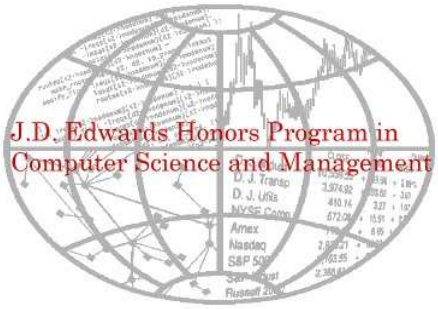


UNIVERSITY OF NEBRASKA–LINCOLN
J.D. EDWARDS HONORS PROGRAM

Short UML Reference

Kalyan Ram – Sebastian Elbaum
Version 1.5

August, 2002

Section 1 - Introduction

Software analysis helps to understand the **vocabulary of the problem domain** from the customer perspective and **set the system's goals**. Software design is concerned with the generation of a **precise abstraction of what the "desired" system should do and a high level strategy for attacking the problem.**

The **Unified Modeling Language** is a **graphical notation** for capturing requirements and expressing the design of a software system, providing a means for developing blueprints of a software system. UML assists software architects in getting the "**big-picture**" of a system by providing a balance between natural language (which is too imprecise) and code (which is too detailed). UML is also the industry-standard modeling language.

Ref:1

Section 2 - Use Cases

Use case modeling is an approach to capture user requirements. A use-case is a **sequence of interactions between the user and the system under consideration to achieve a goal**. A user initiates a use-case with a particular goal in mind, and completes the use-case when the service satisfies that particular goal. The system is treated as a "black box" where the interactions with the system are perceived from outside the system. A **scenario** is an instance of a use-case, representing a single variation of a use case (e.g., triggered by options, error conditions, security breaches, etc.). For each use-case you need to write a detailed description including: use-case name and goal, actor(s), preconditions, main flow of events, alternate flows and post conditions[1].

Example 2.1:

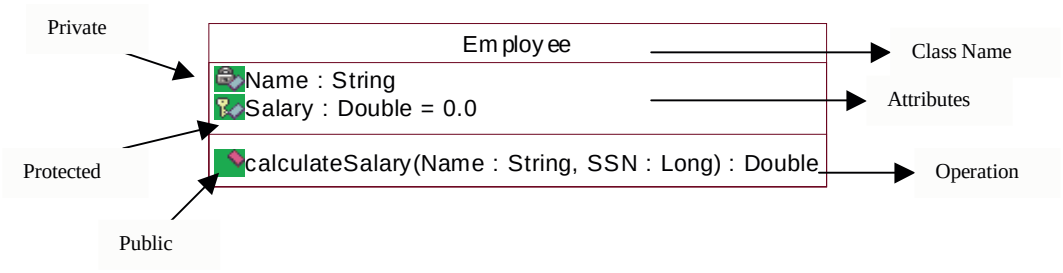
Use case name	ATM-W-01
Goal	ATM Money Withdraw
Precondition	User has inserted a card and has been authenticated
Flow of events	1. ATM asks the user to enter the amount to be withdrawn. 2. User enters the amount needed from the account. 3. ATM authorizes the amount entered by verifying the balance. 4. ATM delivers the amount through the machine slot. 5. ATM delivers the transaction receipt. 6. ATM ejects the card from the slot
Post condition	System in baseline state to receive and initiate another transaction
Alternative scenarios	Insufficient balance At Step3, a) ATM declares insufficient balance in the account. b) ATM suspends the transaction

[1] Graphical representation of use-cases is possible but will not be covered in this document.

Section 3 - Class Diagrams

A class diagram is a **graphical-notation** for depicting the system's classes along with their relationship to one another. A class diagram models the overall structure of classes that make up the system architecture. The fundamental component of a class diagram is a "class". **A class is an abstraction of entities with common characteristics,** and it has three basic components: a class name, attributes and operations.

Example 3.1: Class Structure



Attributes:

An attribute is a property of the object under consideration. For example, "John" and "24" are the name and age attributes of the class Person.

Syntax for an attribute: *visibility name: data-type = default value assigned*

Visibility:

Visibility is expressed in terms of three modifiers namely**: private, public and protected**. Private variables or operations are accessible from only within the same class. Public variables or operations are accessible from anywhere. Protected variables or operations of a class are accessible from the same classes within the package or from the sub-classes to the class under consideration. Protected modifiers give restricted access as compared to the public modifier.

Syntax for visibility is: *+ for public, # for protected and - for private.*

Note: Syntax for visibility modifiers in example 3.1 follows Rational Rose.

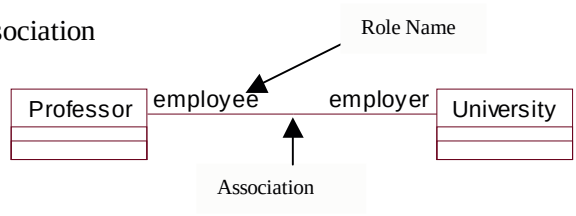
Operations:

The functionality assigned to an object is realized with the help of **operations**. For example, a circle object can have an operation like "calculate the area of the circle". A method is a way of implementing a particular operation. An operation can be implemented by different methods in different ways. For example, let us consider an operation like add two numbers. One method for implementing this operation could be adding two integer numbers and another method could be adding two real numbers.

Syntax for operation is: *visibility methodName (parameter-list): return-type*

Classes relate to each other through different forms of association. In general, an association depicts the relationship between instances of classes. The association between two classes consists of two **association ends** and each association end can be labeled explicitly with a **role name**. We can refine the relationship between classes through the use of multiplicity, navigability, generalization, aggregation, and constraint.

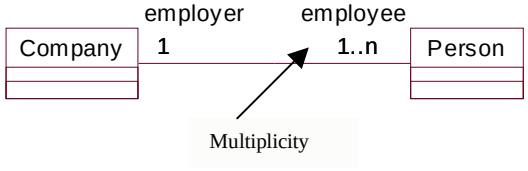
Example 3.2: Association



Multiplicity:

Multiplicity indicates how many instances of a class may relate to a single instance of an associated class. For example, a company has one or more employees, but an employee works for only one company.

Example 3.3: Multiplicity

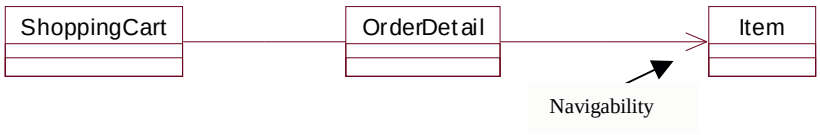


Ref:2

Navigability:

Navigability consists of "directed" association. If the direction vector points from class A to class B, then it means that class A has visibility or access to class B. If the navigability between two classes is non-commutative, then it is called as **unidirectional** association. If the navigability between two classes is commutative, then it is called as a **bi-directional a**ssociation.

Example 3.4: Navigability: OrderDetail class has access to the Item class because OrderDetail has many Items but Item does not have access to what is there in the Order placed. Therefore, the relationship between OrderDetail and Item is Unidirectional, but the relationship between OrderDetail and ShoppingCart is bi-directional.



Generalization:

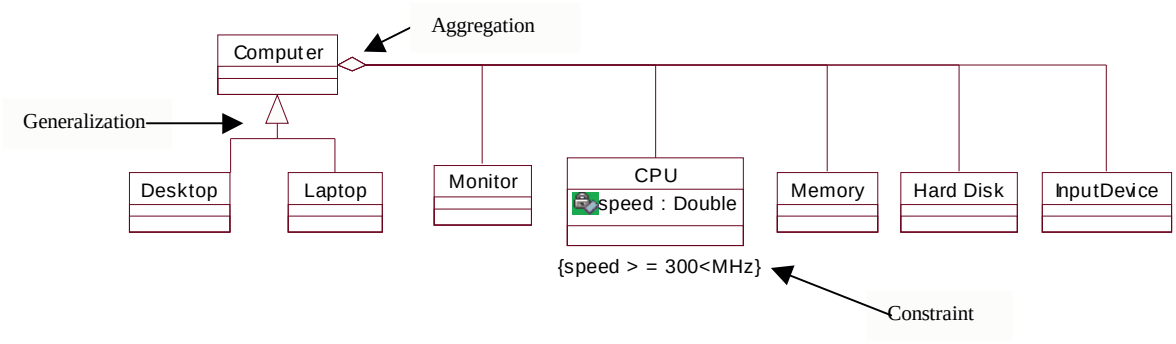
Generalization is a special form of association between a class and one or more refined versions of it. **Generalization is mapped to inheritance at the implementation level**. Inheritance can be defined as a mechanism by which a subclass will share attributes and operations from the super class using the generalization relationship.

Aggregation:

Aggregation is a special form of association in which the aggregate object is made of component objects. In other words, the aggregate object is the result of "and-ing" a series of components.

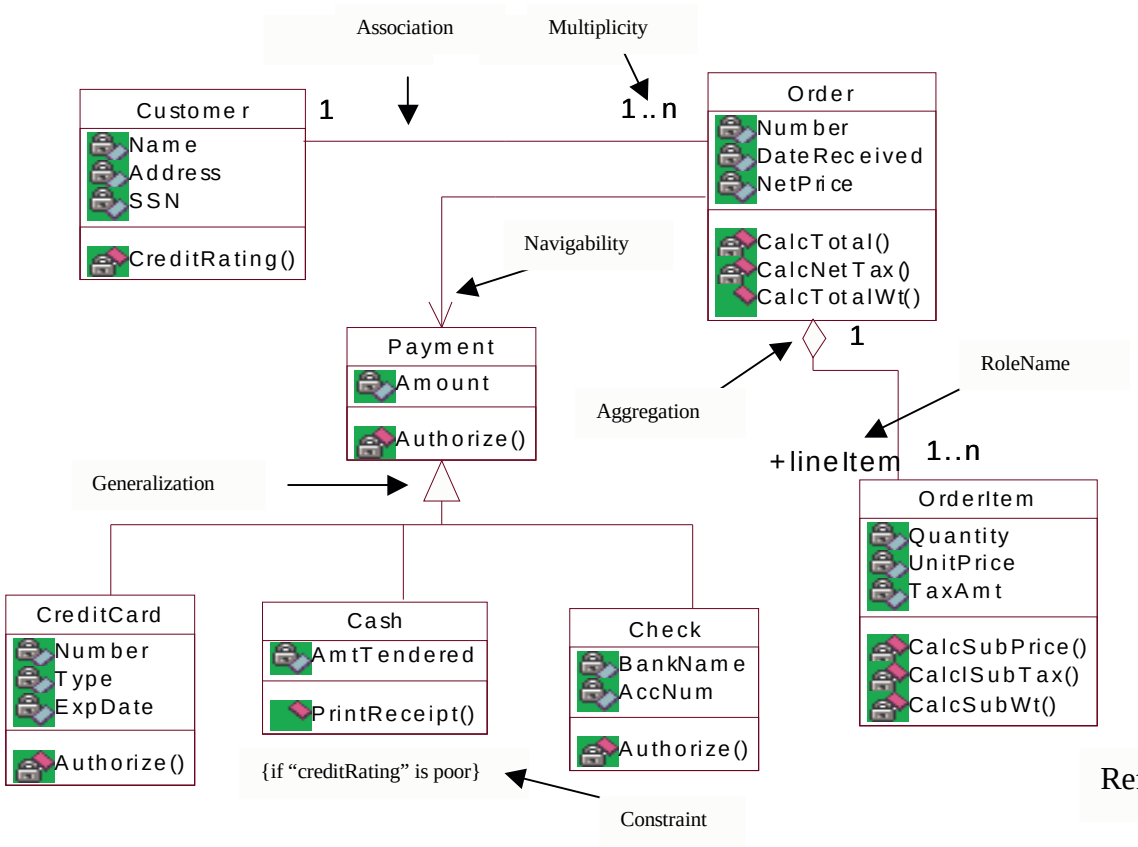
Constraint:
A Constraint controls the values that a class can presume. It is enclosed in between curly braces and is placed near the constrained entity.

Example 3.5: Aggregation, Generalization, and Constraint



Example 3.6: Integrated Illustration

A customer can place one or more orders from an online store. Each customer has a credit rating. If the credit rating is good, the customer can make the payment either with a credit card or a bank check. However, if the credit rating is poor, the customer has to pay in cash. Each order is an aggregate of different order items. A customer's order can have multiple items with varying quantities. Draw a class diagram for the above online store transaction.



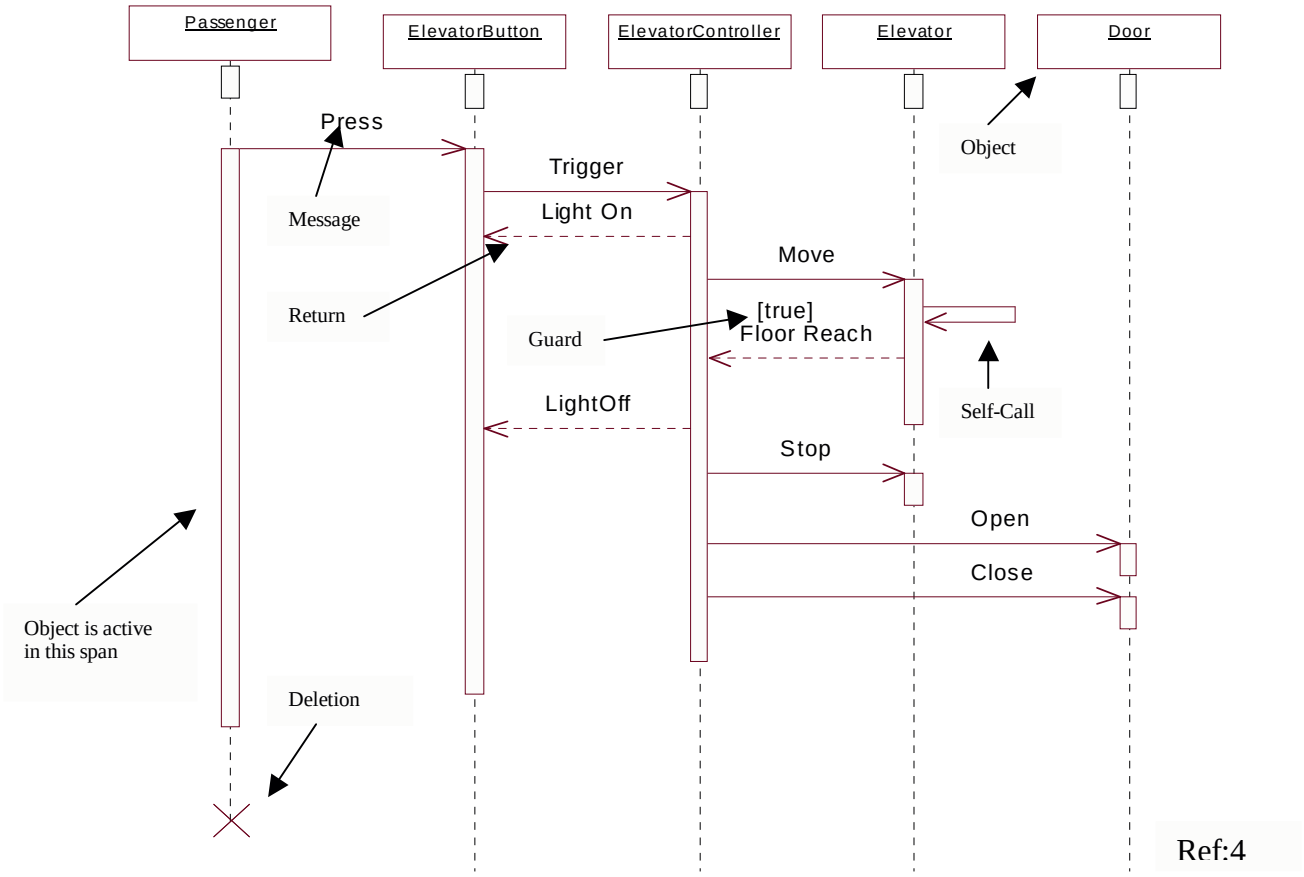
Ref: 3

When to use class diagrams: Class diagrams can be used to get a complete overview of the system or its parts through out the life cycle. The biggest temptation with the class diagrams is to jot down every detail of the system too early. Instead, **the focus should be on key ideas and enlargement should be incremental.**

Section 4 - Sequence Diagrams

Sequence diagrams model the dynamic aspects of a software system. **The emphasis is on the "sequence" of messages rather than relationship between objects.** A sequence diagram maps the **flow of logic** or **flow of control** within a usage scenario into a visual diagram enabling the software architect to both document and validate the logic during the analysis and design stages.

Example 4.1: Sequence diagram for the use case: Servicing the Elevator Button



A **Lifeline** represents the duration during which an object is alive and interacting with other objects in the system. It is represented by dashed lines. The long, thin boxes on the dashed lines represent method activations and they indicate that a process is being performed by the target object to fulfill a message. A **Self-call** occurs when an object sends a message to itself. In

the above example, the elevator object is sending a message to itself when the elevator reaches the specified floor. Overall, the object is making a self-call to itself through its own operations. The flow of messages between two objects can be controlled by placing a **condition** between them. The flow of messages between objects is allowed only if the condition is true. A **Return** represents a callback from a message indicating that the control has come back after the message has been serviced. Too many returns in the diagram will make it cluttered. **Deletion** of an object implies that the object no longer exists or is destroyed. An object can be destroyed either by a message from another object or it could self-destroy. The notation for deletion is X.

When to use sequence diagrams:

Sequence diagram can be a helpful modeling tool when the **dynamic behavior of objects** needs to be observed in a particular use case or when there is a need for visualizing the "**big picture of message flow**".

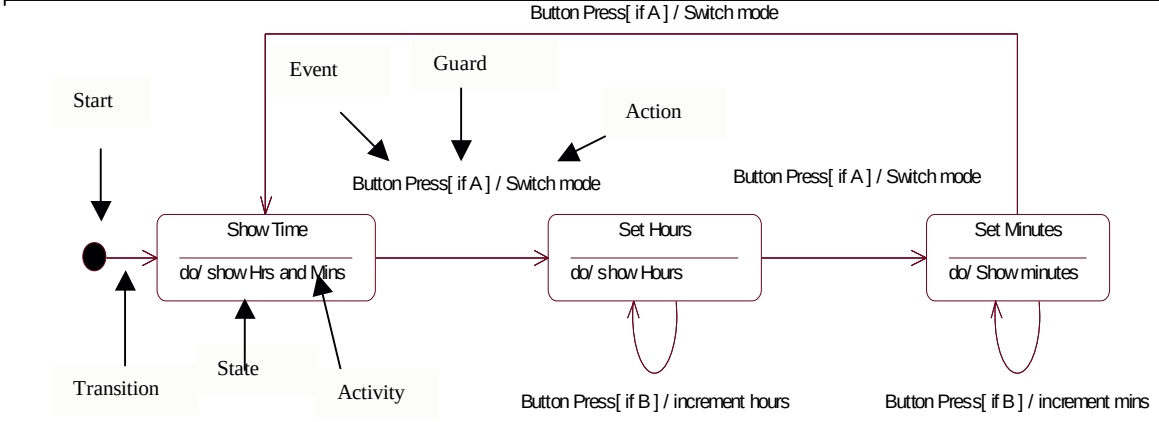
Section 5 - State Diagrams

A state diagram is a **graph in which nodes correspond to states and directed arcs correspond to transitions labeled with event names**. A state diagram combines states and events in the form of a network to model all possible object states during its life cycle, helping to visualize how an object responds to different stimuli.

A **state** can be defined as the duration of time during which an object is doing an activity. An **event** occurs at a point in time and transmits information from one object to another. An **action** occurs in response to an event and cannot be interrupted. An **activity** is an operation with certain duration that can be interrupted by another event. For example, a bulb in the "On" state is doing a continuous activity of "illumination" and this operation can be interrupted by another event like "switch off". A **guard** is a logical condition placed before a transition that returns either a true or a false. A guarded transition occurs only if the return value is true.

Example 5.1: Simple Digital Watch

A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time. In the display time mode, hours and minutes are displayed, separated by a flashing colon. The set time mode has two modes, set hours and set minutes. The A button is used to select modes. Each time A is pressed, the mode advances in sequence: display, set hours, set minutes, display etc. Within the sub modes, the B button is used to advance the hours or minutes once each time it is pressed. Buttons must be released before they can generate another event. Prepare a state diagram of the watch.

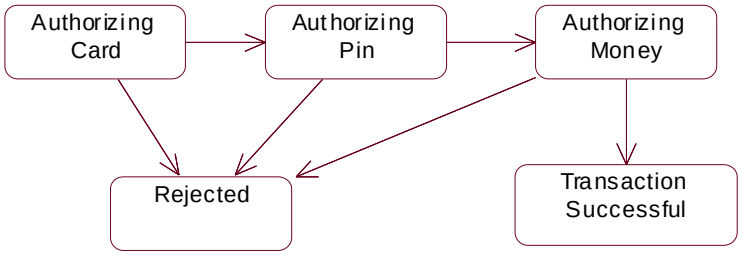


Ref:5

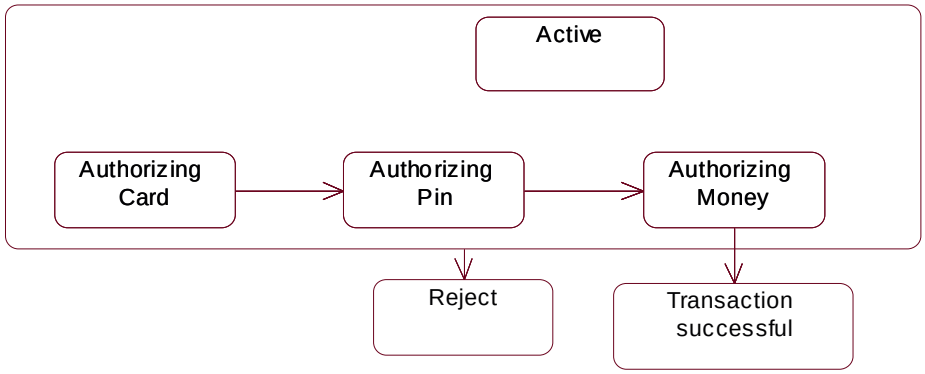
Super-state:

Super-states can simplify complex diagrams. For example, Figure 5.2 shows that the "Rejected" state could arise from either of the Authorizing states. All the three authorizing states can be grouped into a single super state called the active state to streamline the representation.

Example 5.2: Without Super state



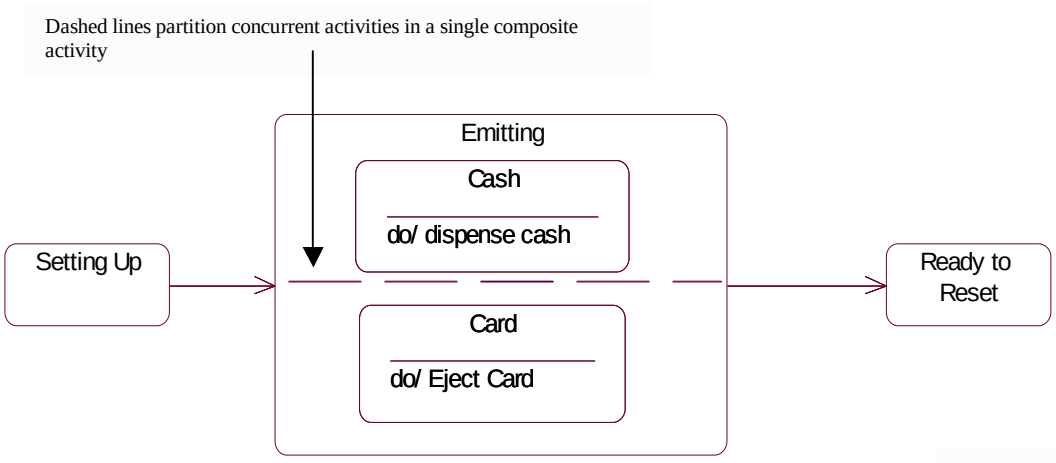
All the authorizing states grouped as "active" Super state



Concurrent State Diagram:

An object could be in more than one simultaneously occurring states when performing an activity. **Concurrent state diagrams are useful in modeling objects that have independent behaviors doing the same activity.**

Example 5.3: ATM in concurrent states



Ref: 5

For example, consider an ATM machine in the emitting state doing two concurrent activities: dispensing cash to the user and ejecting card. The control is split in a composite activity and later merged.

When to use state diagrams:

State diagrams are primarily necessary for those objects whose behavior across many use cases needs to be understood.

Section 6 - Activity Diagrams

An activity diagram is a type of flow chart with additional support for parallel behavior. Activity diagrams include the following new concepts.

Branches and Merges model the conditional behavior of activity diagrams. A branch has a single incoming transition and multiple, conditional, outgoing transitions. The control flow is directed to one of the outgoing transitions depending on the condition satisfied. A merge is a node in the activity diagram at which the conditional behavior terminates. **Each branch in an activity diagram has a corresponding merge**.

Forks and Joins model the parallel behavior of the system. A fork in the activity diagram has a single incoming transition and multiple outgoing transitions **exhibiting parallel behavior**. The incoming transition triggers the parallel outgoing transitions. A join in the activity diagram synchronizes the parallel behavior started at a fork. Join ascertains that all the parallel sets of activities (irrespective of the order) are completed before the next activity starts. It is a synchronization point in the diagram. **Each fork in an activity diagram has a corresponding join where the parallel behavior terminates.**

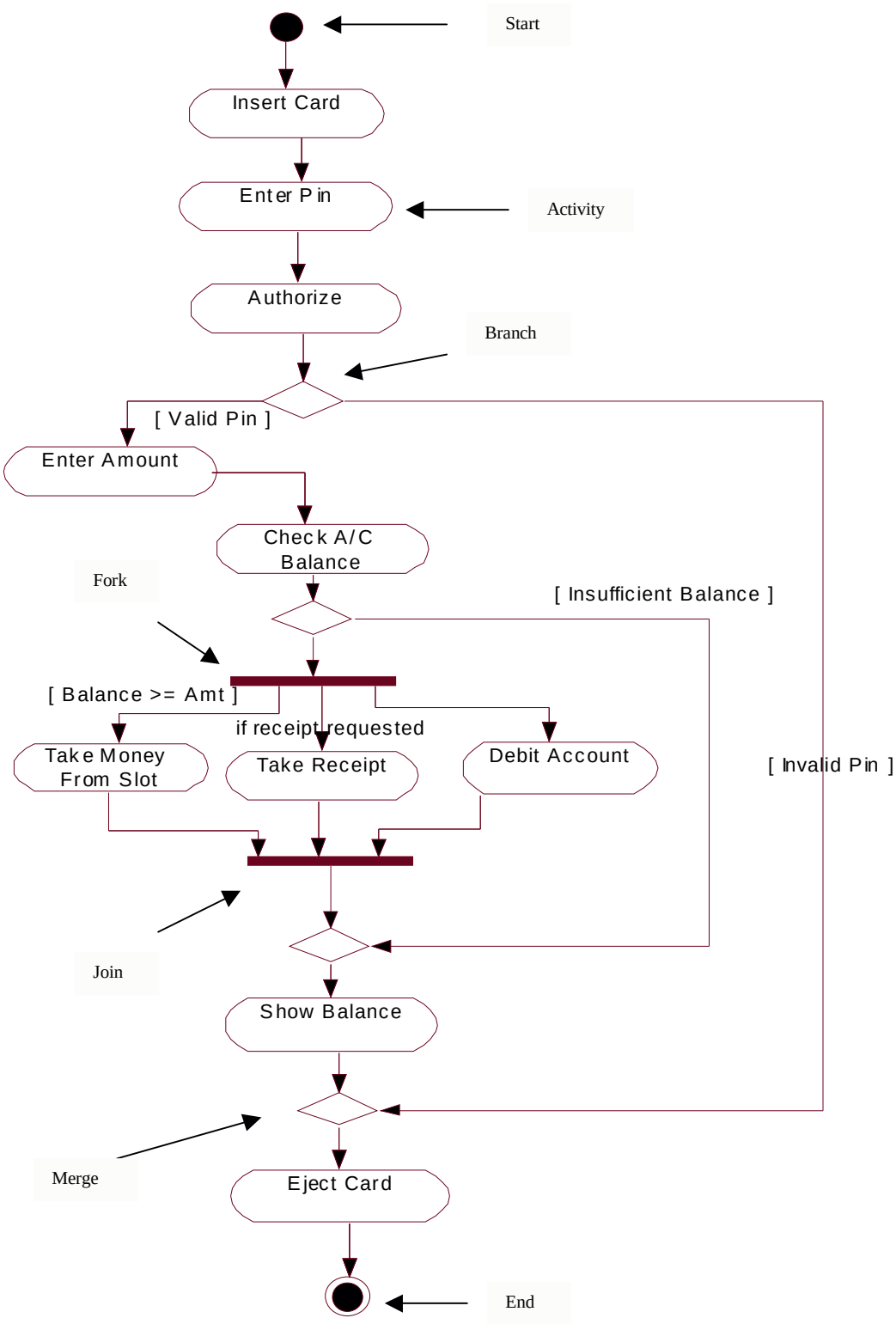
Conditional Thread. A condition is placed on the thread originating from the fork to create an **exception for the join rule**. As per the join rule, the conditional thread is assumed to have completed execution if the condition turns out to be “false”.

Synch State synchronizes different existing activities so that they make a transition to the next activity at the same time.

When to use activity diagrams:

Activity diagrams are most useful when modeling the parallel behavior of a multithreaded system or when documenting the logic of a business process.

Example 6.1 Process – Withdrawing money from the ATM



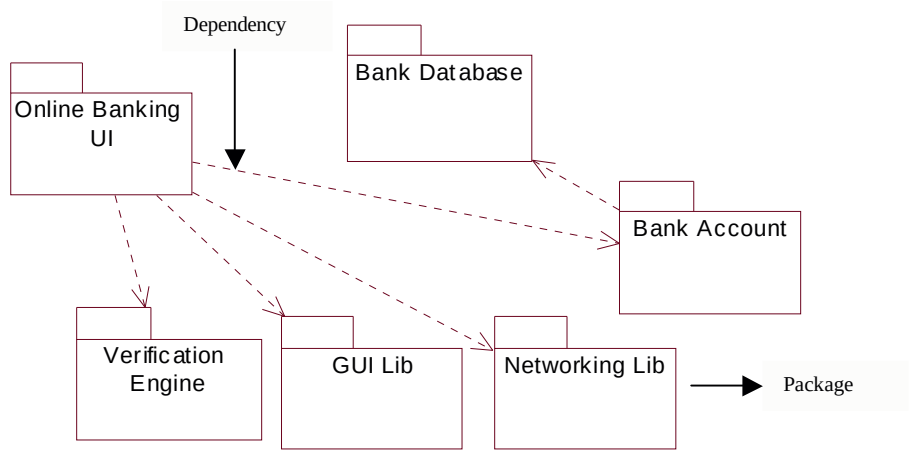
Section 7 - Package Diagrams

Package diagrams **group related classes** to help the software engineer to identify and to understand dependencies. In addition, **a system's overall structure can be controlled** with the help of a package diagram.

Two classes are said to be dependent on each other if changes made to one class causes changes to another class. A dependency is said to exist between two packages if any two classes in the packages are dependent. In example 7.1, package "Online Banking UI" is said to depend on "GUI Library" package because changes made to GUI library classes affect application's user interface.

Example 7.1: Package diagram for "Online Banking System"

In a given bank system, customers can access their account details by using the online facility provided by the bank. The user inputs the required authentication information which is verified before giving access to the account information. The user can then check the account balance, pay bills online, and transfer money from the savings account to checking account. The following package diagram documents an abstract design of the software by grouping related classes into packages. For example, the graphical user interface-classes, which provide user input and output capabilities, are grouped into GUI Library, while the security related classes are grouped into a Verification Engine package. Note: The diagram just shows the interdependencies originating at the Online Banking UI.



Ref:3

When to use package diagrams:
Package diagrams are most useful during the high-level design stages in large software projects to describe system's overall structure.

References:

[1] Rational: "UML Resource Center", www.rational.com , 2002
[2] SmartDraw: " Draw Anything Easily", www.smartdraw.com, 2002
[3] TogetherSoft: "A Hands On Introduction For Developers", www.togethersoft.com, 2002
[4] Geocities: "UML Examples", www.geocities.com, 2001
[5] James Rumbaugh, Michael Blaha, William Premerlani,Frederick Eddy, William Lorensen: "Object-Oriented Modeling And Design", Prentice Hall, New York, 1998