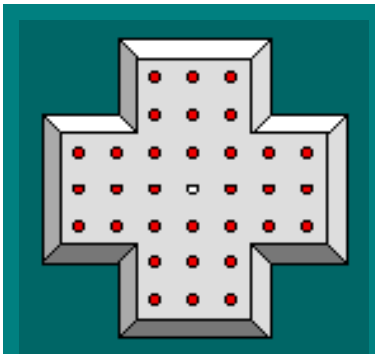


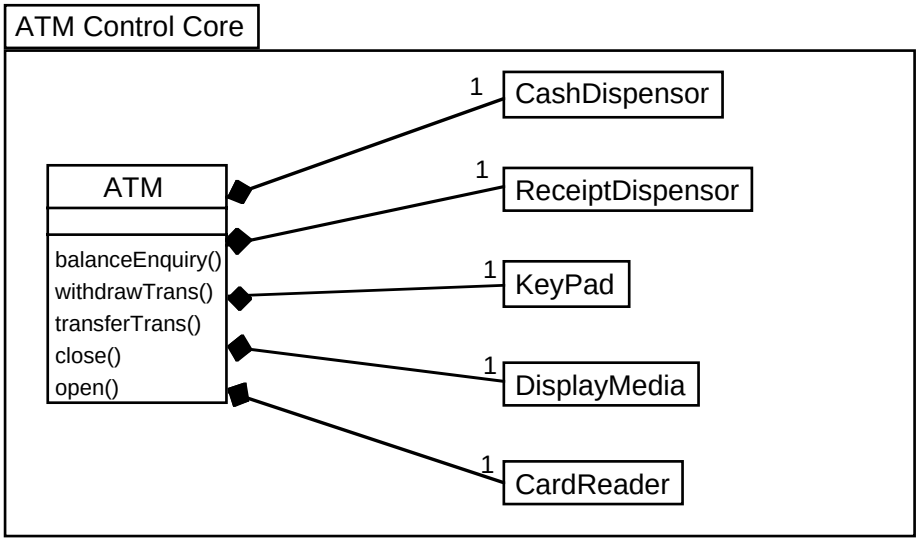
CHAPTER 4

DEVELOPING OBJECT ORIENTED SOFTWARE (ARCHITECTURAL DESIGN)

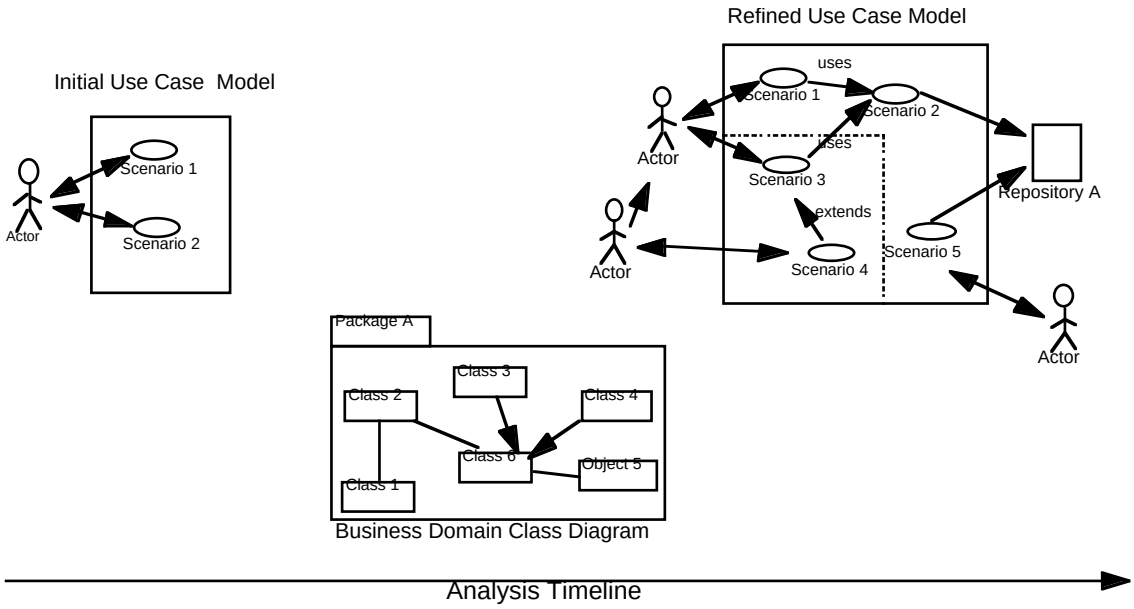
An Architectural Design is an informal object model that has two parts :

- Architectural (Class) Diagram
  which models the key objects (classes) in the problem domain.
- Object-Z Class Definitions
  which define the state changes for the key objects in the model.

ATM PACKAGE EXAMPLE



MODEL DEVELOPMENT



NOTATION (UML)

The Class

A class describes a collection of similar objects

- Same attributes : may have different values
- Same operations : may have differing results
- Similar relationship : some may be null
- Same constraints

> ***"A class is the abstract representation of a set of objects"***

Attributes

Attributes describe the properties of an object

e.g. Company has a name, ACNNumber

e.g. BankAccount has a balance, accNum

An actual object will have a value for each of its attributes

e.g. a Company object has name = "BHP", ACNNumber = "555 555 555"

e.g. a BankAccount object has balance = $50.04, accNum = "5555 55555"

Operations (methods)

Operations are actions/functions/transformations on an object

e.g. withdraw might be an operation on a BankAccount

Operations are specified on the class and are shared by all instances of that class

e.g. withdraw works for myBankAccount and yourBankAccount

Notation

Class Name
Attribute Name • • •
Operation Name • • •

Example

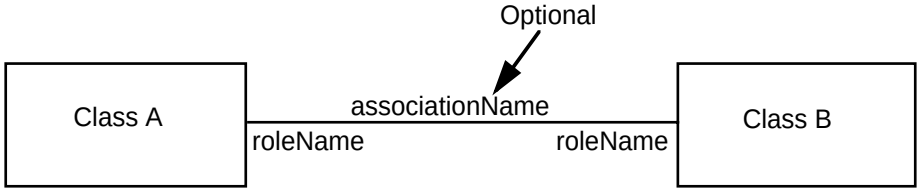
BankAccount
balance accNum
withdraw fundsAvailable

Association

Associations model the "uses-a" relationship between classes

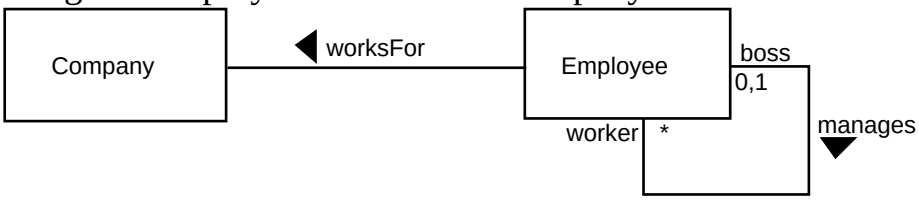
Associations are bi-directional

Notation



Example

e.g. an Employee works for a Company



Note that an association may be self-referencing

Multiplicity (Cardinality)

One object may be associated with any number of instances of another class

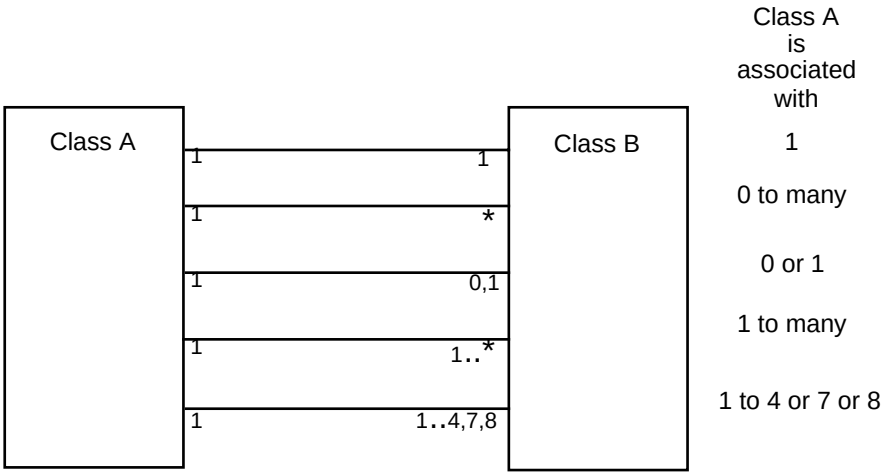
e.g. Customer may be associated with many BankAccount objects

e.g. Company may employ many Employees

Recorded as multiplicity on the association

Multiplicity records the constraints on the association

Notation



Aggregation (Composition)

In UML the strong form of aggregation is known as composition. Used in analysis.

Aggregation represents the "is-part-of" relationship

e.g. a Car has parts which are Engine, Wheel, Door

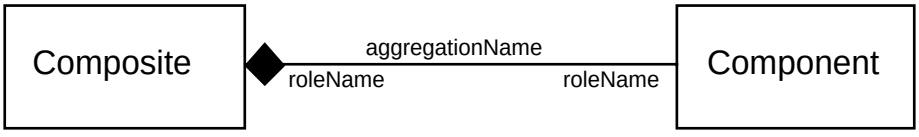
Aggregation is a specialised form of association with the following extra semantics

- Transitivity : If B is part of A and C is part of B then C is part of A
- Antisymmetry : If B is part of A then A is not part of B
- Propagation : The state of the whole is determined by the state of the parts

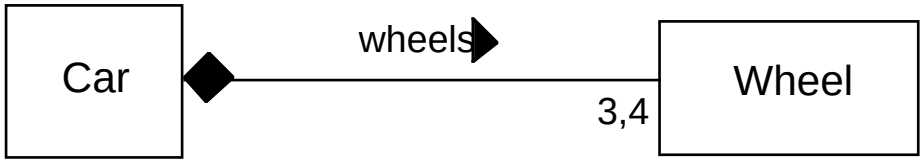
Physical objects often involve aggregation

Be careful when using aggregation - consider whether the extra meaning is really needed.

Notation



Example



Inheritance

Inheritance represents the is-a relationship

e.g. a SavingAccount "is-a" specialised form of a BankAccount

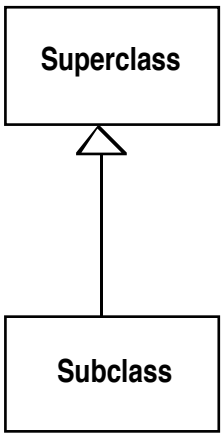
Inheritance is a powerful class abstraction mechanism for relating similar definitions through a specialisation/generalisation relationship

The more general class is called the **superclass**

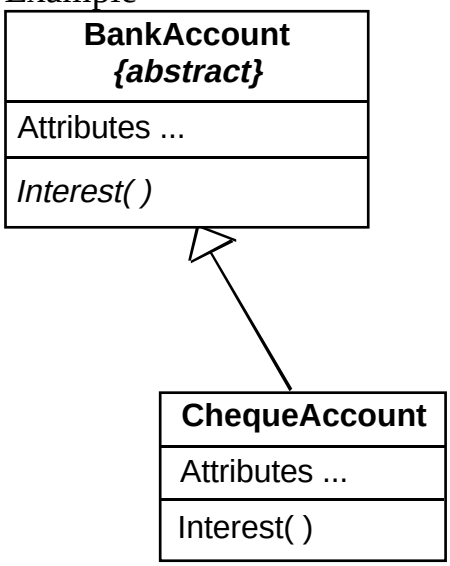
The more specific class is called the **subclass**

A subclass inherits **all** the attributes, operations and relationships of the superclass

Notation

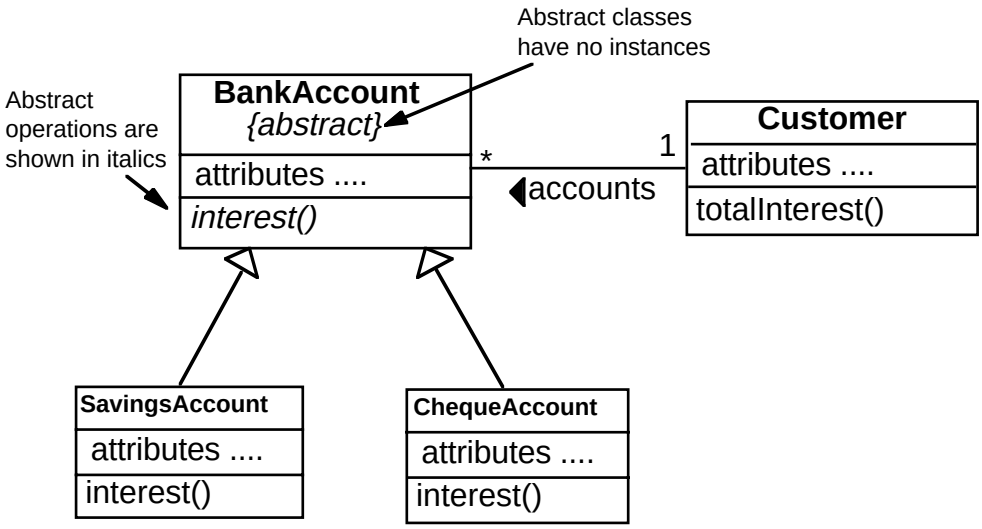


Example



Inheritance & Polymorphism

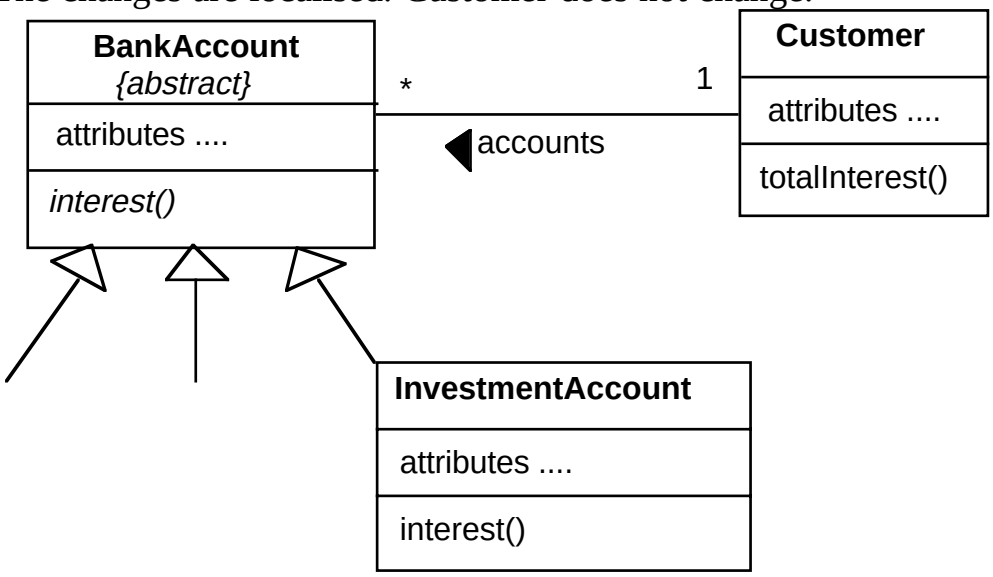
Example



Inheritance & Extendability

Extend example to include a new type of account

NOTE : The changes are localised. Customer does not change.



Package

A **Package** is a high level decomposition unit and represent clusters of related classes.

A Package can contain other packages

Packages can have relationships to other packages

- depends-on
- inheritance

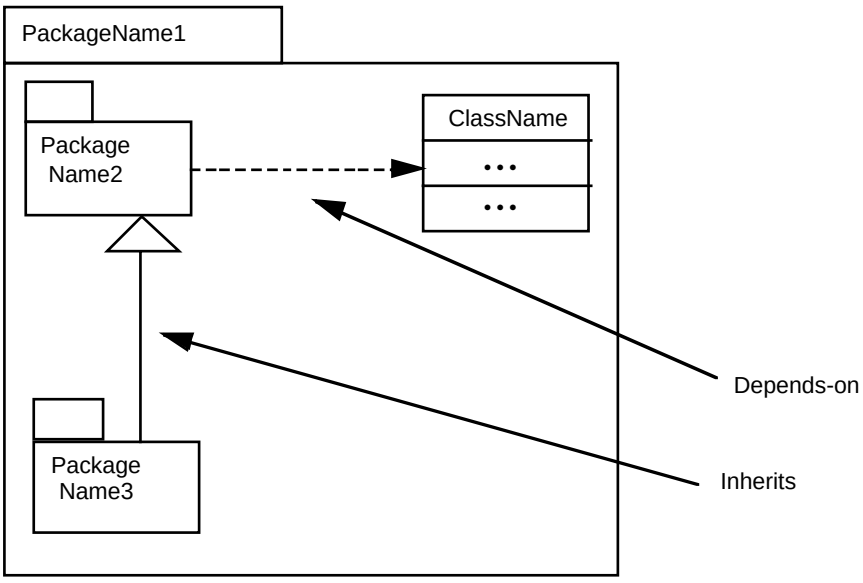
The entire system can be thought of as a single high-level package with everything else in it.

A class in one package can be referenced from a different package

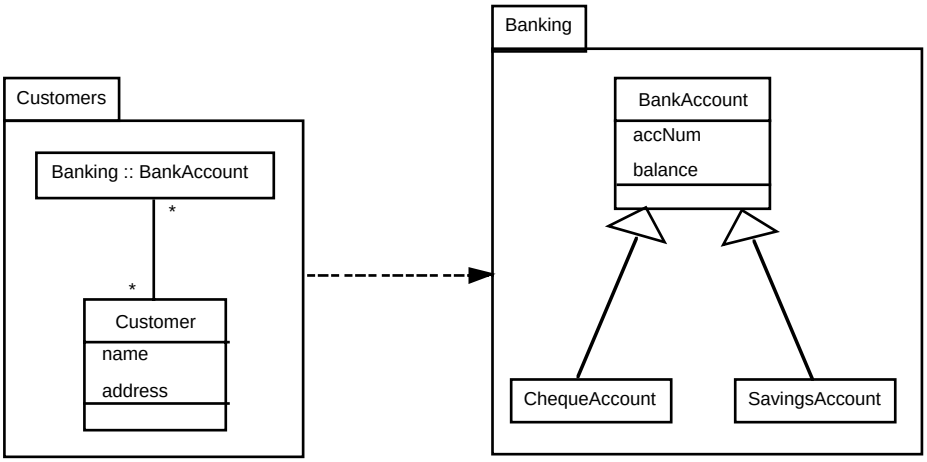
package-name :: class-name

e.g. Currency :: Money

Notation



Example



ACTIVITY 2 : ARCHITECTURAL MODELING

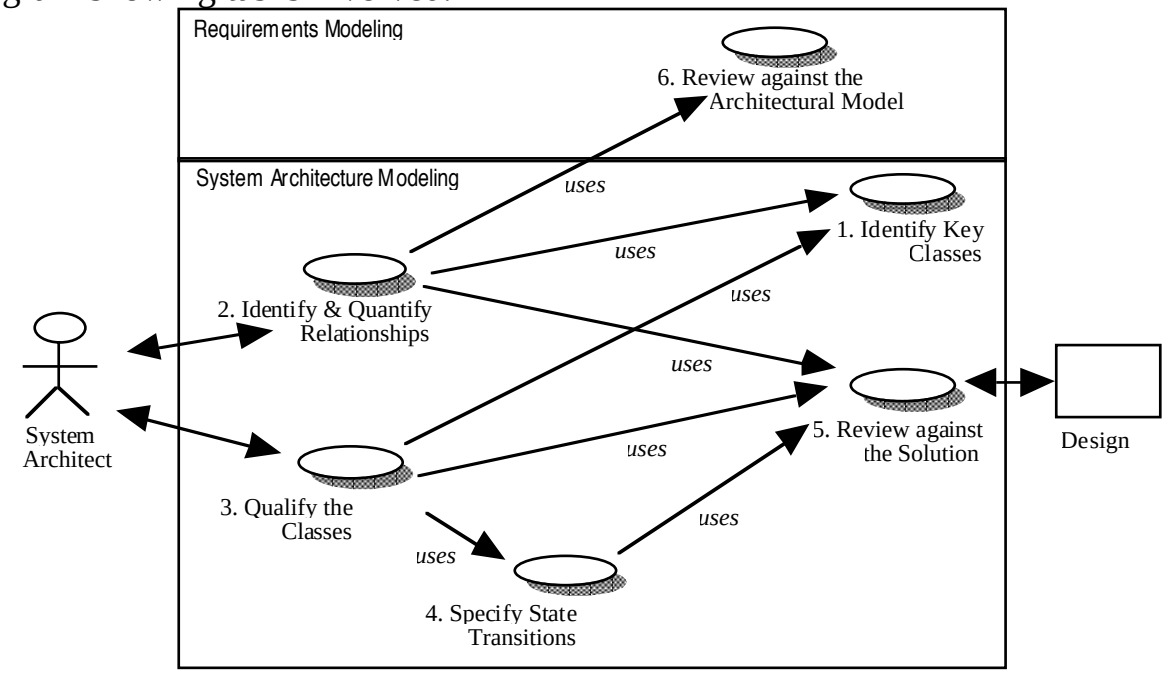
Overview

Architectural Modeling facilitates the identification and qualification of the classes existing within the problem space

It produces a **System Architecture Model** that establishes the key vocabulary for the requirements model. It comprises:

- a class diagram of the key classes in the problem space
- Object-Z class definitions which explore the state changes of the key classes

Diagram showing tasks involved.



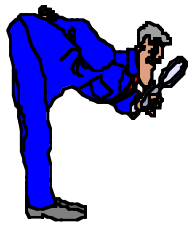
TASKS

- Task 1 : Identify Key Classes
- Task 2: Identify & Quantify Relationships
- Task 3 : Qualify the Classes
- Task 4 : Specify State Transitions
- Task 5 : Review Against the Solution

TASK 1: IDENTIFY KEY CLASSES

Classes of objects are the key abstractions of the problem domain and the solution domain. Hence, they form the link betweem analysis and design.

Where to Look



Scenarios

These contain descriptions of concepts which are important in the domain. These are candidates for becoming the objects of the object-model

Domain Experts

The vocabulary of the domain expert will provide many of the key objects. These should be captured in the scenarios

Informal English Descriptions

Underline the nouns in the problem to identify candidate objects and the verbs to identify the operations

Library Classes

If library classes are available for the domain they should be examined for possible abstractions

Enterprise Model

The key business entities for the business area are important sources of information for the object model

Prototype

A prototyping exercise from the investigation phase will give rise to abstractions that should be used in this phase

What to Look For

Other Systems	The interface to external systems that interact with this system are best modelled as objects. Examples include a screen, a communications protocol, or an external corporate database
Devices	A physical device will probably be reflected as a software object in the object model
Tangible Items	Items in the domain probably have a software counterpart in the object model
Sites	Physical locations that are remembered
Roles Played	Roles played by people and their interactions with the system. Examine Actors from the requirements modeling activity
Organisation Units	The organisational units that people work in

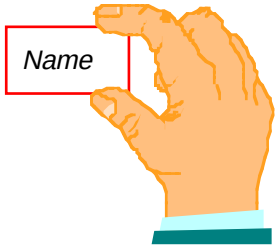
Different Types of Classes

4 major types of classes are identifiable in a class model:

- Interface Objects
- Entity (or Model) Objects
- Controller Objects
- Infrastructure/Framework Objects

Only the Entity (or Model) objects are of interest at this stage. Disregard the other types of object when producing an Architectural Model

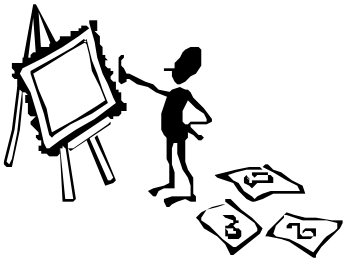
Name the Key Classes



- A class name should be a singular noun, or an adjective and a noun
- Use common noun phrases
- Names should be those used by the client to describe the domain unless explicit renaming is required
- Names should be easy to read and unique
- Our convention is to capitalise the name of a class

Refine the List of Candidate Classes

Once candidate classes are identified, challenge each for its usefulness in the problem. Issues to question, challenge or consider include:



- Needed Remembrance
- Needed Behaviour
- Multiple Attributes
- Always Applicable Attributes
- Always Applicable Services
- Not Merely Derived Results

Example - ATM Candidate Classes

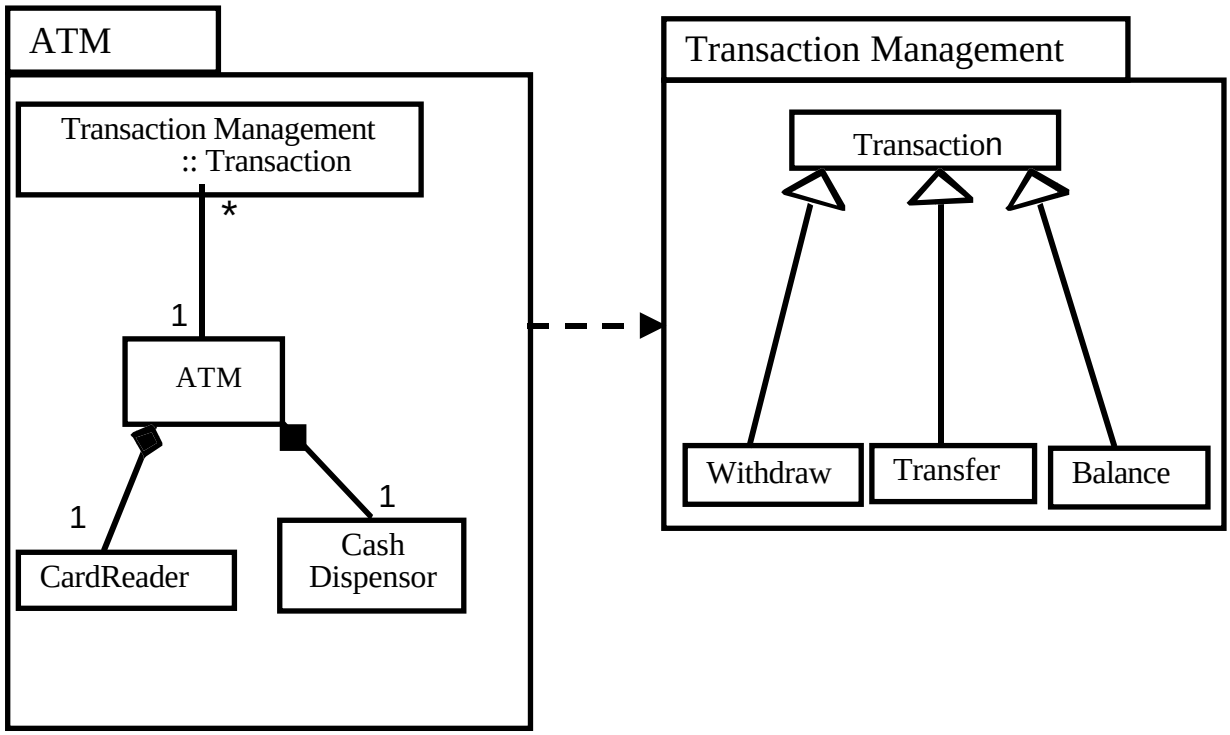
Investigate Primary Scenario for *Verify Customer* and highlight possible candidate classes.

Step :	Source :	Action :
1	User	inserts card into card reader.
2	ATM	checks that card is valid for this ATM.
3	ATM	display request : "enter PIN number then press OK key".
4	User	enters PIN using the numeric keys and presses the OK key.
5	ATM	verifies that card and PIN are valid with bank.
6	ATM	logs the customer connection.
7	ATM	display request : "select transaction type using transaction keys or press the Cancel key to discontinue use of ATM".

Identify Key Subject Areas

When the coverage of the Architectural Model grows, it may encompass several logical groupings and each grouping is termed a subject area and is modeled using packages. These become subsystems as a solution emerges.

Example - ATM Subject Areas



SOFTWARE ARCHITECTURE

An abstract view of the system's major components (key subject areas), the behaviour of these components, and the ways in which these components interact and coordinate (connectors).

- Components
  Perform the required functionality of the system.
- Connectors
  Represent the interactions between the component interfaces in the system.

THREE-TIER ARCHITECTURE

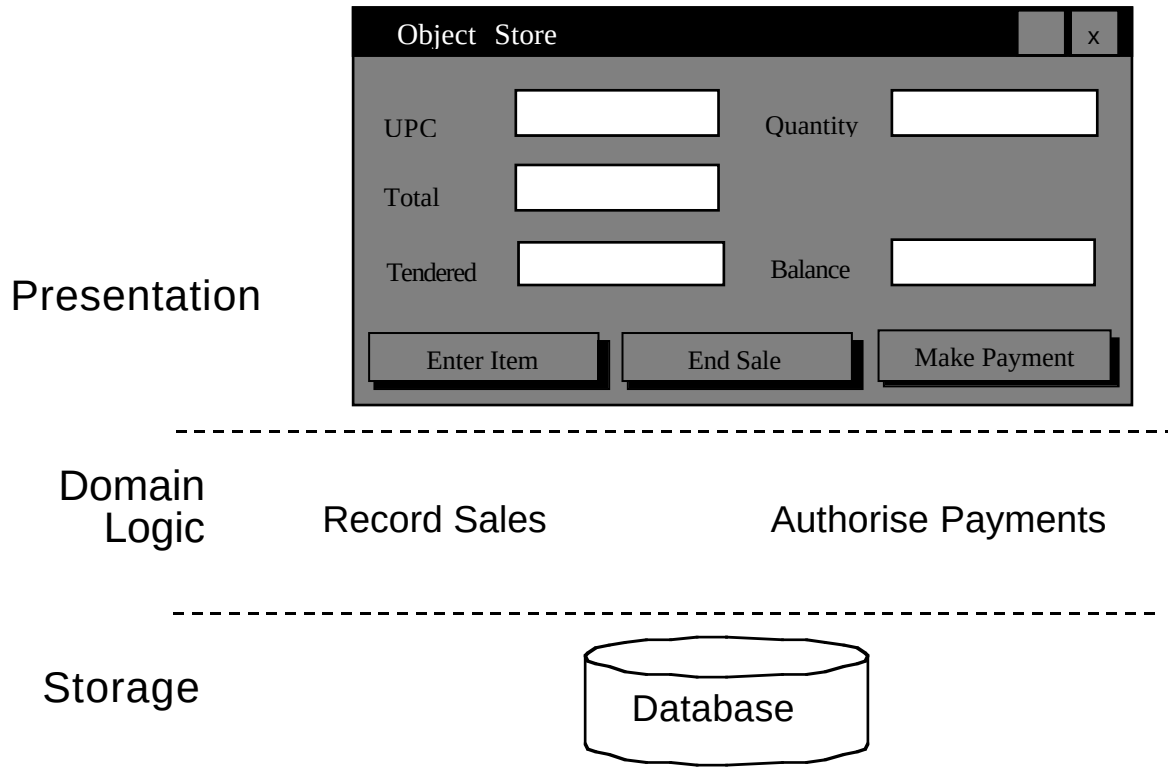
The three-tier architecture consists of three layers, each of which may be viewed as a component:

- Presentation - interface, eg. windows, reports, etc.
- Domain Logic - tasks and rules that govern the process.
- Storage - persistent storage mechanism.

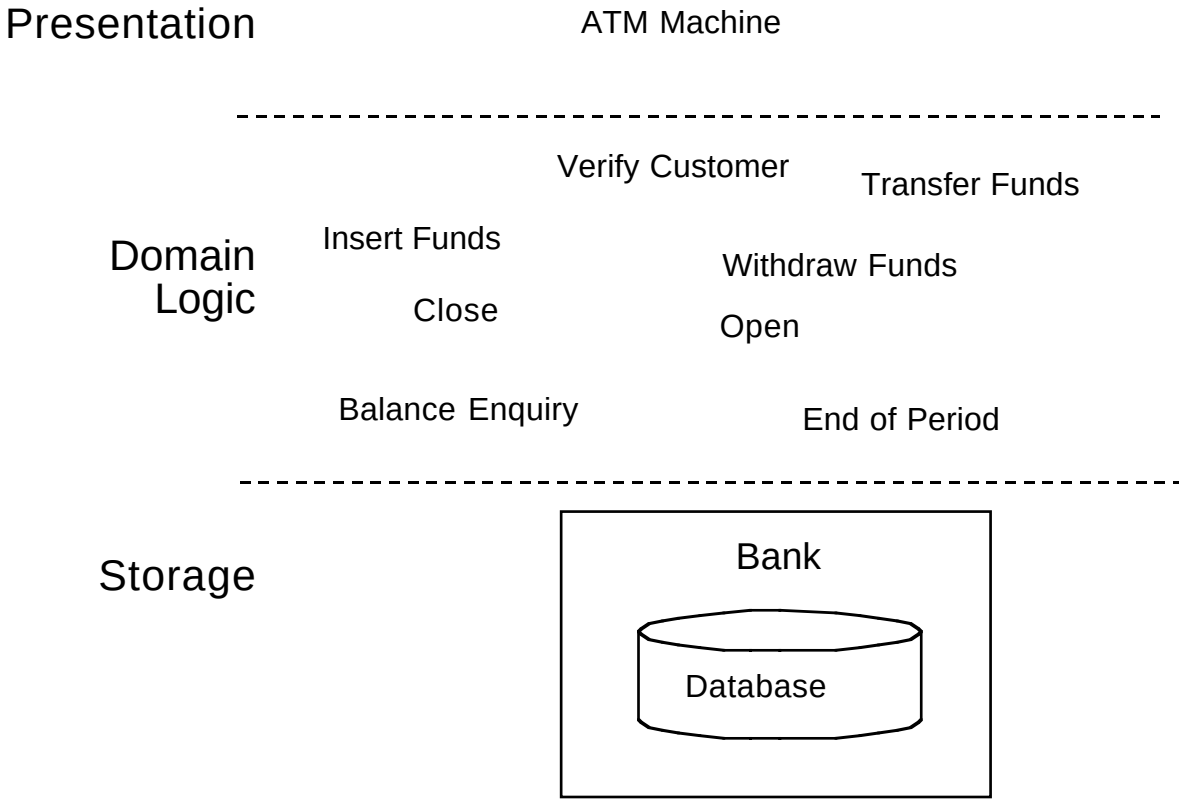
The advantages of the three-tier architecture include:

- reuse due to the isolation of the domain logic; and
- allocation of the development team to construct individual tiers, thus supporting the ability to run parallel team development and specialised expertise in view of development skills.

EG - THREE-TIER ARCHITECTURE



EG – ATM - THREE-TIER ARCHITECTURE



DESIGN PATTERNS

- Design patterns are successful solutions to common software development problems.
- Design patterns:
  - communicate successful solution strategies
  - prevent traps and pitfalls
  - prevent 're-invention of the wheel'

MODEL-VIEW PATTERN

- The separation of domain functionality from the presentation functionality.
  - Model - domain layer
  - View - presentation layer

- Model components should not have direct knowledge of, or be directly coupled to view components.

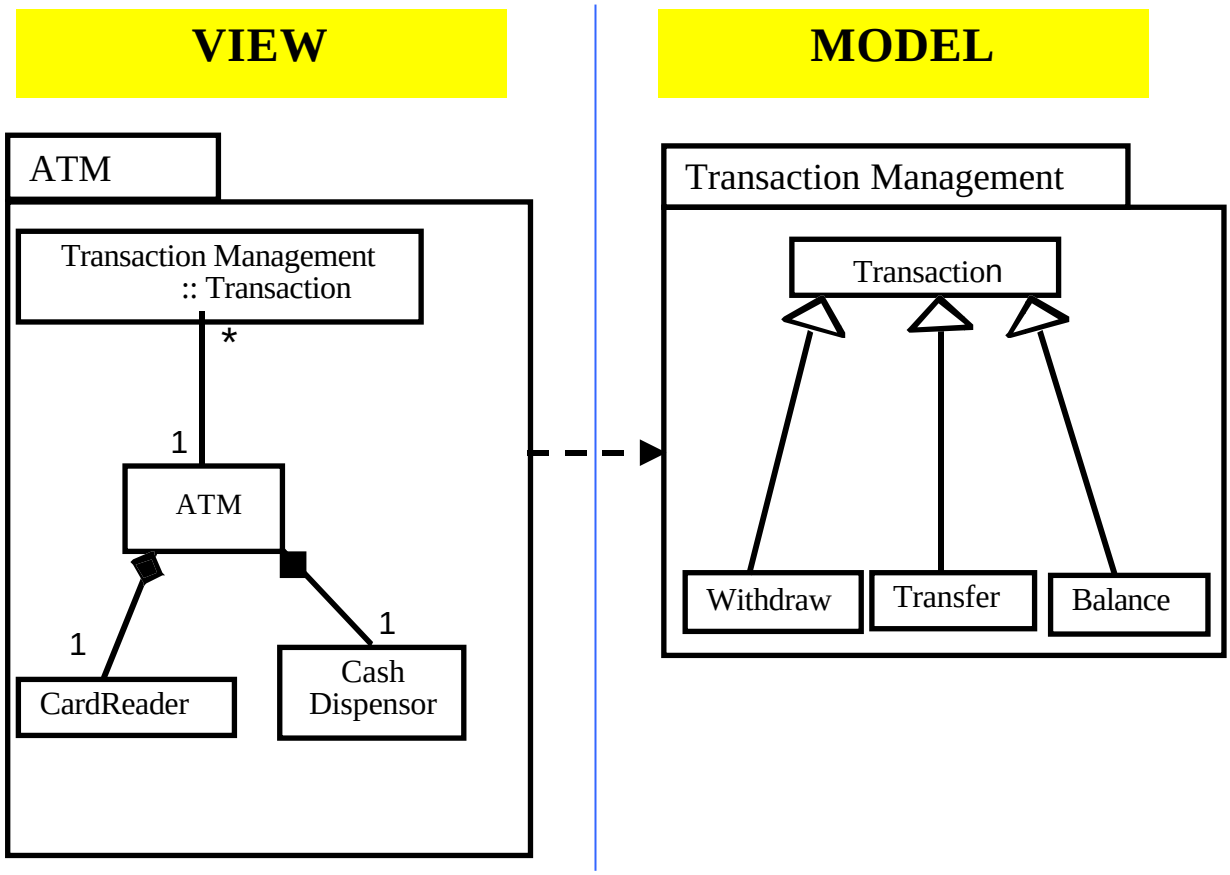
Separating these concerns reduces the complexity in developing the system and it also improves the maintainability of the system.

Generally, it is the responsibility of the view to do the following:
- instantiate and arrange required window objects;
- instantiate and initialise the model;

- handle any events generated by the user by sending them to the model, e.g. button clicks, menu selection; and
- accurately represent the model to the user.

The responsibilities of the model are as follows:
- define and manage the domain data;
- responding to any messages sent from the view.

EG – ATM - MODEL-VIEW PATTERN



TASK 2: IDENTIFY & QUANTIFY RELATIONSHIPS

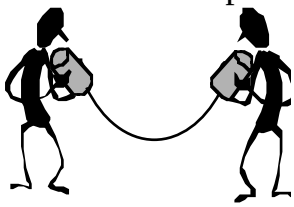
An Architectural Model only ever contains 3 types of relationships:

- Association
- Aggregation
- Inheritance

In addition, to quantify the number of instances of particular objects within a problem domain, cardinality adornments may also be shown on the model

Identify Association Relationships

Association captures the static "*uses-a*" relationships between classes.



Association relationships denote some bi-directional conceptual relationship between two classes. Similar to the "Relationship" in ER modeling

Appropriate during analysis and early design but refined later as implementation decisions are made.

None of the current OO programming languages implement the concept of association. Therefore the association relationship must be transformed for implementation (typically into a pointer or reference).

Association relationships can be identified from scenarios and confirmed with the domain experts.

Identify Aggregation Relationships

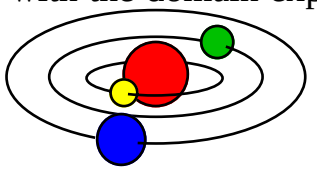
The "*is-a-part-of*" relationship models:

- an object which is an aggregation of (made up of) other objects.
- a class which is using another class only for its own internal implementation.

It is identified from the scenarios and confirmed with the domain expert.

Aggregation may model:

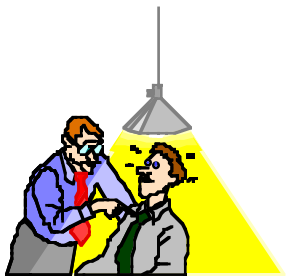
- Assembly - Parts
- Container - Contents
- Collection - Members



Qualifying the Aggregate Relationship

A class could be a potential part or a potential composite. Challenge each class in the relationship by asking:

- Are there objects of this class in the problem domain?
- Is the object of this class within the system's responsibilities?
- Does the class capture more than a status value? i.e. is it a useful abstraction in dealing with the problem domain? If not, then just include an attribute for it within the composite class.
- What are the parts of the objects of this class?



Identify Cardinality

Once basic association and aggregation relationships are established, cardinality can be assigned based on the requirements and scenarios.

Cardinality should be as general as possible (i.e. 0..*) to encourage reuse but, certain operational rules may dictate specific cardinalities. These are recorded on the class diagram.

Identify the Inheritance Relationships

Inheritance models a generalisation/specialisation ("*is-a*") relationship. A subclass automatically has all the attributes and operations of the superclass.

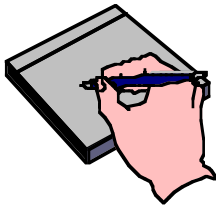
The list of classes must be assessed to factor out commonality into superclasses.

Inheritance is generally identified last in the requirements modeling process because commonalities between classes can not accessed until the classes have been completely investigated.

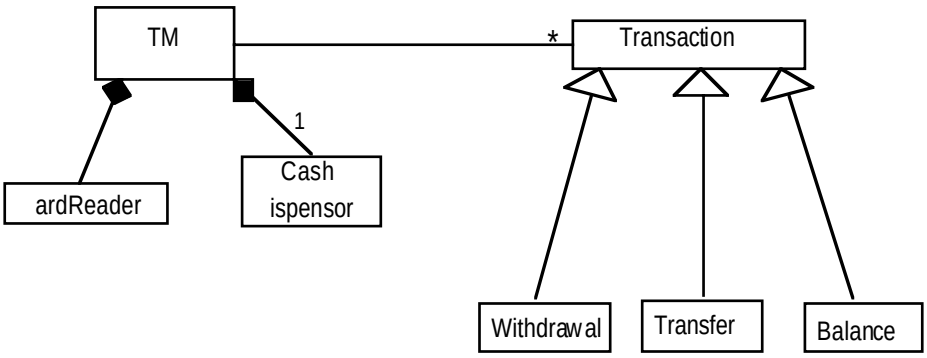
Multiple inheritance may be modelled, but exercise caution when using multiple inheritance too early/freely in the design. **(don't do it!)**

Drawing the Architectural Model

The major output from the work so far is a draft class diagram.



Example - ATM



TASK 3: QUALIFY THE CLASSES

Identifying Attributes

Work through the scenarios to identify and allocate attributes to the candidate classes. A class may be used in more than one scenario, therefore a complete picture of its attributes requires that all scenarios be examined.

Questions to ask about each class are:

- How is it described in general?
- How is it described in this problem?
- How is it described in the context of the system responsibilities?
- What information does the class need to know?
- What states can the class be in?

Finding Operations

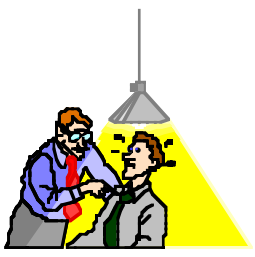
Assign operations to classes using the following guidelines:

- Evenly distribute system functionality
- State operations as generally as possible
- Insure operations are related to attributes
- Keep information about one thing in one place
- Similar behaviour should be given the same name
- Reuse is more likely if there are few parameters
- What must an object:
  - do for others?
  - look after internally? (what knowledge must it maintain)

Qualifying the Operations

Questions to ask regarding the operations of a class include:

- What must a class do for others and what must it maintain?
- What calculations is the object responsible for performing?
- What is the object responsible for monitoring in order to detect and respond to external changes in the environment?



CRC Forms

You may find it useful to organise the details of classes with CRC (class responsibility collaboration) forms to record the information, using one for each class. The idea is to organise the relationships between the classes on the basis of fulfilling a scenario. The CRC forms provide a mechanism for following through a scenario, identifying relationships from the scenario, and allocating attributes and operations to classes in a review process.

NOTE : CRC forms can also be used early in the design process as a discovery aid. When the operations and responsibilities of a class are identified through design reviews, details are added to the form.

Class name :	
Superclass :	
Role :	
Attributes :	
Responsibilities	**Collaborators**

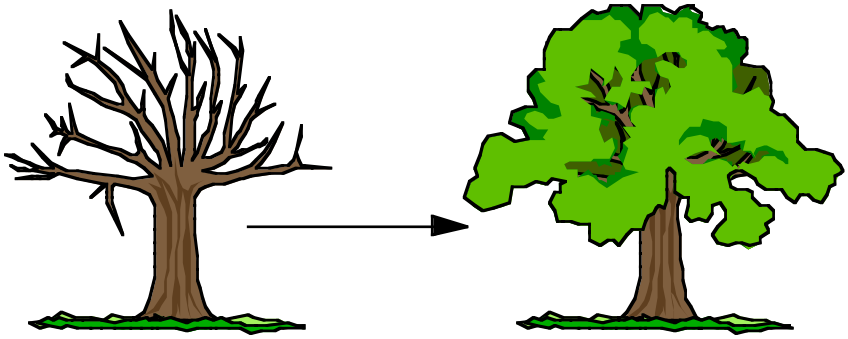
Example CRC Forms

Class name : Bank	
Superclass : nil	
Role : Validates transactions and Customers by communicating with the Bank's central accounts database.	
Attributes : customers, accounts	
Responsibilities	**Collaborators**
Validate a Customer	Card, PIN
Validate a withdraw transaction	Card, Account

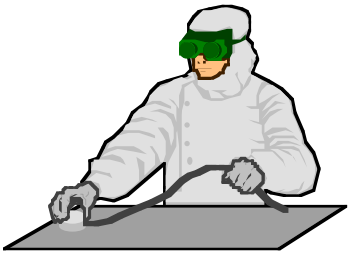
Class name : Withdrawal	
Superclass : Transaction	
Role : Store withdraw transaction information	
Attributes : date, time, account, card Id, ATM Id, amount	
Responsibilities	**Collaborators**
Knowing date of transaction	Date
Knowing time of transaction	Time
Knowing amount of transaction	Money
Knowing card used in transaction	Card

TASK 4: SPECIFY STATE TRANSITIONS

The state changes of key objects will be described by using the formal specification language known as Object-Z. We will study this in detail in the next chapter.



TASK 5: REVIEW AGAINST THE SOLUTION



The third phase (Development Phase) of the process will refine the definition of a solution by designing, implementing and testing components of the system. As this work proceeds it is likely that definitions for key classes in the Architectural Model will need to be refined or altered. The review insures that the Architectural Model is updated to reflect the current understanding.

Exercises : Relationships and Inheritance

QUESTION 1

(a) Given the model at the right answer the following questions:

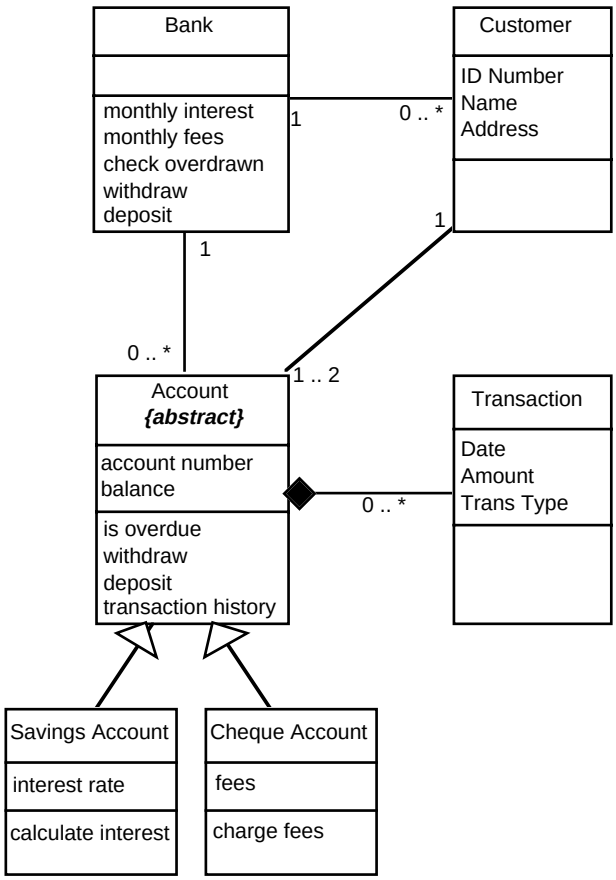
(i) What is the maximum number of ACCOUNTs that each CUSTOMER can have ?

(ii) What is the name of the relationship between ACCOUNT and SAVINGS ACCOUNT ?

(iii) What type of class is ACCOUNT ?

(iv) What is the maximum number of CUSTOMERs that each ACCOUNT can have ?

(v) What is the name of the relationship between ACCOUNT and TRANSACTION?



(b) Given the model at the right answer the following questions:

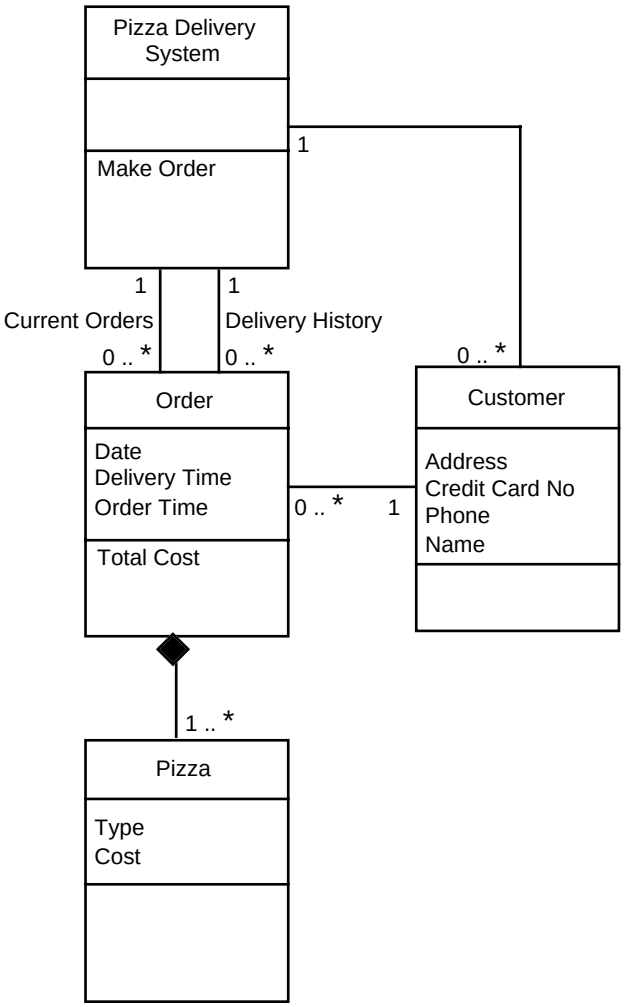
(i) What is the minimum number of ORDERs that each CUSTOMER can have ?

(ii) What is the name of the relationship between ORDER and PIZZA ?

(iii) What is the name of the relationship between ORDER and CUSTOMER ?

(iv) What is the maximum number of CUSTOMERs that each ORDER can have ?

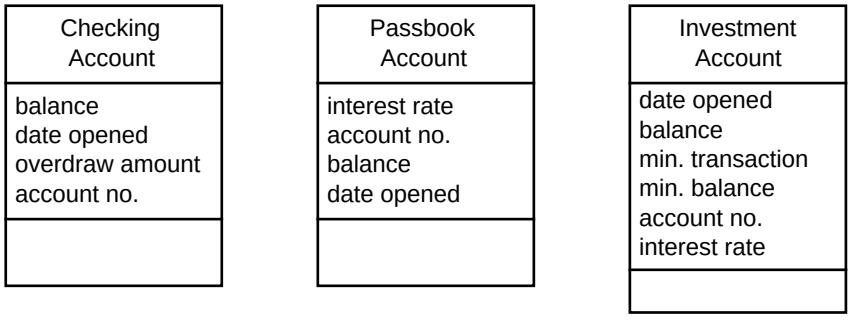
(v) What is the minimum number of PIZZAs that each ORDER can have ?



QUESTION 2

Construct class hierarchies from the following classes. HINT, you may need to include one or more other classes in your hierarchies.

(a)



(b)