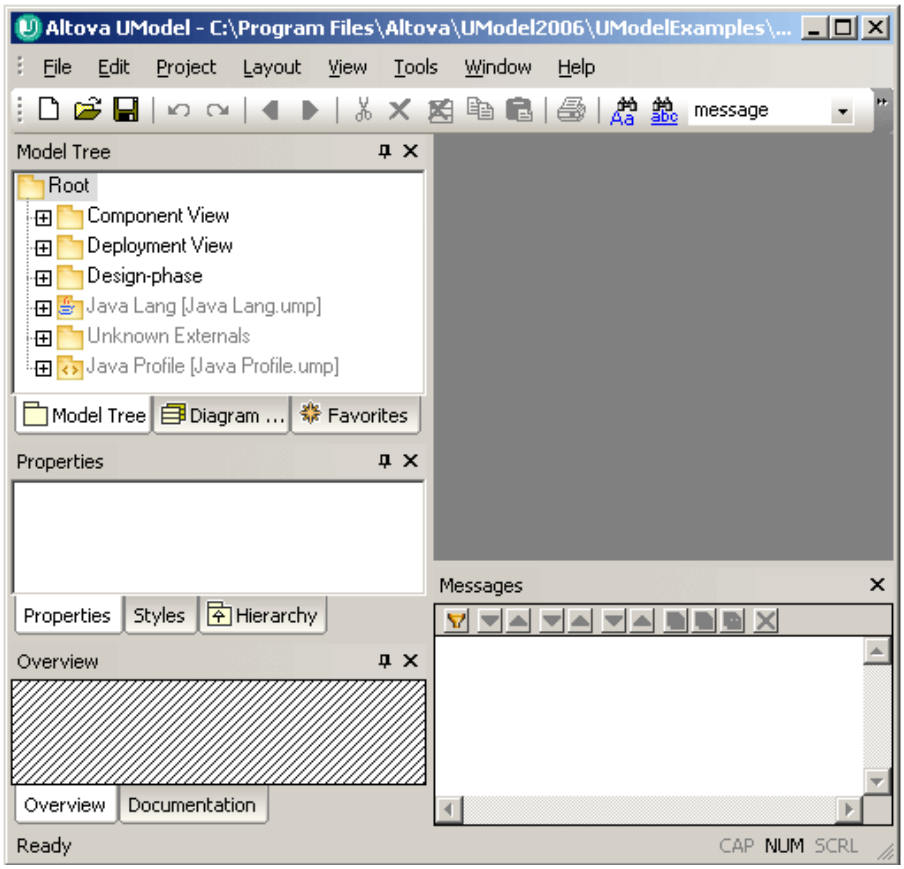


Note the major parts of the user interface: the three panes on the left hand side and the empty diagram pane at right.

Two default packages are visible in the Model Tree tab, "Root" and "Component View". These two packages cannot be deleted or renamed in a project.

To open the BankView-start project:

1. Select the menu option **File | Open** and navigate to the ...**UModelExamples** folder of UModel.
2. Open the **BankView-start.ump** project file.
   The project file is now loaded into UModel. Several predefined packages are now visible under the Root package



The Model Tree pane
supplies you with various views of your modeling project:

The **Model Tree** tab contains and displays all modeling elements of your UModel project. Elements can be directly manipulated in this tab using the standard editing keys as well as drag and drop.

.

The **Diagram Tree** tab allows you quick access to the modeling diagrams of you project wherever they may be in the project structure. Diagrams are grouped according to their diagram

· type.

· The **Favorites** tab is a user-definable repository of modeling

elements. Any type of modeling element can be placed in this tab using the "Add to Favorites" command of the context menu.

The Properties pane
supplies you with two views of specific model properties:

- The **Properties** tab displays the properties of the currently selected element in the Model Tree pane or in the Diagram tab. Element properties can defined or updated in this tab.
- The **Styles** tab displays attributes of diagrams, or elements that are displayed in the Diagram view. These style attributes fall into two general groups: Formatting and display settings.

The Overview pane
displays two tabs:

- The **Overview** tab, which displays an outline view of the currently active diagram
- The **Documentation** tab which allows you to document your classes on a per-class basis.