



**University of the Punjab**  
**Gujranwala Campus**

---

## **Second Deliverable**

---

**Project ID: BIT-2109**



## TABLE OF CONTENTS

1 INTRODUCTION.....	3
1.1 USECASE DESCRIPTIONS.....	3
1.2 USE CASE DIAGRAM (REFINED).....	4
1.3 DOMAIN MODEL .....	4
1.4 SEQUENCE DIAGRAM .....	5
1.4.1. <i>Defining a Sequence diagram</i> .....	6
1.4.2. <i>Basic Sequence Diagram Symbols and Notations</i> .....	6
1.4.3 <i>Example</i> .....	9
1.4.4 <i>Distributing Control Flow in Sequence Diagrams</i> .....	9
1.5 COLLABORATION DIAGRAM.....	12
1.5.1 <i>Contents of Collaboration Diagrams</i> .....	13
1.5.2 <i>Constructs of Collaboration Diagram:</i> .....	13
1.6 OPERATION CONTRACTS.....	14
1.7 DESIGN CLASS DIAGRAM.....	15
1.7.1 <i>Create Initial Design Classes</i> .....	15
1.7.2 <i>Designing Boundary Classes</i> .....	16
1.7.3 <i>Designing Entity Classes</i> .....	16
1.7.4 <i>Designing Control Classes</i> .....	16
1.7.5 <i>Identify Persistent Classes</i> .....	17
1.7.6 <i>Define Class Visibility</i> .....	18
1.7.11 <i>Design Class Relationships</i> .....	21
1.8 DATA MODEL.....	27



## 1 Introduction

Third deliverable is all about the usecase modeling and software design. In the previous deliverable, analysis of the system is completed. So we understand the current situation of the problem domain. Now we are ready to strive for a solution for the problem domain by using object-oriented approach. Following artifacts must be included in this deliverable.

1. Use case description
2. Use case diagram refined
3. Domain Model
4. Sequence Diagram
5. Collaboration Diagram
6. Operation Contracts
7. Design Class Diagram
8. Data Model

Now we discuss these artifacts one by one as follows:

### 1.1 Usecase Description

While technically not part of UML, use case documents are closely related to UML use cases. A use case document is text that captures the detailed functionality of a use case. Description of all use case's are written down. Use case description typically contains the following parts:

#### **Brief description**

Used to describe the overall intent of the use case. Typically, the brief description is only a few paragraphs, but it can be longer or shorter as needed. It describes what is considered the happy path—the functionality that occurs when the use case executes without errors. It can include critical variations on the happy path, if needed.

#### **Preconditions**

Conditionals that must be true before the use case can begin to execute. Note that this means the author of the use case document does not need to check these conditions during the basic flow, as they must be true for the basic flow to begin.

#### **Basic flow**

Used to capture the normal flow of execution through the use case. The basic flow is often represented as a numbered list that describes the interaction between an actor and the system. Decision points in the basic flow branch off to alternate flows. Use case extension points and inclusions are typically documented in the basic flow.

#### **Alternate flows**

Used to capture variations to the basic flows, such as user decisions or error conditions. There are typically multiple alternate flows in a single use case. Some alternate flows rejoin the basic flow at a specified point, while others terminate the use case.



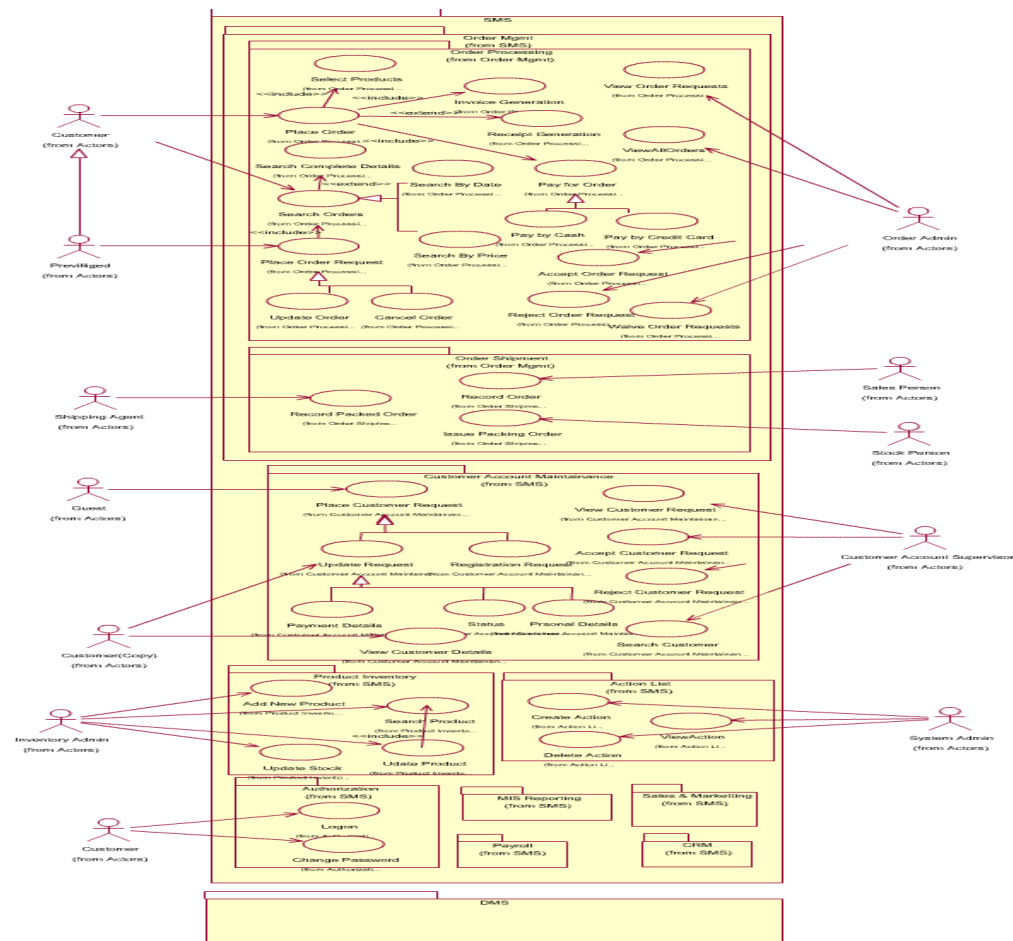
## Post conditions

Conditions that must be true for the use case to completed. Post conditions are typically used by the testers to verify that the realization of the use case is implemented correctly.

## 1.2 Usecase Diagram (refined and updated)

Analysis level usecase diagram is a refined High level use case diagram and is actually the explanation of high level usecases diagram. In this diagram high level usecases are expanded in a way that exhibit how high level usecases will reach to their functionality. Two types of relationships are used in this diagram. Which are:

- Extend
- Include



## 1.3 Domain Model

Domain models represent the set of requirements that are common to systems within a product line. There may be many domains, or areas of expertise, represented in a single



product line and a single domain may span multiple product lines. The requirements represented in a domain model include:

- Definition of scope for the domain
- Information or objects
- Features or use cases, including factors that lead to variation
- Operational/behavioral characteristics

A product line definition will describe the domains necessary to build systems in the product line.

### 1.3.1 What is domain modeling?

According to Rational Unified Process,<sup>®</sup> or RUP,<sup>®</sup> a domain model is a business object model that focuses on "product, deliverables, or events that are important to the business domain." A domain model is an "incomplete" business model, in that it omits individual worker responsibilities. The point of domain modeling is to provide "the big picture" of the interrelationships among business entities in a complex organization. The domain model typically shows the major business entities, and the relationships among the entities. A model that typically does not include the responsibilities people carry is often referred to as a domain model.

It also provides a high-level description of the data that each entity provides. Domain modeling plays a central role in understanding the current environment and planning for the future.

- The typical steps involved in domain modeling are:
- Illustrate meaningful conceptual classes in a real-world problem domain
- Identify conceptual classes or domain objects
- Show associations between them
- Indicate their attributes when appropriate
- Purposely incomplete

## 1.4 Sequence Diagram

A Sequence diagram depicts the sequence of actions that occur in a system. The invocation of methods in each object, and the order in which the invocation occurs is captured in a Sequence diagram. This makes the Sequence diagram a very useful tool to easily represent the dynamic behavior of a system.

A Sequence diagram is two-dimensional in nature. On the horizontal axis, it shows the life of the object that it represents, while on the vertical axis, it shows the sequence of the creation or invocation of these objects.

Because it uses class name and object name references, the Sequence diagram is very useful in elaborating and detailing the dynamic design and the sequence and origin of invocation of objects. Hence, the Sequence diagram is one of the most widely used dynamic diagrams in UML.

### 1.4.1. Defining a Sequence diagram

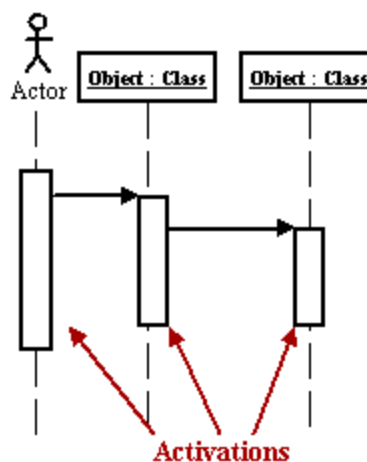
A sequence diagram is made up of objects and messages. Objects are represented exactly how they have been represented in all UML diagrams—as rectangles with the underlined class name within the rectangle.

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

### 1.4.2. Basic Sequence Diagram Symbols and Notations

#### Class roles

Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

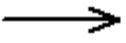
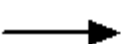





#### Activation

Activation boxes represent the time an object needs to complete a task.

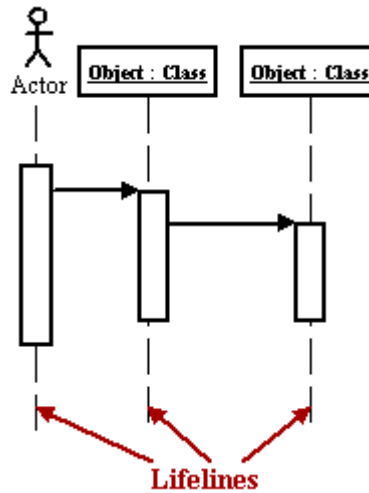
#### Messages

Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

Arrow	Message type
	Simple
	Synchronous
	Asynchronous
	Balking
	Time out

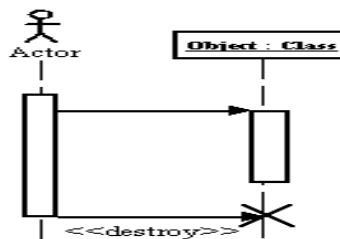
## Lifelines

Lifelines are vertical dashed lines that indicate the object's presence over time.



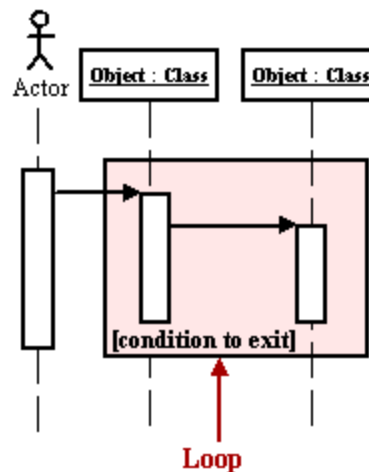
## Destroying Objects

Objects can be terminated early using an arrow labeled "<<destroy>>".



## Loops

A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].





## Objects

An object is shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the object at a particular time. An object symbol is drawn at the head of the lifeline, and shows the name of the object and its class underlined, and separated by a colon: `objectname : classname`

You can use objects in sequence diagrams in the following ways:

- A lifeline can represent an object or its class. Thus, you can use a lifeline to model both class and object behavior. Usually, however, a lifeline represents all the objects of a certain class.
- An object's class can be unspecified. Normally you create a sequence diagram with objects first, and specify their classes later.
- The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.
- Several lifelines in the same diagram can represent different objects of the same class; but, as stated previously, the objects should be named that so you can discriminate between the two objects.
- A lifeline that represents a class can exist in parallel with lifelines that represent objects of that class. The object name of the lifeline that represents the class can be set to the name of the class.

## Actors

Normally an actor instance is represented by the first (left-most) lifeline in the sequence diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them either at the left-most, or the right-most lifelines.

## Messages

A message is a communication between objects that conveys information with the expectation that activity will ensue; in sequence diagrams, a message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. In the case of a message from an object to itself, the arrow may start and finish on the same lifeline. The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often omitted in sequence diagrams, in which the physical location of the arrow shows the relative sequence.

A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message and is not the name of an operation of the receiving object. You can later assign the message by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

## Scripts

Scripts describe the flow of events textually in a sequence diagram.

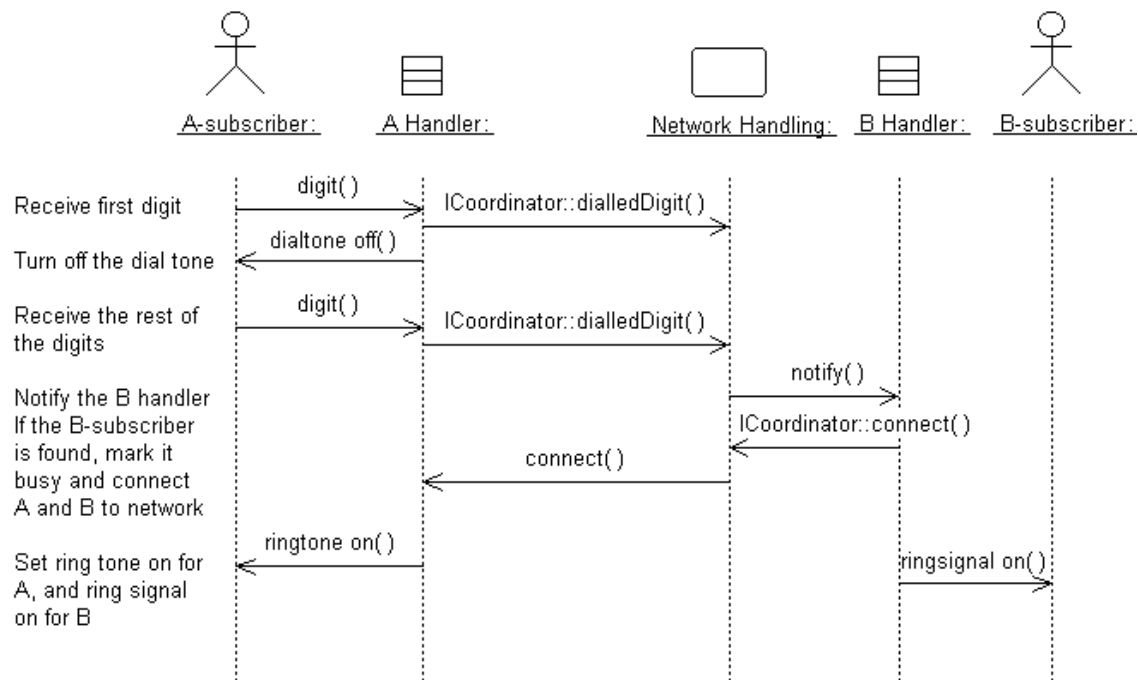
You should position the scripts to the left of the lifelines so that you can read the complete flow from top to bottom (see figure above). You can attach scripts to a certain message, thus ensuring that the script moves with the message.





### 1.4.3 Example

A sequence diagram that describes part of the flow of events of the use case Place Local Call in a simple Phone Switch.

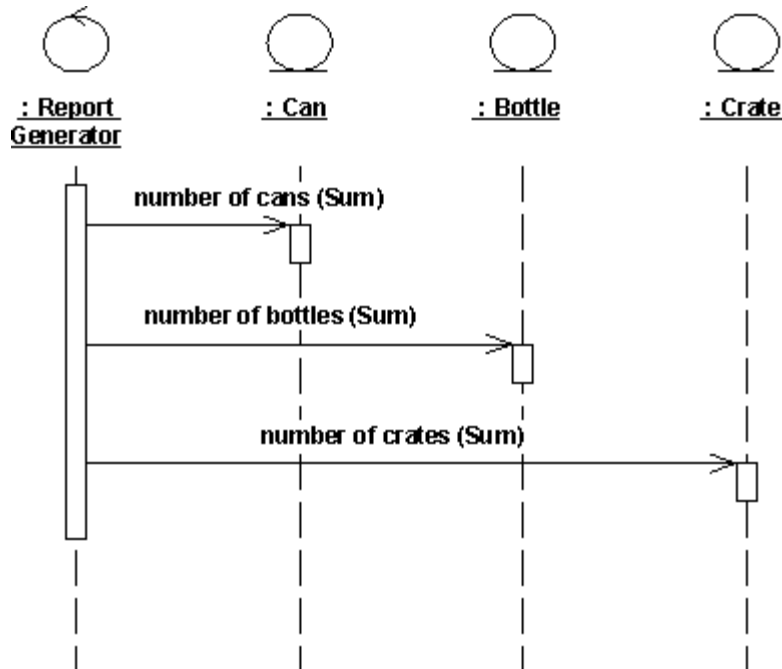


### 1.4.4 Distributing Control Flow in Sequence Diagrams

**Centralized control** of a flow of events or part of the flow of events means that a few objects steer the flow by sending messages to, and receiving messages from other objects. These controlling objects decide the order in which other objects will be activated in the use case. Interaction among the rest of the objects is very minor or does not exist.

#### Example

In the Recycling-Machine System, the use case Print Daily Report keeps track of - among other things - the number and type of returned objects, and writes the tally on a receipt. The Report Generator control object decides the order in which the sums will be extracted and written.



The behavior structure of the use case Print Daily Report is centralized in the Report Generator control object.

This is an example of centralized behavior. The control structure is centralized primarily because the different sub-event phases of the flow of events are not dependent on each other. The main advantage of this approach is that each object does not have to keep track of the next object's tally. To change the order of the sub-event phases, you merely make the change in the control object. You can also easily add still another sub-event phase if, for example, a new type of return item is included. Another advantage to this structure is that you can easily reuse the various sub-event phases in other use cases because the order of behavior is not built into the objects.

Decentralized control arises when the participating objects communicate directly with one another, not through one or more controlling objects.

### Example

In the use case Send Letter someone mails a letter to another country through a post office. The letter is first sent to the country of the addressee. In the country, the letter is sent to a specific city. The city, in turn, sends the letter to the home of the addressee.

The behavior structure of the use case **Send Letter** is decentralized.

The use case behavior is a decentralized flow of events. The sub-event phases belong together. The sender of the letter speaks of "sending a letter to someone." He neither needs nor wants to know the details of how letters are forwarded in countries or cities. (Probably, if someone were mailing a letter within the same country, not all these actions would occur.)

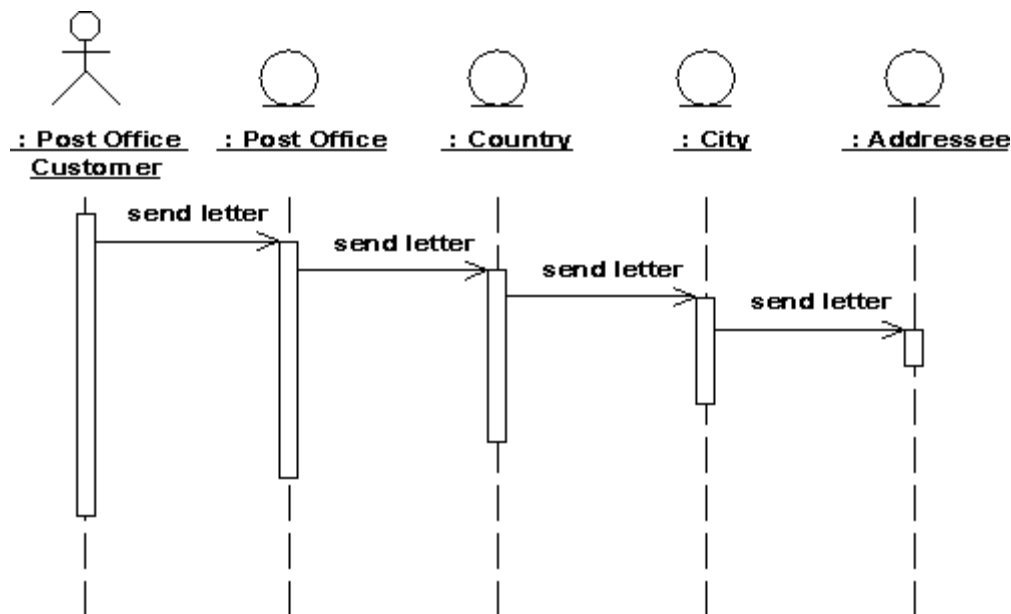


The type of control used depends on the application. In general, you should try to achieve independent objects, that is, to delegate various tasks to the objects most naturally suited to perform them.

A flow of events with centralized control will have a "fork-shaped" sequence diagram. On the other hand, a "stairway-shaped" sequence diagram illustrates that the control-structure is decentralized for the participating objects.

A centralized control structure in a flow of events produces a "fork-shaped" sequence diagram. A decentralized control structure produces a "stairway-shaped" sequence diagram.

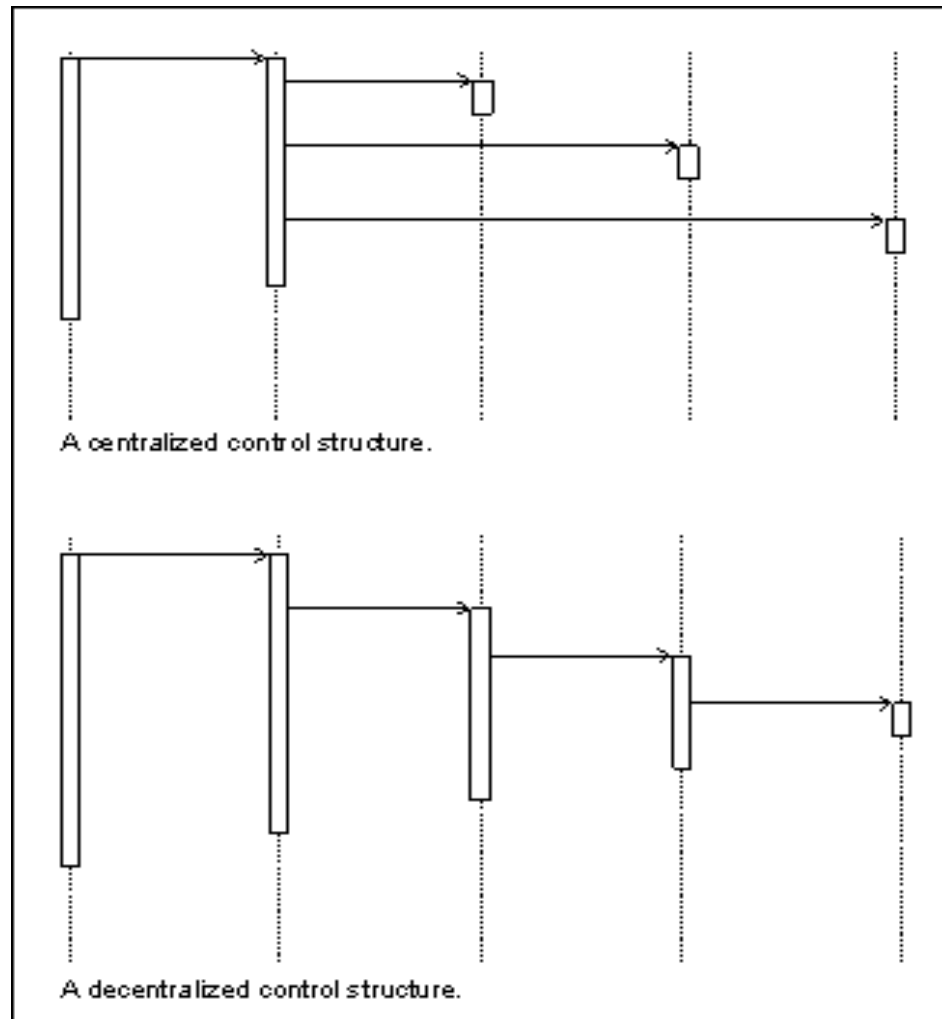
The behavior structure of a use-case realization most often consists of a mix of centralized and decentralized behavior.



A decentralized structure is appropriate:

- If the sub-event phases are tightly coupled. This will be the case if the participating objects:
- Form a part-of or consists-of hierarchy, such as Country - State - City;
- Form an information hierarchy, such as CEO - Division Manager - Section Manager;
- Represent a fixed chronological progression (the sequence of sub-event phases will always be performed in the same order), such as Advertisement - Order - Invoice - Delivery - Payment; or
- Form a conceptual inheritance hierarchy, such as Animal - Mammal - Cat.
- If you want to encapsulate, and thereby make abstractions of, functionality. This is good for someone who always wants to use the whole functionality, because the functionality can become unnecessarily hard to grasp if the behavior structure is centralized.

- A centralized structure is appropriate:
- If the order in which the sub-event phases will be performed is likely to change.
- If you expect to insert new sub-event phases.
- If you want to keep parts of the functionality reusable as separate pieces.



## 1.5 Collaboration Diagram

A collaboration diagram describes a pattern of interaction among objects; it shows the objects participating in the interaction by their links to each other and the messages that they send to each other.

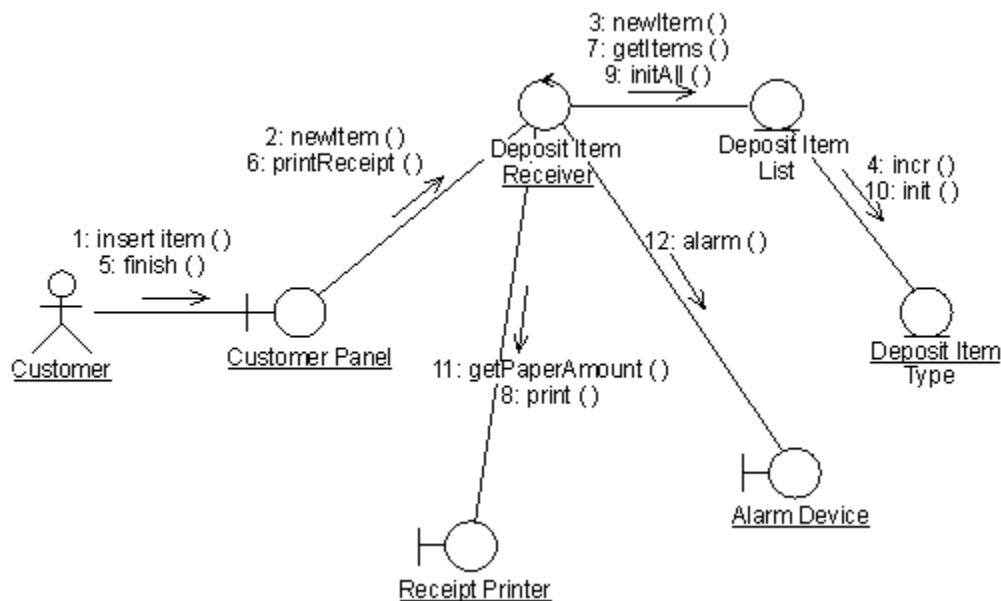
Collaboration diagrams are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaborations are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case. They are the primary source of information used to determining class responsibilities and interfaces.



Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. Collaboration diagrams show the relationships among objects and are better for understanding all the effects on a given object and for procedural design. Because of the format of the collaboration diagram, they tend to be better suited for analysis activities. Specifically, they tend to be better suited to depicting simpler interactions of smaller numbers of objects. As the number of objects and messages grows, the diagram becomes increasingly hard to read. In addition, it is difficult to show additional descriptive information such as timing, decision points, or other unstructured information that can be easily added to the notes in a sequence diagram.

### 1.5.1 Contents of Collaboration Diagrams

You can have objects and actor instances in collaboration diagrams, together with links and messages describing how they are related and how they interact. The diagram describes what takes place in the participating objects, in terms of how the objects communicate by sending messages to one another. You can make a collaboration diagram for each variant of a use case's flow of events.



A collaboration diagram that describes part of the flow of events of the use case Receive Deposit Item in the Recycling-Machine System.

### 1.5.2 Constructs of Collaboration Diagram:

#### Objects

An object is represented by an object symbol showing the name of the object and its class underlined, separated by a colon:

objectname : classname



You can use objects in collaboration diagrams in the following ways:

An object's class can be unspecified. Normally you create a collaboration diagram with objects first and specify their classes later.

The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.

An object's class can itself be represented in a collaboration diagram, if it actively participates in the collaboration.

### **Actors**

Normally an actor instance occurs in the collaboration diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

### **Links**

Links are defined as follows:

A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects.

An object interacts with, or navigates to, other objects through its links to these objects.

A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.

Message flows are attached to links.

### **Messages**

A message is a communication between objects that conveys information with the expectation that activity will ensue. In collaboration diagrams, a message is shown as a labeled arrow placed near a link. This means that the link is used to transport, or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message). The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often used in collaboration diagrams, because they are the only way of describing the relative sequencing of messages.

A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. You can later assign the message by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

## **1.6 Operation Contracts**

A UML Operation contract identifies system state changes when an operation happens. Effectively, it will define what each system operation does. An operation is taken from a system sequence diagram. It is a single event from that diagram. A domain model can be used to help generate an operation contract.

### **Operation Contract Syntax**



Name: appropriateName

Responsibilities: Perform a function

Cross References: System functions and Use Cases

Exceptions: none

Preconditions: Something or some relationship exists

Postconditions: An association was formed

When making an operation contract, think of the state of the system before the action (snapshot) and the state of the system after the action (a second snapshot). The conditions both before and after the action should be described in the operation contract. Do not describe how the action or state changes were done. The pre and post conditions describe state, not actions.

Typical postcondition changes:

- Object attributes were changed.
- An instance of an object was created.
- An association was formed or broken.
- Postconditions are described in the past tense. They declare state changes to the system. Fill in the name, then responsibilities, then postconditions.

## **1.7 Design Class Diagram**

Classes are the work-horses of the design effort—they actually perform the real work of the system. The other design elements—subsystems, packages and collaborations simply describe how classes are grouped or how they interoperate.

Capsules are also stereotyped classes, used to represent concurrent threads of execution in real-time systems. In such cases, other design classes are 'passive' classes, used within the execution context provided by the 'active' capsules. When the software architect and designer choose not to use a design approach based on capsules, it is still possible to model concurrent behavior using 'active' classes.

Active classes are design classes, which coordinate and drive the behavior of the passive classes - an active class is a class whose instances are active objects, owning their own thread of control.

### **1.7.1 Create Initial Design Classes**

Start by identifying one or several (initial) design classes from the domain model, and assign trace dependencies. The design classes created in this step will be refined, adjusted, split and/or merged in the subsequent steps when assigned various "design" properties, such as operations, methods, and a state machine, describing how the analysis class is designed.

Depending on the type of the analysis class (boundary, entity, or control) that is to be designed, there are specific strategies that can be used to create initial design classes.



### 1.7.2 Designing Boundary Classes

The general rule in analysis is that there will be one boundary class for each window, or one for each form, in the user interface. The consequence of this is that the responsibilities of the boundary classes can be on a fairly high level, and need then be refined and detailed in this step.

The design of boundary classes depends on the user interface (or GUI) development tools available to the project. Using current technology, it is quite common that the user interface is visually constructed directly in the development tool, thereby automatically creating user interface classes that need to be related to the design of control and/or entity classes. If the GUI development environment automatically creates the supporting classes it needs to implement the user interface, there is no need to consider them in design - only design what the development environment does not create for you.

Additional input to this work are sketches, or screen dumps from an executable user-interface prototype, that may have been created to further specify the requirements made on the boundary classes.

Boundary classes which represent the interfaces to existing systems are typically modeled as subsystems, since they often have complex internal behavior. If the interface behavior is simple (perhaps acting as only a pass-through to an existing API to the external system) one may choose to represent the interface with one or more design classes. If this route is chosen, use a single design class per protocol, interface, or API, and note special requirements about used standards and so on in the special requirements of the class.

### 1.7.3 Designing Entity Classes

During analysis, entity classes represent manipulated units of information; entity objects are often passive and persistent. In analysis, these entity classes may have been identified and associated with the analysis mechanism for persistence. Performance considerations may force some re-factoring of persistent classes, causing changes to the Design Model, which are discussed jointly between the Database Designer and the Designer.

### 1.7.4 Designing Control Classes

A control object is responsible for managing the flow of a use case and thus coordinates most of its actions; control objects encapsulate logic that is not particularly related to user interface issues (boundary objects), or to data engineering issues (entity objects). This logic is sometimes called application logic, or business logic.

Given this, at least the following issues need to be taken into consideration when control classes are designed:

#### **Complexity:**

Simple controlling or coordinating behavior can be handled by boundary and/or entity classes. As the complexity of the application grows, however, significant drawbacks to this approach surface:

- The use case coordinating behavior becomes imbedded in the UI, making it more difficult to change the system.
- The same UI cannot be used in different use case realizations without difficulty.
- The UI becomes burdened with additional functionality, degrading its performance.





- The entity objects may become burdened with use-case specific behavior, reducing their generality.

To avoid these problems, control classes are introduced to provide behavior related to coordinating flows-of-events

### **Change probability**

If the probability of changing flows of events is low, or the cost is negligible, the extra expense and complexity of additional control classes may not be justified.

### **Distribution and performance**

The need to run parts of the application on different nodes or in different process spaces introduces the need for specialization of design model elements. This specialization is often accomplished by adding control objects and distributing behavior from the boundary and entity classes onto the control classes. In doing this, the boundary classes migrate toward providing purely UI services, and the entity classes toward providing purely data services, with the control classes providing the rest.

### **Transaction management:**

Managing transactions is a classic coordination activity. Absent a framework to handle transaction management, one would have one or more transaction manager classes which would interact to ensure that the integrity of transactions is maintained.

Note that in the latter two cases, if the control class represents a separate thread of control it may be more appropriate to use an active class to model the thread of control.

### **1.7.5 Identify Persistent Classes**

Classes which need to be able to store their state on a permanent medium are referred to as 'persistent'. The need to store their state may be for permanent recording of class information, for back-up in case of system failure, or for exchange of information. A persistent class may have both persistent and transient instances; labeling a class 'persistent' means merely that some instances of the class may need to be persistent.

Identifying persistent classes serves to notify the Database Designer that the class requires special attention to its physical storage characteristics. It also notifies the Software Architect that the class needs to be persistent, and the Designer responsible for the persistence mechanism that instances of the class need to be made persistent.

Because of the need for a coordinated persistence strategy, the Database Designer is responsible for mapping persistent classes into the database, using a persistence framework. If the project is developing a persistence framework, the framework developer will also be responsible for understanding the persistence requirements of design classes. To provide these people with the information they need, it is sufficient at this point to simply indicate that the class (or more precisely, instances of the class) are persistent. Also incorporate any design mechanisms corresponding to persistency mechanisms found during analysis.

### **Example**

The analysis mechanism for persistency might be realized by one of the following design mechanisms:



- In-memory storage
- Flash card
- Binary file
- Database Management System (DBMS)

depending on what is required by the class.

Note that persistent objects may not only be derived from entity classes; persistent objects may also be needed to handle non-functional requirements in general. Examples are persistent objects needed to maintain information relevant to process control, or to maintain state information between transactions.

### 1.7.6 Define Class Visibility

For each class, determine the class visibility within the package in which it resides. A 'public' class may be referenced outside the containing package. A 'private' class (or one whose visibility is 'implementation') may only be referenced by classes within the same package.

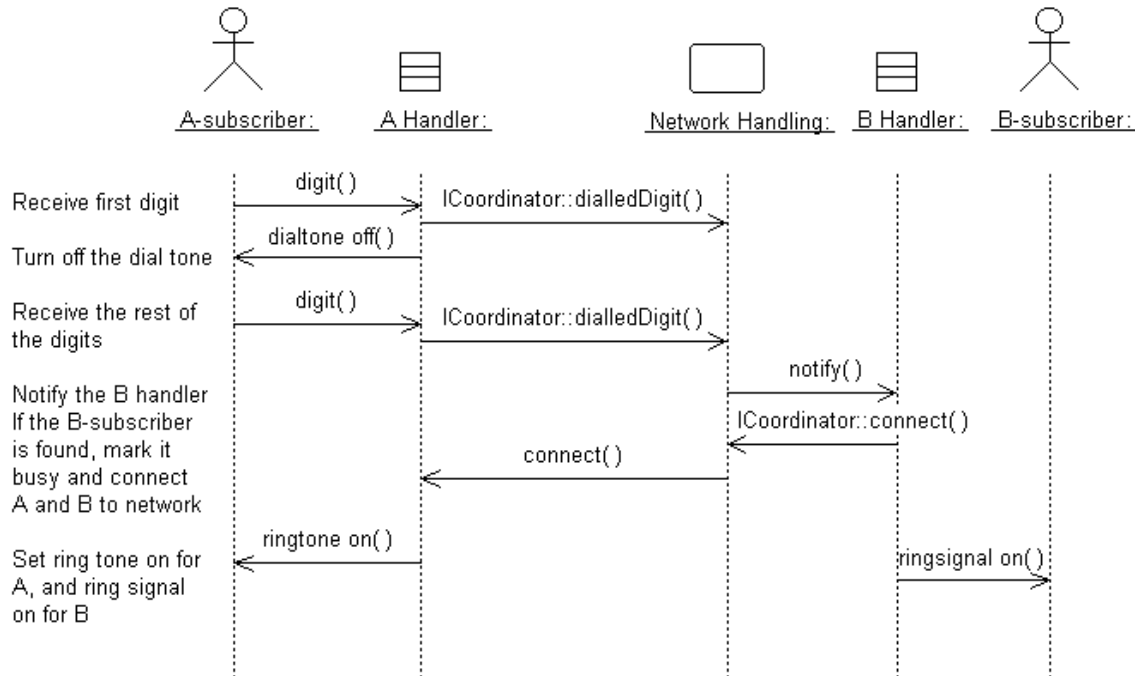
- Define Operations
- Identify Operations
- Name and Describe the Operations
- Define Operation Visibility
- Define Class Operations

### 1.7.7 Identify Operations

To identify Operations on design classes:

- Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
- Study the use-case realizations in the class participates to see how the operations are used by the use-case realizations. Extend the operations, one use-case realization at the time, refining the operations, their descriptions, return types and parameters. Each use-case realization's requirements as regards classes are textually described in the Flow of Events of the use-case realization.
- Study the use case Special Requirements, to make sure that you do not miss implicit requirements on the operation that might be stated there.

Operations are required to support the messages that appear on sequence diagrams because scripts; messages (temporary message specifications) which have not yet been assigned to operations describe the behavior the class is expected to perform. An example sequence diagram is shown below:



Messages form the basis for identifying operations.

Do not define operations, which merely get and set the values of public attributes. These are generally generated by code generation facilities and do not need to be explicitly defined.

### 1.7.8 Name and Describe the Operations

The naming conventions of the implementation language should be used when naming operations, return types, and parameters and their types.

For each operation, you should define the following:

Operation name:

The name should be short and descriptive of the result the operation achieves.

The names of operations should follow the syntax of the implementation language. Example: `find_location` would be acceptable for C++ or Visual Basic, but not for Smalltalk (in which underscores are not used); a better name for all would be `findLocation`.

Avoid names that imply how the operation is performed (example: `Employee.wages()` is better than `Employee.calculateWages()`, since the latter implies a calculation is performed. The operation may simply return a value in a database).

The name of an operation should clearly show its purpose. Avoid unspecific names, such as `getData`, that are not descriptive about the result they return. Use a name that shows exactly what is expected, such as `getAddress`. Better yet, simply let the operation name be the name of the property which is returned or set; if it has a parameter, it sets the property, if it has no parameter it gets the property. Example: the operation `address` returns the address of a Customer, while `address(aString)` sets or changes the address of the Customer. The 'get' and 'set' nature of the operation are implicit from the signature of the operation.

Operations that are conceptually the same should have the same name even if different classes define them, they are implemented in entirely different ways, or they have a



different number of parameters. An operation that creates an object, for example, should have the same name in all classes.

If operations in several classes have the same signature, the operation must return the same kind of result, appropriate for the receiver object. This is an example of the concept of polymorphism, which says that different objects should respond to the same message in similar ways. Example: the operation name should return the name of the object, regardless how the name is stored or derived. Following this principle makes the model easier to understand.

The return type:

The return type should be the class of object that is returned by the operation.

A short description:

As meaningful as we try to make it, the name of the operation is often only vaguely useful in trying to understand what the operation does. Give the operation a short description consisting of a couple of sentences, written from the operation user's perspective.

The parameters. For each parameter, create a short descriptive name, decide on its class, and give it a brief description. As you specify parameters, remember that fewer parameters mean better reusability. A small number of parameters makes the operation easier to understand and hence there is a higher likelihood of finding similar operations. You may need to divide an operation with many parameters into several operations. The operation must be understandable to those who want to use it. The brief description should include the following:

- The meaning of the parameters (if not apparent from their names).
- Whether the parameter is passed by value or by reference
- Parameters which must have values supplied
- Parameters which can be optional, and their default values if no value is provided
- Valid ranges for parameters (if applicable)
- What is done in the operation.
- Which by reference parameters are changed by the operation.

Once you have defined the operations, complete the sequence diagrams with information about which operations are invoked for each message.

### 1.7.9 Define Operation Visibility

For each operation, identify the export visibility of the operation. The following choices exist:

- Public: the operation is visible to model elements other than the class itself.
- Implementation: the operation is visible only within to the class itself.
- Protected: the operation is visible only to the class itself, to its subclasses, or to friends of the class (language dependent)
- Private: the operation is only visible to the class itself and to friends of the class

Choose the most restricted visibility possible which can still accomplish the objectives of the operation. In order to do this, look at the sequence diagrams, and for each message



determine whether the message is coming from a class outside the receiver's package (requires public visibility), from inside the package (requires implementation visibility), from a subclass (requires protected visibility) or from the class itself or a friend (requires private visibility).

#### 1.7.10 Define Class Operations

For the most part, operations are 'instance' operations, that is, they are performed on instances of the class. In some cases, however, an operation applies to all instances of the class, and thus is a class-scope operation. The 'class' operation receiver is actually an instance of a metaclass, the description of the class itself, rather than any specific instance of the class. Examples of class operations include messages, which create (instantiate) new instances, which return all instances of a class, and so on.

To denote a class-scope operation, the operation string is underlined.

A method specifies the implementation of an operation. In many cases, methods are implemented directly in the programming language, in cases where the behavior required by the operation is sufficiently defined by the operation name, description and parameters. Where the implementation of an operation requires use of a specific algorithm, or requires more information than is presented in the operation's description, a separate method description is required. The method describes how the operation works, not just what it does.

- The method, if described, should discuss:
- How operations are to be implemented.
- How attributes are to be implemented and used to implement operations.
- How relationships are to be implemented and used to implement operations.

The requirements will naturally vary from case to case. However, the method specifications for a class should always state:

- What is to be done according to the requirements?
- What other objects and their operations are to be used?
- More specific requirements may concern:
- How parameters are to be implemented.
- Any special algorithms to be used.

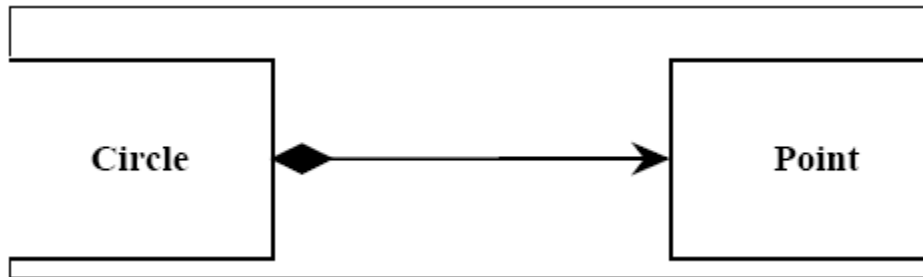
Sequence diagrams are an important source for this. From these it is clear what operations are used in other objects when an operation is performed. A specification of what operations are to be used in other objects is necessary for the full implementation of an operation. The production of a complete method specification thus requires that you identify the operations for the objects involved and inspect the corresponding sequence diagrams.

#### 1.7.11 Design Class Relationships

##### Composition Relationship



Each instance of type *Circle* seems to contain an instance of type *Point*. This is a relationship known as composition. It can be depicted in UML using a class relationship. Figure shows the composition relationship.



The black diamond represents composition. It is placed on the *Circle* class because it is the *Circle* that is composed of a *Point*. The arrowhead on the other end of the relationship denotes that the relationship is navigable in only one direction. That is, *Point* does not know about *Circle*. In UML relationships are presumed to be bidirectional unless the arrowhead is present to restrict them. Had I omitted the arrowhead, it would have meant that *Point* knew about *Circle*. At the code level, this would imply a `#include "circle.h"` within `point.h`. For this reason, I tend to use a lot of arrowheads. Composition relationships are a strong form of containment or aggregation. Aggregation is a whole/part relationship. In this case, *Circle* is the whole, and *Point* is part of *Circle*. However, composition is more than just aggregation. Composition also indicates that the lifetime of *Point* is dependent upon *Circle*. This means that if *Circle* is destroyed, *Point* will be destroyed with it. For those of you who are familiar with the Booch-94 notation, this is the Has-by-value relationship.

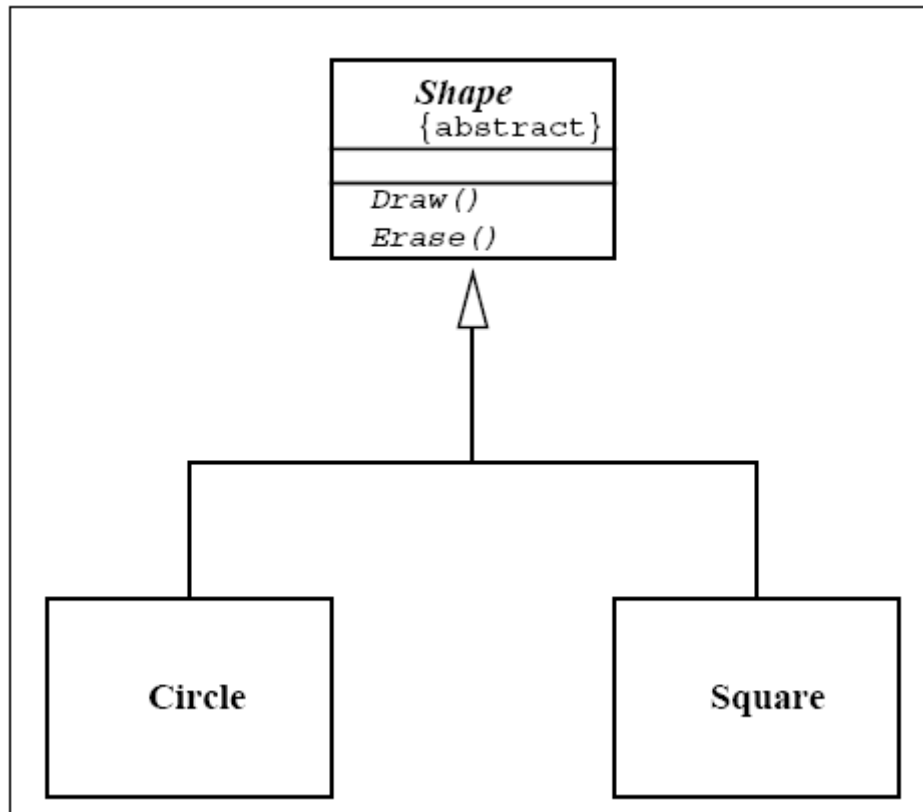
In C++ we would represent this as shown in Listing 1. In this case we have represented the composition relationship as a member variable. We could also have used a pointer so long as the destructor of *Circle* deleted the pointer.

```

class Circle
{
public:
    void SetCenter(const Point&);
    void SetRadius(double);
    double Area() const;
    double Circumference() const;
private:
    double itsRadius;
    Point itsCenter;
};
  
```

### Inheritance Relationship

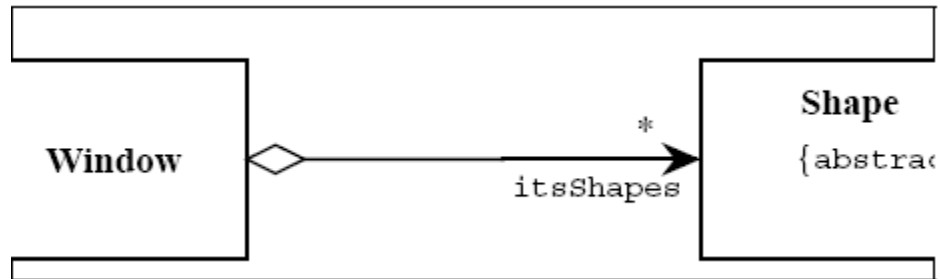
A peculiar triangular arrowhead depicts the inheritance relationship in UML. This arrowhead, that looks rather like a slice of pizza, points to the base class. One or more lines proceed from the base of the arrowhead connecting it to the derived classes. Figure shows the form of the inheritance relationship. In this diagram we see that *Circle* and *Square* both derive from *Shape*. Note that the name of class *Shape* is shown in italics. This indicates that *Shape* is an abstract class. Note also that the operations, *Draw ()* and *Erase ()* are also shown in italics. This indicates that they are pure virtual.



Italics are not always very easy to see. Therefore, as shown in Figure, an abstract class can also be marked with the `{abstract}` property. What's more, though it is not a standard part of UML, I will often write `Draw()=0` in the operations compartment to denote a pure virtual function.

### Aggregation / Association

The weak form of aggregation is denoted with an open diamond. This relationship denotes that the aggregate class (the class with the white diamond touching it) is in some way the “whole”, and the other class in the relationship is somehow “part” of that whole. Figure shows an aggregation relationship. In this case, the *Window* class contains many *Shape* instances. In UML the ends of a relationship are referred to as its “roles”. Notice that the role at the *Shape* end of the aggregation is marked with a “\*”. This indicates that the *Window* contains many *Shape* instances. Notice also that the role has been named. This is the name that *Window* knows its *Shape* instances by. i.e. it is the name of the instance variable within *Window* that holds all the *Shapes*.



Above figure might be implemented in C++ code as under:

```

class Window
{
public:
    //...
private:
    vector<Shape*> itsShapes;
};
  
```

There are other forms of containment that do not have whole / part implications. For example, each window refers back to its parent Frame. This is not aggregation since it is not reasonable to consider a parent Frame to be part of a child Window. We use the association relationship to depict this.

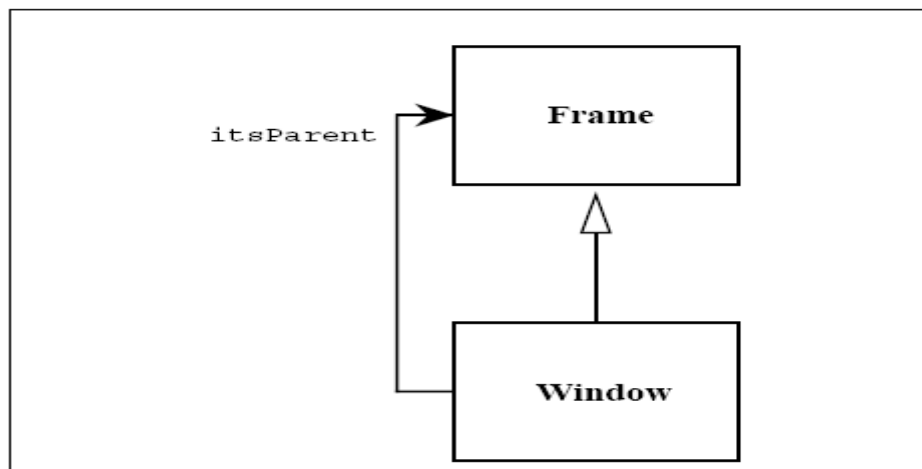


Figure shows how we draw an association. An association is nothing but a line drawn between the participating classes. In Figure 6 the association has an arrowhead to denote that `Frame` does not know anything about `Window`. Once again note the name on the role. This relationship will almost certainly be implemented with a pointer of some kind. What is the difference between an aggregation and an association? The difference is one of implication. Aggregation denotes whole/part relationships whereas associations do not. However, there is not likely to be much difference in the way that the two relationships are implemented. That is, it would be very difficult to look at the code and determine whether





a particular relationship ought to be aggregation or association. For this reason, it is pretty safe to ignore the aggregation relationship altogether. As the amigos said in the UML 0.8 document: “...if you don’t understand [aggregation] don’t use it.” Aggregation and Association both correspond to the Has-by-reference relationship from the Booch-94 notation.

### Dependency

Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments. Consider, for example, the Draw function of the Shape class. Suppose that this function takes an argument of type Drawing Context.

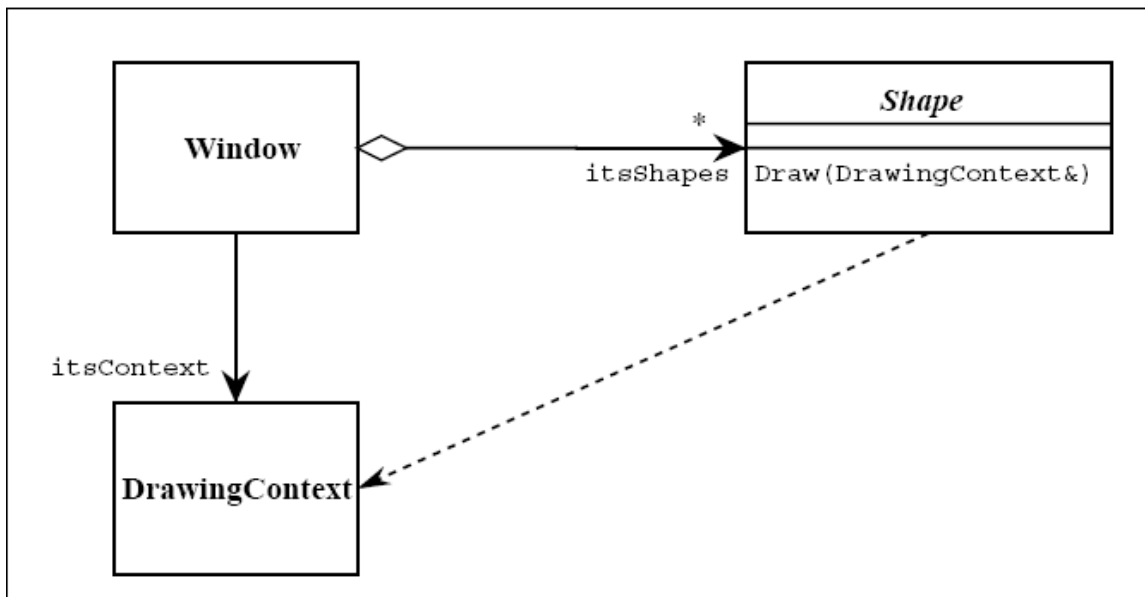


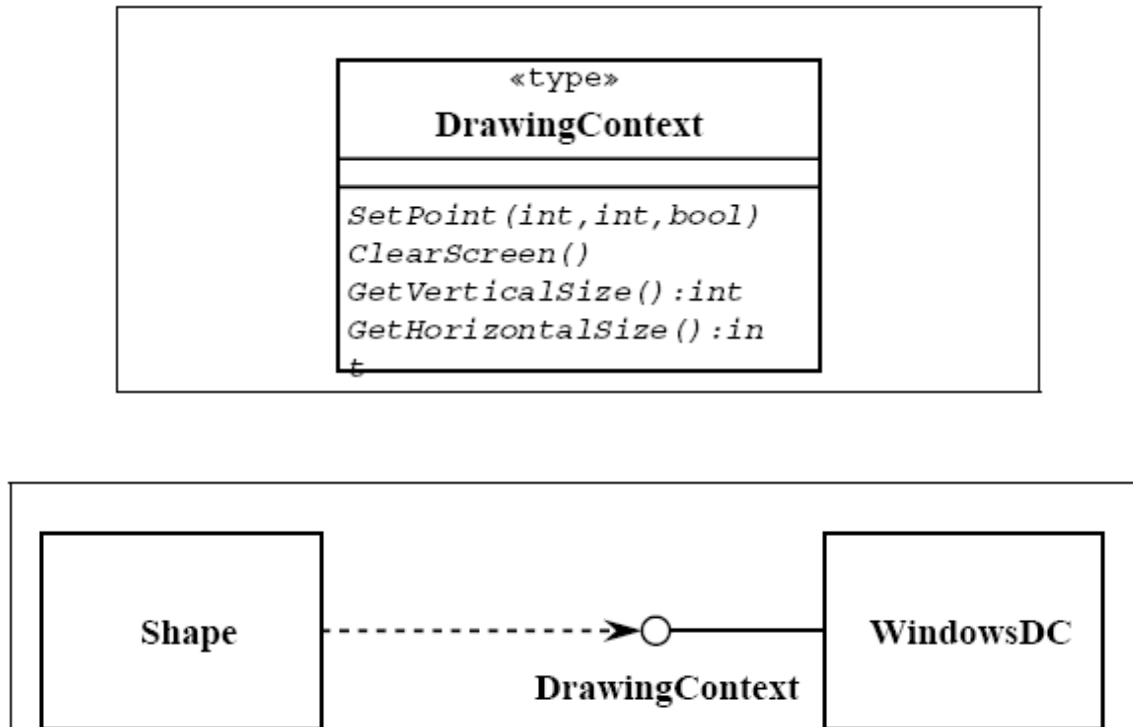
Figure shows a dashed arrow between the Shape class and the DrawingContext class. This is the dependency relationship. In Booch94 this was called a ‘using’ relationship. This relationship simply means that Shape somehow depends upon DrawingContext. In C++ this almost always results in a #include.

### Interfaces

There are classes that have nothing but pure virtual functions. In Java such entities are not classes at all; they are a special language element called an *interface*. UML has followed the Java example and has created some special syntactic elements for such entities. The primary icon for an interface is just like a class except that it has a special denotation called a stereotype. Figure shows this icon. Note the «type» string at the top of the class. The two surrounding characters “«»” are called guillemots (pronounced **Gee-may**). A word or phrase surrounded by guillemots is called a “stereotype”. Stereotypes are one of the mechanisms that can be used to extend UML. When a stereotype is used above the name of a class it indicates that this class is a special kind of class that conforms to a rather rigid specification. The «type» stereotype indicates that the class is an interface.



This means that it has no member variables, and that all of its member functions are pure virtual. UML supplies a shortcut for «type» classes. Figure 9 shows how the “lollypop” notation can be used to represent an interface. Notice that the dependency between `Shape` and `DrawingContext` is shown as usual. The class `WindowsDC` is derived from, or conforms to, the `DrawingContext` interface. This is a shorthand notation for an inheritance relationship between

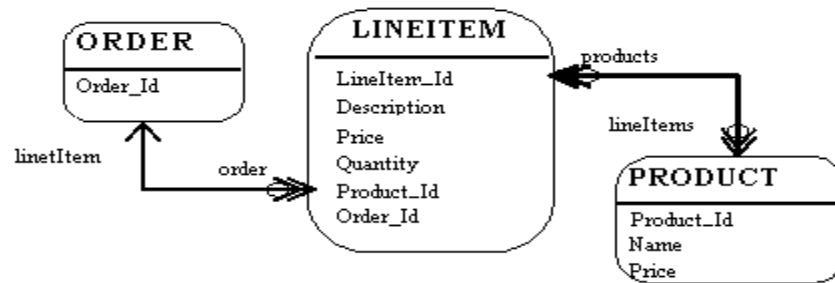


## 1.8 Data Model

The data model is a subset of the implementation model, which describes the logical and physical representation of persistent data in the system.

### The Relational Data Model

The relational model is composed of entities and relations. An entity may be a physical table or a logical projection of several tables also known as a view. The figure below illustrates `LINEITEM` and `PRODUCT` tables and the various relationships between them.



A relational model has the following elements:

An entity has columns. A name and a type identify each column. In the figure above, the LINEITEM entity has the columns LineItem\_Id (the primary key), Description, Price, Quantity, Product\_Id and Order\_Id (the latter two are foreign keys that link the LINEITEM entity to the ORDER and PRODUCT entities).

An entity has records or rows. Each row represents a unique set of information, which typically represents an object's persistent data. Each entity has one or more primary keys. The primary keys uniquely identify each record (for example, Id is the primary key for LINEITEM table).

Support for relations is vendor specific. The example illustrates the logical model and the relation between the PRODUCT and LINEITEM tables. In the physical model relations are typically implemented using foreign key / primary key references. If one entity relates to another, it will contain columns, which are foreign keys. Foreign key columns contain data, which can relate specific records in the entity to the related entity.

Relations have multiplicity (also known as cardinality). Common cardinalities are one to one (1:1), one to many (1:m), many to one (m:1), and many to many (m:n). In the example, LINEITEM has a 1:1 relationship with PRODUCT and PRODUCT has a 0:m relationship with LINEITEM.

### Example

A company has several departments. Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments. At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects. The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.

#### 1. Identify Entities

The entities in this system are Department, Employee, Supervisor and Project. One is tempted to make Company an entity, but it is a false entity because it has only one instance in this problem. True entities must have more than one instance.

#### 2. Find Relationships

We construct the following Entity Relationship Matrix:

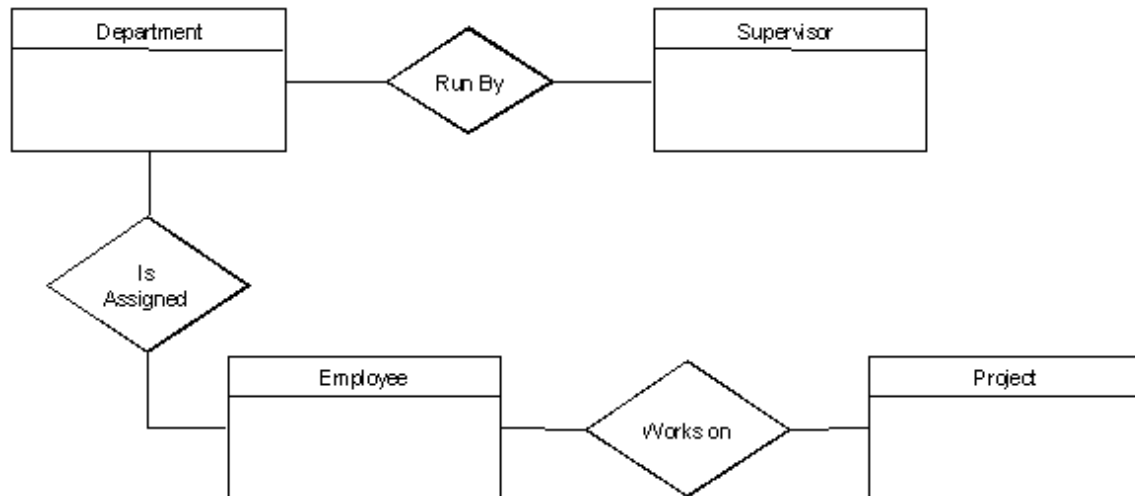
	Department	Employee	Supervisor	Project
Department		is assigned	run by	
Employee	belongs to			works on



Supervisor	runs			
Project		uses		

### 3. Draw Rough ERD

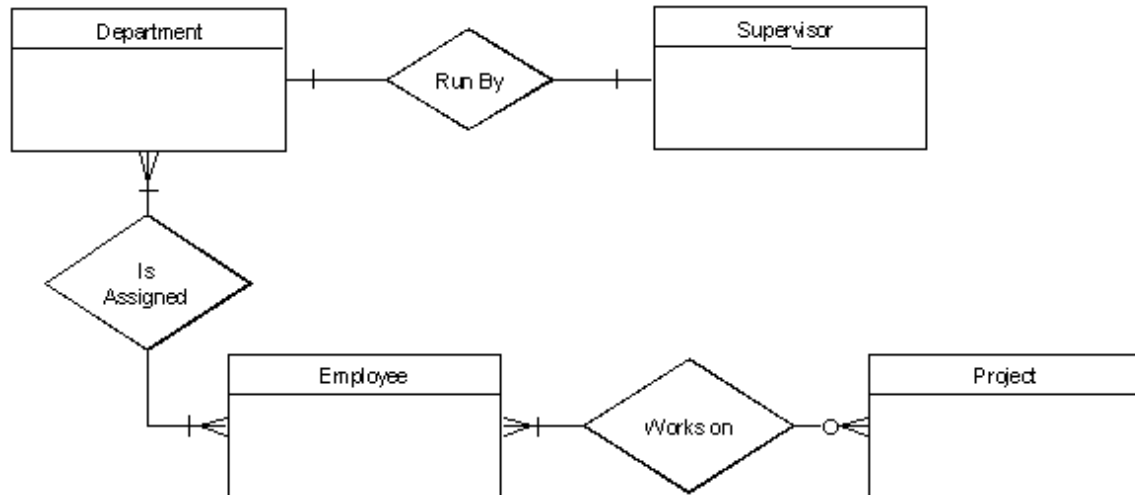
We connect the entities whenever a relationship is shown in the entity Relationship Matrix.



### 4. Fill in Cardinality

From the description of the problem we see that:

- Each department has exactly one supervisor.
- A supervisor is in charge of one and only one department.
- Each department is assigned at least one employee.
- Each employee works for at least one department.
- Each project has at least one employee working on it.
- An employee is assigned to 0 or more projects.

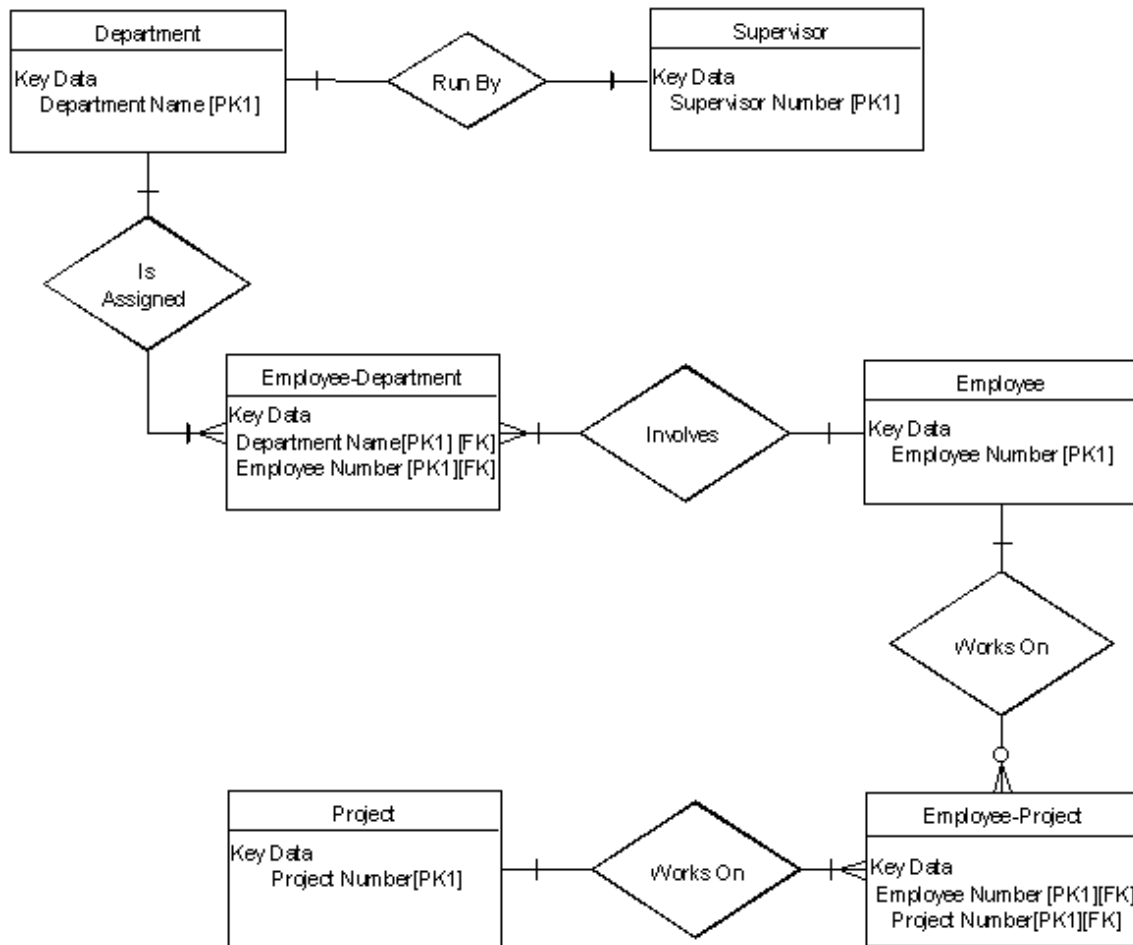


### 5. Define Primary Keys

The primary keys are Department Name, Supervisor Number, Employee Number, Project Number.

### 6. Draw Key-Based ERD

There are two many-to-many relationships in the rough ERD above, between Department and Employee and between Employee and Project. Thus we need the associative entities Department-Employee and Employee-Project. The primary key for Department-Employee is the concatenated key Department Name and Employee Number. The primary key for Employee-Project is the concatenated key Employee Number and Project Number.



### 7. Identify Attributes

The only attributes indicated are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee NUMBER and a unique project number.

### 8. Map Attributes

Attribute	Entity	Attribute	Entity
Department Name	Department	Supervisor Number	Supervisor
Employee Number	Employee	Supervisor Name	Supervisor
Employee Name	Employee	Project Name	Project
		Project Number	Project

### 9. Draw Fully Attributed ERD

