

UNIT II

OBJECT ORIENTED METHODOLOGIES

INTRODUCTION

- Object-oriented methodology is a set of methods, models, and rules for developing systems.
- Modelling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle.

2.1 RUMBAUGH'S OBJECT MODELING TECHNIQUE

- The object modelling technique (OMT) presented by Jim Rumbaugh and his co-workers describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
- OMT is a fast, intuitive approach for identifying and modelling all the objects making up a system.
- Class, attributes, methods, inheritance & association also can be expressed easily.
- The dynamic behavior of objects can be described using the **OMT dynamic model**.
- A process description and consumer-producer relationships can be expressed using **OMT's functional model**.

OMT consists of four phases, which can be performed iteratively:

1. **Analysis.** The results are objects and dynamic and functional models.
2. **System design.** The results are a structure of the basic architecture of the system along with high-level strategy decisions.
3. **Object design.** This phase produces a design document, consisting of detailed objects static, dynamic, and functional models.
4. **Implementation.** This activity produces reusable, extendible, and robust code.

OMT separates modeling into three different parts:

1. An **object model**, presented by the object model and the data dictionary.
2. A **dynamic model**, presented by the state diagrams and event flow diagrams.
3. A **functional model**, presented by data flow and constraints.

THE OBJECT MODEL

- The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations.
- The object diagram contains classes interconnected by association lines.
- Each class represents a set of individual objects.
- The association lines establish relationships among the classes.
- Each association line represents a set of links from the objects of one class to the objects of another class.

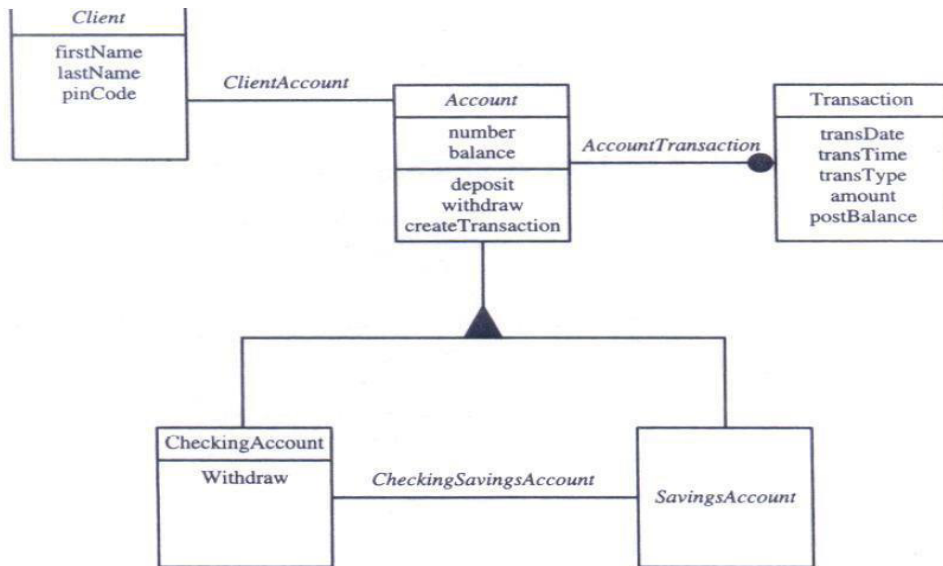


Fig. 2.1. OMT object model of a bank system

Boxes- represents classes, Filled Triangle – represents Specialization, Association between account and transaction represents one to many, Filled Circle – represents many (zero or more). Association between Client and Account represents one to one.

THE OMT DYNAMIC MODEL

- OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions.
- The OMT state transition diagram is a network of states and events (see Fig. 2).
- Each state receives one or more events, at which time it makes the transition to the next state.
- The next state depends on the current state as well as the events.

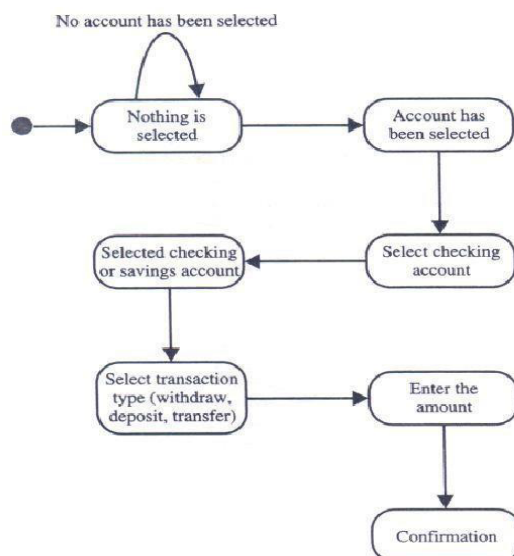
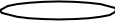





Fig. 2.2. State transition diagram for the bank application user interface. The roundboxes represent states and the arrows represent transitions.

THE OMT FUNCTIONAL MODEL

- The OMT data flow diagram (DFD) shows the flow of data between different processes in a business.

Data flow diagrams use four primary symbols:

1.  The **process** is any function being performed; for example, verify Password or PIN in the ATM system (see Fig .3).
2.  - The **data flow** shows the direction of data element movement; for example, PIN code.
3.  - The **data store** is a location where data are stored; for example, account is a data store in the ATM example.
4.  - An **external entity** is a source or destination of a data element; for example, the ATM card reader.

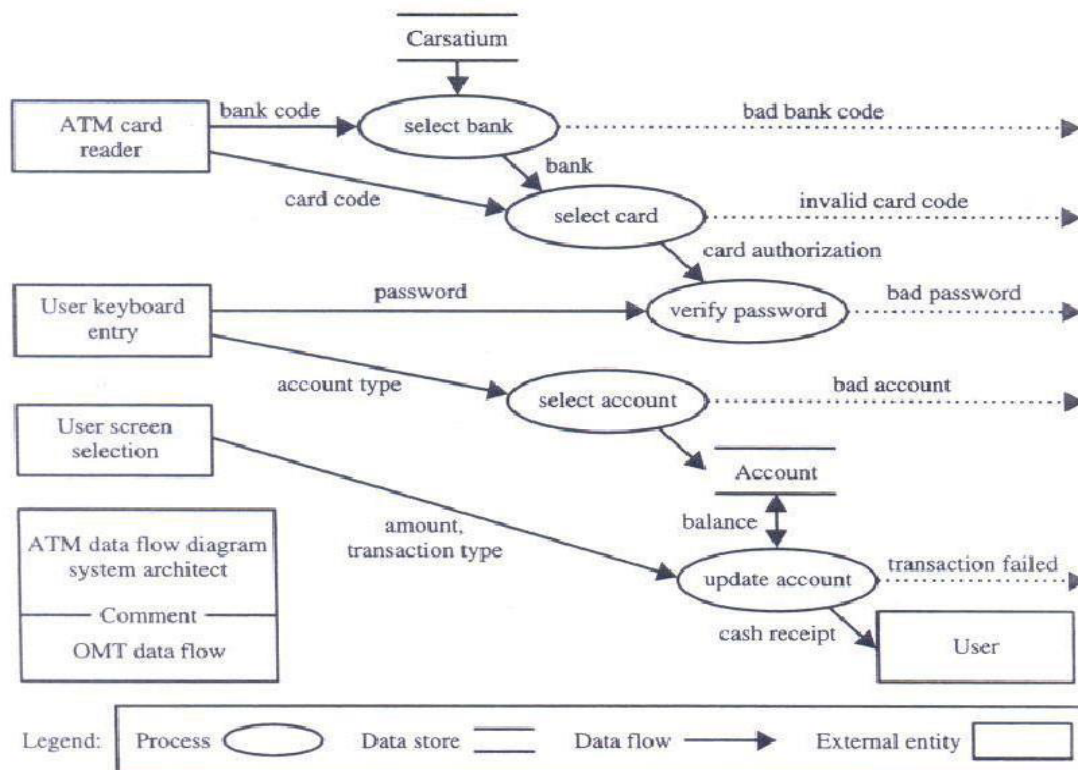


Fig 2.3. OMT DFD of the ATM system.

2.2 BOOCH METHODOLOGY

- The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm.
- The Booch method consists of the following diagrams:
 - Class diagrams
 - Object diagrams
 - State transition diagrams
 - Module diagrams
 - Process diagrams
 - Interaction diagrams
- The Booch methodology prescribes a **macro development process** and a **micro development process**.

THE MACRO DEVELOPMENT PROCESS

- The macro process serves as a controlling framework for the micro process.
- The primary concern of the macro process is technical management of the system.

The macro development process consists of the following steps:

1. **Conceptualization.** During conceptualization,
 - establish the core requirements of the system.
 - establish a set of goals and
 - develop a prototype to prove the concept.
2. **Analysis and development of the model.**
 - use the **class diagram** to describe the roles and responsibilities objects
 - **object diagram** to describe the desired behavior of the system in terms of scenarios
3. **Design or create the system architecture.** In the design phase,
 - use the **class diagram** to decide what classes exist and how they relate to each other.
 - use the **object diagram** to decide what mechanisms are used to regulate how objects collaborate.
 - use the **module diagram** to map out where each class and object should be declared.
 - use the **process diagram** to determine to which processor to allocate a process.
4. **Evolution or implementation.**
 - refine the system through many iterations.
 - Produce a stream of software implementations (or executable releases), each of which is a refinement of the prior one.
5. **Maintenance.** Make localized changes to the system to add new requirements and eliminate bugs.

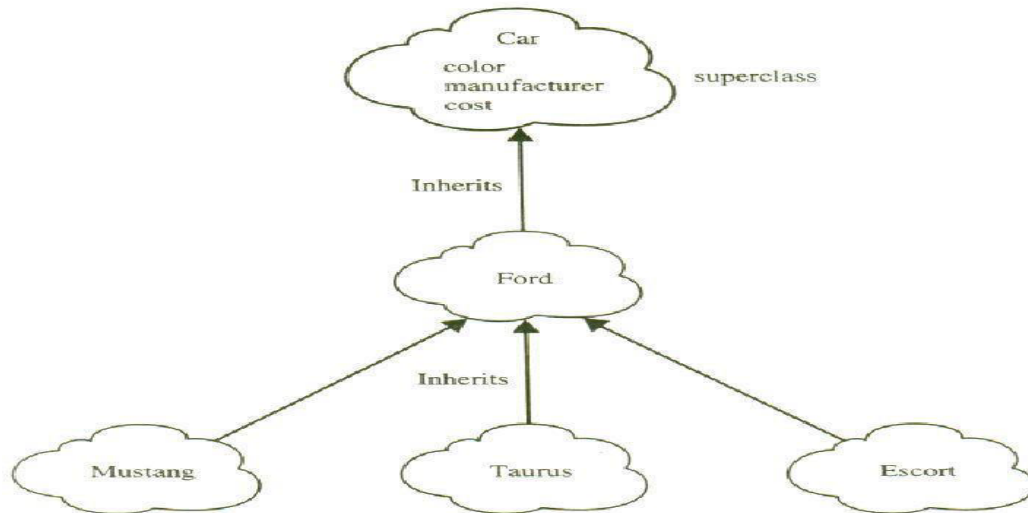


Fig. 2.4. Object modeling using Booch notation

The arrows represent specialization; for example, the class Taurus is subclass of the class Ford.

THE MICRO DEVELOPMENT PROCESS

- Each macro development process has its own micro development processes.
- The micro process is a description of the day-to-day activities by a single or small group of software developers.

The micro development process consists of the following steps:

1. Identify classes and objects.
2. Identify class and object semantics.
3. Identify class and object relationships.
4. Identify class and object interfaces and implementation.

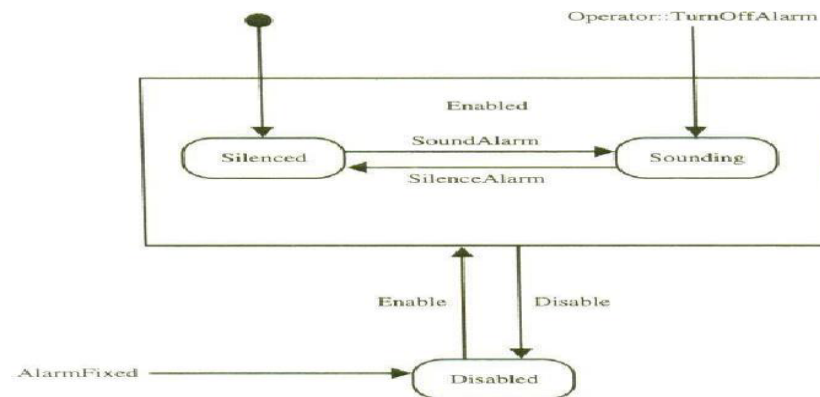


Fig. 2.5. An alarm class state transition diagram with Booch notation.

This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

2.3 THE JACOBSON METHODOLOGIES

- The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire life cycle and stress traceability between the different phases, both forward and backward.
- This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code.
- At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

USE CASES

- Use cases are scenarios for understanding system requirements.
- A use case is an interaction between users and a system.
- The use-case model captures
 - the goal of the user and
 - the responsibility of the system to its users

Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.

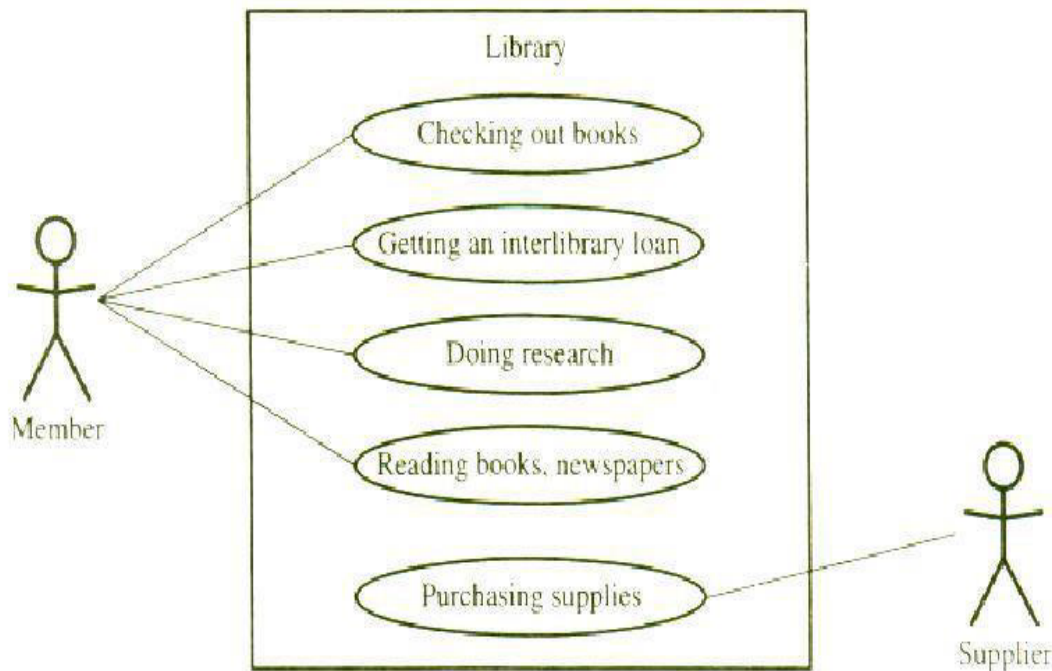


Fig. 2.6. Some Uses of a Library.

The use case description must contain

- How and when the use case begins and ends.
- The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.

- How and when the use case will need data stored in the system or will store data in the system.
Exceptions to the flow of events.
- How and when concepts of the problem domain are handled.

OBJECT-ORIENTED SOFTWARE ENGINEERING: OBJECTORY

- Object-oriented software engineering (OOSE), also called **Objectory**, is a **method** of object-oriented development with the specific aim to fit the development of large, real-time systems.

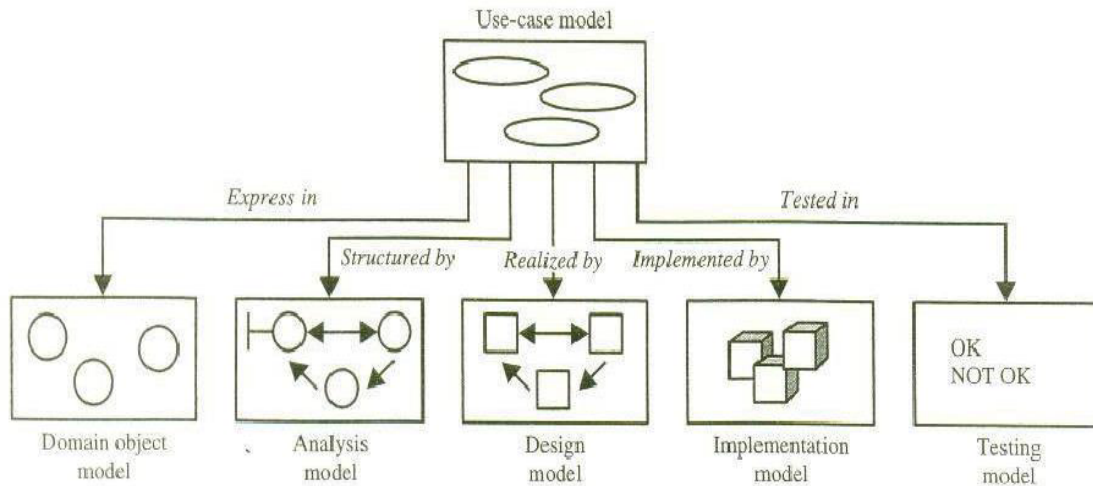


Fig. 2.7. The use case model is considered in every model and phase.

Objectory is built around several different models:

- **Use case-model.** The use-case model defines the outside (actors) and inside (usecase) of the system's behavior.
- **Domain object model.** The objects of the "real" world are mapped into the domain object model.
- **Analysis object model.** The analysis object model presents how the source code (implementation) should be carried out and written.
- **Implementation model.** The implementation model represents the implementation of the system.
- **Test model.** The test model constitutes the test plans, specifications, and reports.

OBJECT-ORIENTED BUSINESS ENGINEERING

- Object-oriented business engineering (OOBE) is object modelling at the enterprise level.
 - Analysis phase
 - Design and implementation Phase
 - Testing Phase

1. **Analysis phase.** It defines the system to be built in terms of
 - a. problem-domain object model,

- b. the requirements model, and
 - c. the analysis models.
- 2. **Design and implementation phases.** This includes **factors** such as
 - a. Database Management System (DBMS),
 - b. distribution of process,
 - c. constraints due to the programming language,
 - d. available component libraries, and
 - e. incorporation of graphical user interface tools
- 3. **Testing phase.**
 Jacobson describes several testing levels and techniques.
 The levels include
 - unit testing,
 - integration testing, and
 - system testing.

2.4 PATTERNS

Definition:

A pattern is [an] instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

- A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints.

proto-pattern.

- A "pattern in waiting," which is not yet known to recur, sometimes is called a **proto-pattern**.

A good pattern will do the following:

- **It solves a problem.** Patterns capture solutions, not just abstract principles or strategies.
- **It is a proven concept.** Patterns capture solutions with a track record, not theories or speculation.
- **The solution is not obvious.** The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
- **It describes a relationship.** Patterns do not just describe modules, but describe deeper system structures and mechanisms.
- **The pattern has a significant human component.** All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Most pattern encompass all aspects of S/W Engg including

- Development organization
- The s/w development process
- Project Planning
- Requirement engineering
- s/w configuration management

Generative and Non generative Patterns

- **Generative patterns** describe how to generate something

- **Nongenerative patterns** are static and passive:
 - They describe how to reproduce them.

Patterns Template

- Every pattern must be expressed "in the form of a rule [template] which establishes a relationship between a context and a configuration"

Essential components should be clearly recognizable on reading a pattern:

- **Name.** A meaningful name.
- **Problem.** A statement of the problem
- **Context.** The preconditions
- **Forces.** how they interact or conflict with one another and with the goals
- **Solution.** how to realize the desired outcome?
- **Examples.** how the pattern is applied to and transforms that context. One or more sample applications
- **Resulting context.** The state of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context.
- **Rationale.** how it works, why it works, and why it is "good."
- **Related patterns.** Related patterns often share common forces.
- **Known uses.** The known occurrences

ANTIPATTERNS

- A pattern represents a "best practice," whereas an antipattern represents "worst practice" or a "lesson learned."

Anti-patterns come in two varieties:

- Those describing a bad solution to a problem that resulted in a bad situation.
- Those describing how to get out of a bad situation and how to proceed from there to a good solution.

Anti-patterns are valuable because often it is just as important to see and understand bad solutions as to see and understand good ones.

Capturing Patterns

- Writing good patterns is very difficult
- A pattern should help its users comprehend existing systems, customize systems to fit user needs, and construct new systems.
- The process of looking for patterns to document is called **pattern mining (or sometimes reverse architecting)**.

2.5. FRAMEWORKS

A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development.

A definition of an object-oriented software framework is given by Gamma:

- A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
- A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations.
- A developer customizes a framework to a particular application by subclassing and composing instances of framework classes.
- The framework captures the design decisions that are common to its application domain.
- A single framework typically encompasses several design patterns.

In fact, **a framework can be viewed as the implementation of a system of design patterns.**

Differences between frameworks and design patterns:

- *. A framework is executable software, whereas design patterns represent knowledge and experience about software.
- *. Frameworks are of a physical nature, while patterns are of a logical nature.
- *. Frameworks are the physical realization of one or more software pattern solution; patterns are the instructions for how to implement those solution.

Gamma et al. describe **the major differences between design patterns and frameworks as follows:**

Design patterns are more abstract than frameworks.

Design patterns are smaller architectural elements than frameworks.

Design patterns are less specialized than frameworks.

2.6 THE UNIFIED APPROACH

- The unified approach (UA) used to describe, model, and document the software development process.
- The idea behind the UA is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and system development.
- The unified approach to software development revolves around the following processes and concepts (see Fig.8). The processes are:
 - Use-case driven development
 - Object-oriented analysis
 - Object-oriented design
 - Incremental development and prototyping
 - Continuous testing

the methods and technology employed include

1. Unified modeling language used for modeling.
2. Layered approach.

3. Repository for object-oriented system development patterns and frameworks.
4. Component based development.

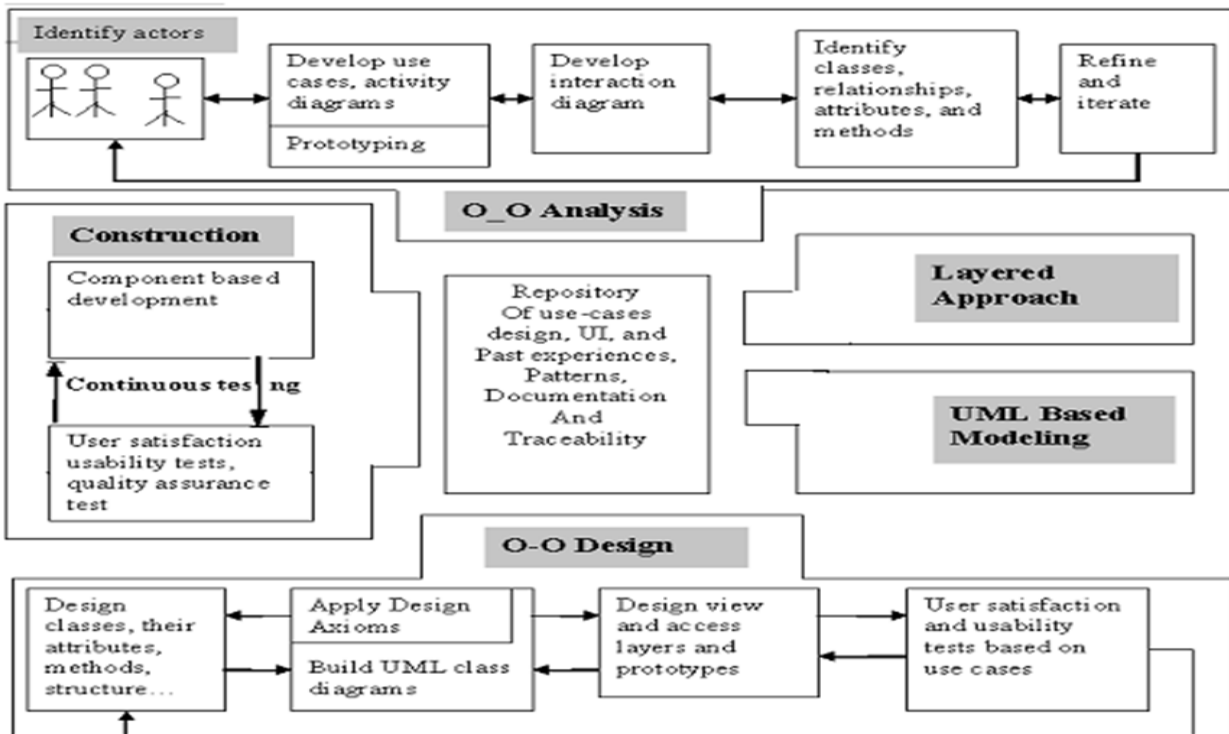


Fig. 2.8. The Process and components of the unified approach

OBJECT-ORIENTED ANALYSIS

- Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements.
- The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system.
- This is accomplished by constructing several models of the system.
- These models concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the user's perspective rather than that of the machine.

OOA Process consists of the following Steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

OBJECT-ORIENTED DESIGN

OOD Process consists of:

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterate and refine the design

ITERATIVE DEVELOPMENT AND CONTINUOUS TESTING

You must iterate and reiterate until, eventually, you are satisfied with the system.

1. MODELING BASED ON THE UNIFIED MODELING LANGUAGE

The UA uses the UML to describe and model the analysis and design phases of system development.

The UA Proposed Repository

In a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format.

The **advantage of repositories** is that for reuse.

2. The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known as **two-layered architecture**: interface and data (see Fig). In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens

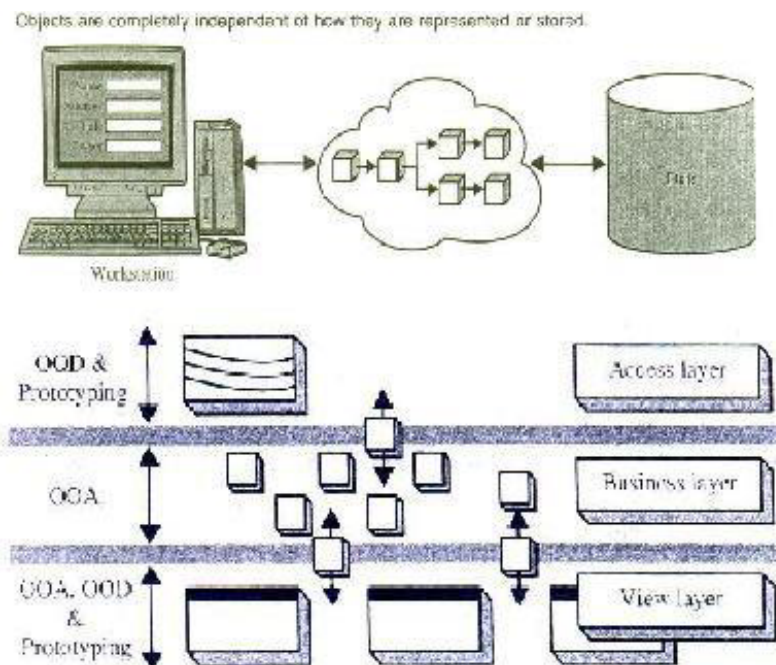


Fig. 2.9. The Layered Approach

The **three-layered approach** consists of

1. a view or user interface layer,
2. a business layer, and
3. an access layers

The Business Layer The business layer contains all the objects that represent the business (both data and behavior). This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist. Most modern object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.

The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes. When creating the business layer, however, it is important to keep in mind a couple of things. These objects should not be responsible for the following:

Displaying details. Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.

Data access details. Business objects also should have no special knowledge of "where they come from." It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object-oriented analysis. A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. For example, a customer could have more than one account or an order could be aggregated from one or more-line items. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability. An individual business object can appear in different business models. Business models also incorporate control objects that direct their processes. The business objects are identified during the object oriented analysis. Use cases can provide a wonderful tool to capture business objects.

The User Interface (View) Layer: The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The user interface layer also is called the view layer. This layer typically is responsible for two major aspects of the applications:

Responding to user interaction. The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process; remember, the business logic does not exist here, just the knowledge of which message to send to which business object.

Displaying business objects. This layer must paint the best possible picture of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.

The Access layer: The access layer contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database mainframe, internet or

file. The Access layer has 2 major responsibilities.

1. **Translate request:** The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
2. **Translate results:** the access Layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back up into the business layer.

Access objects are identified during object-oriented design.

2.9 UNIFIED MODELING LANGUAGE

A **model** is an abstract representation of a system, constructed to understand the system prior to building or modifying it. Most of the modeling techniques involve graphical languages.

Modeling frequently is used during many of the phases of the software life cycle, such as analysis, design, and implementation. For example, Objectory is built around several different models:

Use-case model. The use-case model defines the outside (actors) and inside (usecase) of the system's behavior.

Domain object model. Objects of the "real" world are mapped into the domain object model.

Analysis object model. The analysis object model presents how the source code (i.e., the implementation) should be carried out and written.

Implementation model. The implementation model represents the implementation of the system.

Test model. The test model constitutes the test plans, specifications, and reports. Modeling is an iterative process.

Static or Dynamic Models

<i>Static Model</i>	<i>Dynamic Model</i>
<ul style="list-style-type: none">• A static model can be viewed as "snapshot" of a system's parameters at rest or at a specific point in time.• The classes' structure and their relationships to each other frozen in time are examples of static models.	<ul style="list-style-type: none">• Is a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time.• For example, an order interacts with inventory to determine product availability.

WHY MODELING?

Building a model for a software system prior to its construction is as essential as having a blueprint for building a large building. Good models are essential for communication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Many other factors add to a project's success, but having a rigorous modeling language is essential. A modeling language must include Model elements-fundamental modeling concepts and semantics.

Turban cites the following advantages of modeling:

1. Models make it easier to express complex ideas. For example, an architect builds a model to communicate ideas more easily to clients.
2. The main reason for modelling is the reduction of complexity. Models reduce complexity by separating those aspects that are unimportant from those that are important. Therefore, it makes complex situations easier to understand.
3. Models enhance and reinforce learning and training.
4. The cost of the modelling analysis is much lower than the cost of similar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulating a real system.

key ideas regarding modeling:

- A model is rarely correct on the first try.
- Always seek the advice and criticism of others. You can improve a model by reconciling different perspectives.
- Avoid excess model revisions, as they can distort the essence of your model.

What Is the UML?

The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as object constraint language (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system.

The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

What it is/isn't? Is NOT

- A process
- A formalism

Is

- A way to describe your software

- more precise than English
- less detailed than code

What is UML Used For?

- Trace external interactions with the software
- Plan the internal behavior of the application
- Study the software structure
- View the system architecture
- Trace behavior down to physical components

The primary goals in the design of the UML were as follows:

1. Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

UML DIAGRAMS

The UML defines nine graphical diagrams:

1. Class diagram (static)
2. Use-case diagram
3. Behavior diagrams (dynamic):
 - 3.1. Interaction diagram:
 - 3.1.1. Sequence diagram
 - 3.1.2. Collaboration diagram
 - 3.2. State chart diagram
 - 3.3. Activity diagram
4. Implementation diagram:
 - 4.1. Component diagram
 - 4.2. Deployment diagram

UML CLASS DIAGRAM

The UML **class diagram**, also referred to as **object modelling**, is the main static analysis diagram. These diagrams show the static structure of the model.

A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents; for example, the things that exist (such as classes), their internal structures, and their relationships to other classes.

Class diagrams do not show temporal information, which is required in dynamic modeling.

- A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them.
- A graphical representation of a static view on declarative static elements.
- A central modelling technique that runs through nearly all object-oriented methods.
- The richest notation in UML.
- A class diagram shows the existence of classes and their relationships in the logical view of a system

Class Notation: Static Structure

A class is drawn as a rectangle with three components separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes, are in the middle compartment, and the bottom compartment holds a list of operations

Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it.

The class name and other properties should be displayed in up to three sections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

Essential Elements of a UML Class Diagram

- Class
- Attributes
- Operations
- Relationships
 - Associations
 - Generalization
- Dependency
- Realization

Constraint Rules and Notes

A class is the description of a set of objects having similar attributes, operations, relationships and behavior.

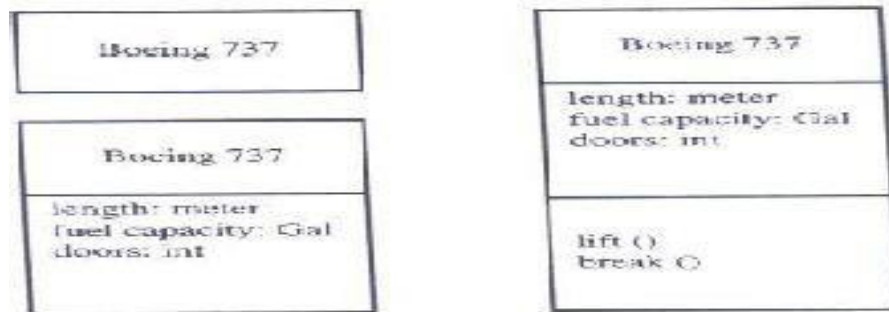


Fig. 2.12. In class notation, either or both the attributes and operation compartments may be suppressed.

Attributes

- Classes have attributes that describe the characteristics of their objects.
- Attributes are atomic entities with no responsibilities.
- Attribute syntax (partial):
 - o [visibility] name [: type] [= default Value]
- Class scope attributes are underlined

Visibility

- Visibility describes whether an attribute or operation is visible and can be referenced from classes other than the one in which they are defined.
- Language dependent
 - o Means different things in different languages
- UML provides four visibility abbreviations: + (public) – (private) # (protected) ~ (package)

Object Diagram

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.

UML modeling elements in class diagrams

- Classes and their structure, association, aggregation, dependency, and inheritance relationships
- Multiplicity and navigation indicators, etc.

Class Interface Notation

Class interface notation is used to describe the externally visible behavior of a class; for example, an operation with public visibility. Identifying class interfaces is a design activity of object-oriented system development.

The UML notation for an interface is a small circle with the name of the interface

connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations.

For example, a Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Fig13. with UML class interface notation.



Fig. 2.13. Interface notation of a Class

Binary Association Notation

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. The association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name. The end of an association, where it connects to a class, is called the **association role** (see Fig. 14).

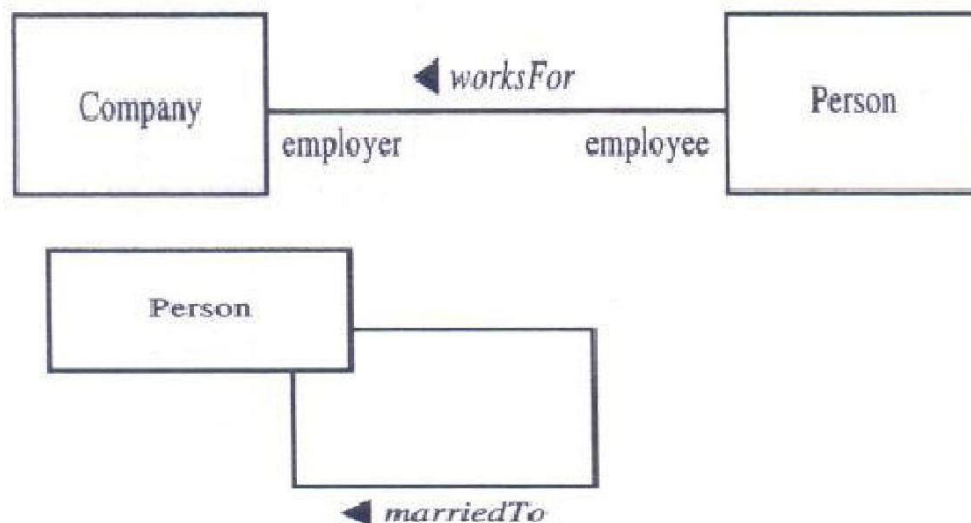


Fig. 2.14. Association Notation.

Association Role

A simple association- **binary association**-is drawn as a solid line connecting two class symbols. The end of an association, where it connects to a class, shows the association role. The role is part of the association, not part of the class. Each association has two or more roles to which it is connected.

In above Fig14. the association worksfor connects two roles, employee and employer. A Person is an employee of a Company and a Company is an employer of a Person.

The UML uses the term association navigation or navigability to specify a role affiliated with each end of an association relationship. An arrow may be attached to the end of the path

to indicate that navigation is supported in the direction of the class pointed to. An arrow may be attached to neither, one, or both ends of the path. In particular, arrows could be shown whenever navigation is supported in a given direction.

In the UML, association is represented by an open arrow, as represented in Fig.15. Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

Association notation.



Fig. 2.15. Association Notation

In this example, the association is navigable in only one direction, from the BankAccount to Person, but not the reverse. This might indicate a design decision, but it also might indicate an analysis decision, that the Person class is frozen and cannot be extended to know about the BankAccount class, but the BankAccount class can know about the Person class.

Qualifier

A **qualifier** is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the account#. The account# is the qualifier of this association. (Fig 16)

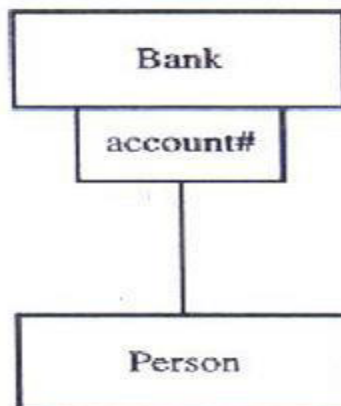


Fig. 2.16. Association Qualifier.

A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle (see above Fig).

Multiplicity

Multiplicity specifies the range of allowable associated classes. It is given for roles within associations, parts within compositions, repetitions, and other purposes. A multiplicity specification is shown as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in this format (see Fig 17):
lower bound. upper bound.

The terms lower bound and upper bound are integer values, specifying the range of integers including the lower bound to the upper bound. The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer range contains the single values.

For example, 0..10..*

1..3, 7..10, 15, 19..*

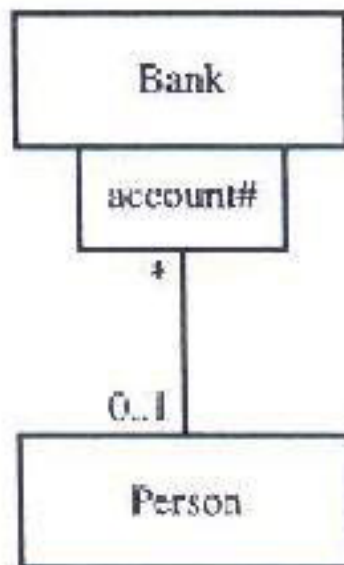


Fig. 2.17. Association Qualifier and its multiplicity.

OR Association

An **OR association** indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string {or} labeling the dashed line (see Fig 18). In other words, any instance of the class may participate in, at most, one of the associations at one time.

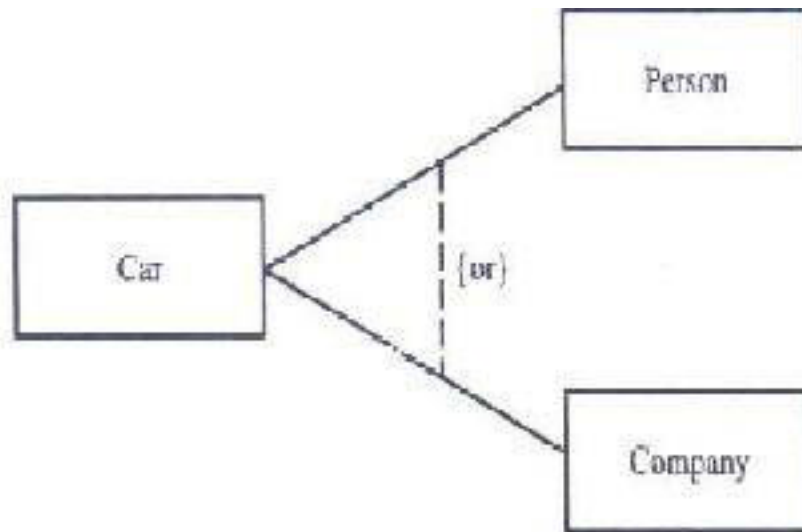


Fig. 2.18. An OR association notation. A car may associate with a person or a company.

Association Class

An **association class** is an association that also has class properties. An association class is shown as a class symbol attached by a dashed line to an association path. The name in the class symbol and the name string attached to the association path are the same (see Fig 19). The name can be shown on the path or the class symbol or both. If an association class has attributes but no operations or other associations, then the name may be displayed on the association path and omitted from the association class to emphasize its "association nature." If it has operations and attributes, then the name may be omitted from the path and placed in the class rectangle to emphasize its "class nature."

Association class:

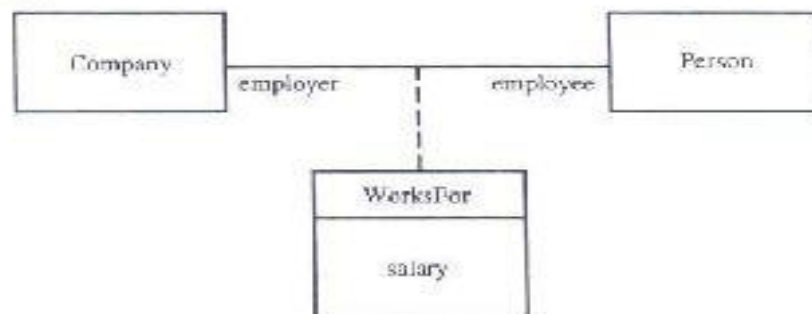


Fig. 2.19. Association Class

N-Ary Association

An n-ary association is an association among more than two classes. Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association.

An n-ary association is shown as a large diamond with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. The role attachment may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted. An association class symbol may be attached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation, or associations. The example depicted in Fig 20 shows the grade book of a class in each semester.

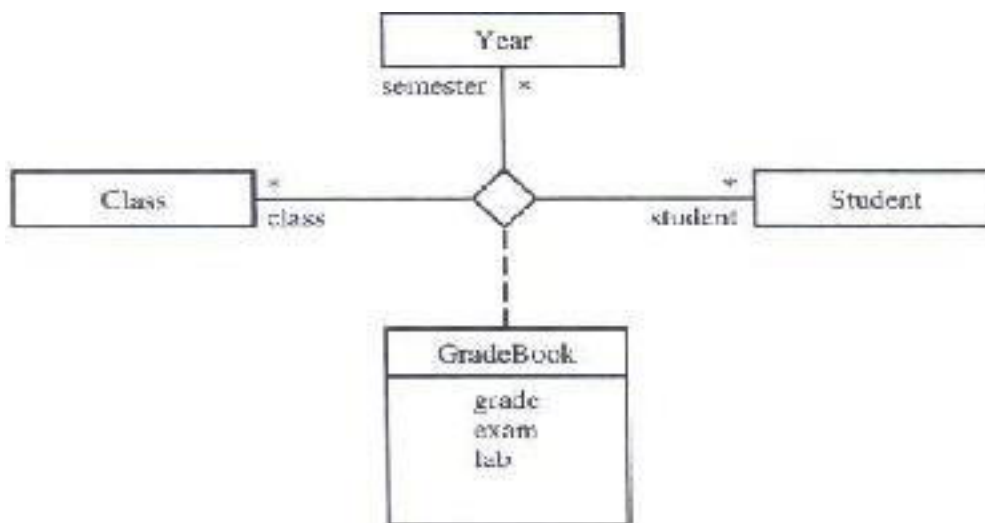


Fig. 2.20. An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab.

Aggregation and Composition (a.part.of)

Aggregations are a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, and it need not be presented at all (see Fig 21).

Composition, also known as the **apart-of**, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as **a part-whole relationship**. The UML notation for composition is a solid diamond at the end of a path. Alternatively, the UML provides a graphically nested form that, in many cases, is more convenient for showing composition (see Fig 22).

Parts with multiplicity greater than one may be created after the aggregate itself but, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.



Fig. 2.21. Association Path

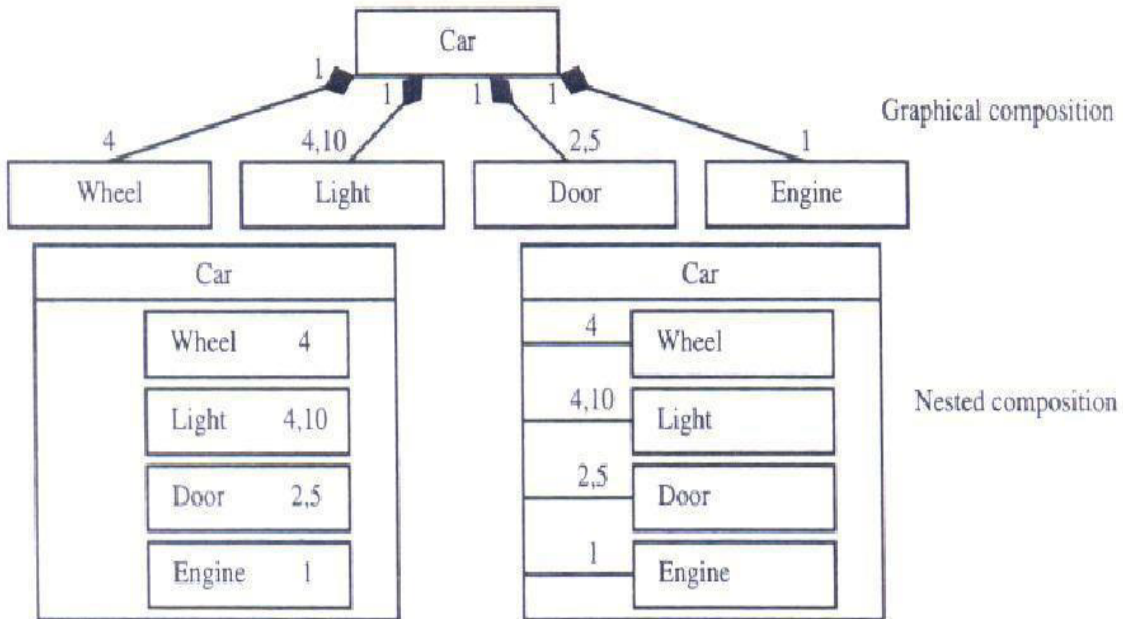


Fig. 2.22. Different ways to show Composition.

Generalization

Generalization is the relationship between a more general class and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowhead at the superclass end (see Fig 23). The UML allows a **discriminator label** to be attached to a generalization of the superclass.

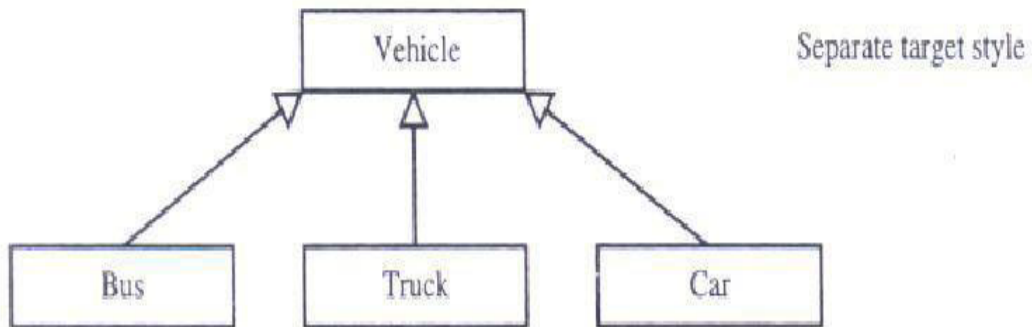
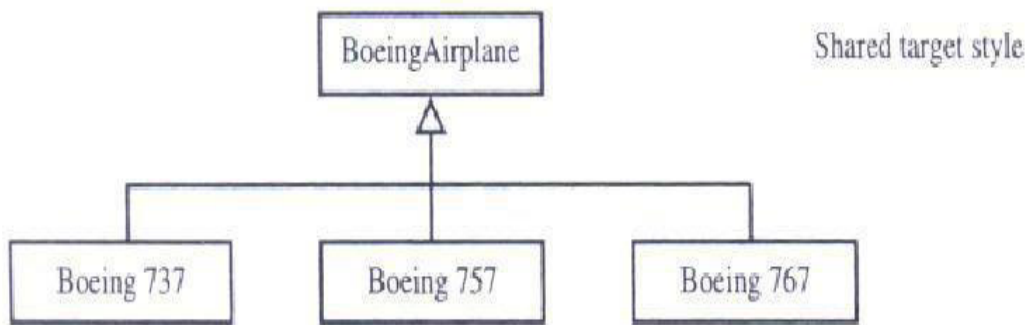


Fig. 2.23. Generalization Notation



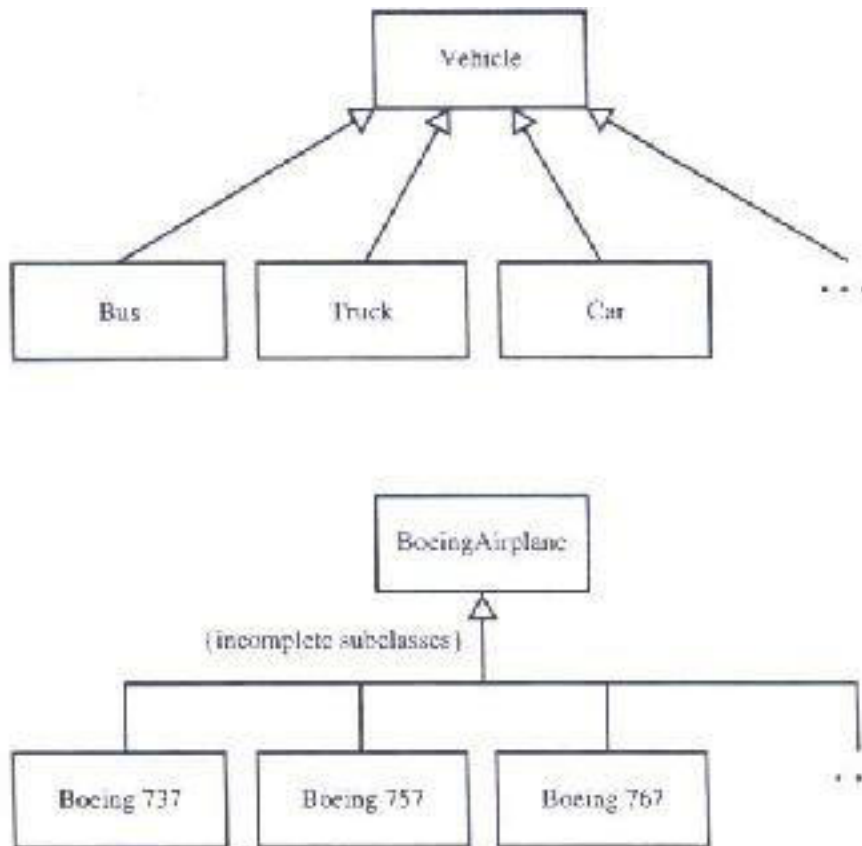


Fig. 2.24. Ellipses(...) indicate that additional classes exist and are not shown.

USE-CASE DIAGRAM

The use-case concept was introduced by Ivar Jacobson in the object-oriented software engineering (OOSE) method. The functionality of a system is described in a number of different use cases, each of which represents a specific flow of events in the system.

A use case corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors) and engages internal objects to interact with one another and with the system's surroundings.

The description of a **use case defines what happens in the system when the use case is performed**. In essence, the use-case model defines the outside (**actors**) and inside(**use case**) of the system's behavior. Use cases represent specific flows of events in the system. The **use cases** are initiated by actors and describe the flow of events that these actors set off. **An actor** is anything that interacts with a use case: It could be a human user, external hardware, or another system. An actor represents a category of user rather

than a physical user. Several physical users can play the same role. For example, in terms of a member actor, many people can be members of a library, which can be represented by one actor called **Member**.

A **use-case diagram** is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.

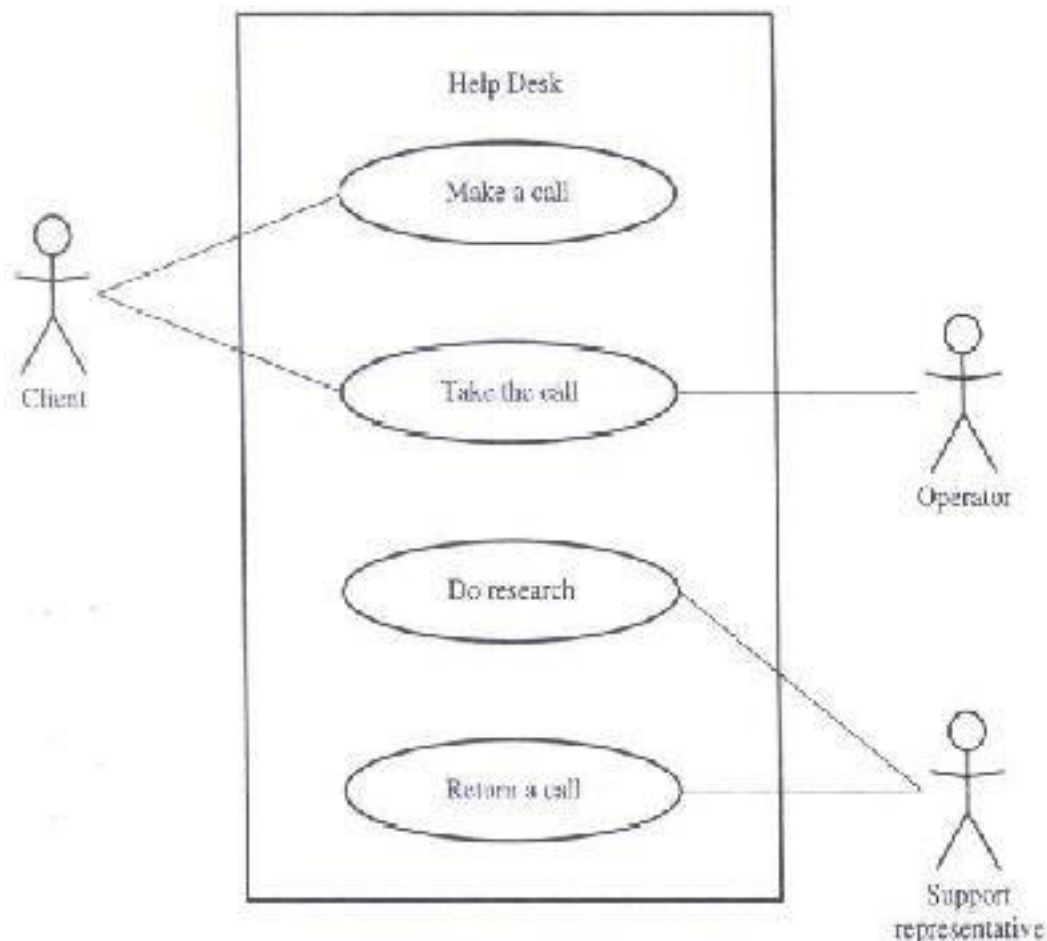


Fig. 2.25. Use-case diagram shows the relationship among actors and use cases within a system.

Fig 25. diagrams use cases for a Help Desk. A use-case diagram shows the relationship among the actors and use cases within a system. A client makes a call that is taken by an operator, who determines the nature of the problem. Some calls can be answered immediately; other calls require research and a return call.

A use case is shown as an ellipse containing the name of the use case. The name of the use case can be placed below or inside the ellipse. Actors' names and use case names should follow the capitalization and punctuation guidelines of the model.

An actor is shown as a class rectangle with the label <<actor>>, or the label and a stick fig, or just the stick fig with the name of the actor below the fig (see Fig 26).



Fig. 2.26. The three representations of an actor are equivalent.

These relationships are shown in a use-case diagram:

1. **Communication.** The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid path. The actor is said to "communicate" with the use case.
2. **Uses.** A uses relationship between use cases is shown by a generalization arrow from the use case.
3. **Extends.** The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it is like a subclass.

UML DYNAMIC MODELING (BEHAVIOR DIAGRAMS)

Booch explains that describing a systematic event in a static medium such as on a sheet of paper is difficult, but the problem confronts almost every discipline.

The Dynamic semantics of a problem with the following diagrams:

Behavior diagrams (Dynamic)

- Interaction Diagrams:
 - Sequence diagrams
 - Collaboration diagrams
- State Chart diagrams
- Activity diagrams

Each class may have an associated activity diagram that indicates the behavior of the class's instance (its object). In conjunction with the use-case model, we may provide a script or an interaction diagram to show the time or event ordering of messages as they are evaluated.

UML INTERACTION DIAGRAMS

Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done.

Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. The diagram shows a number of example objects and the messages passed between those objects within the use case. There are two kinds of interaction models: **sequence diagrams and collaboration diagrams.**

UML Sequence Diagram: Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. A sequence diagram shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence.

A sequence diagram has two dimensions: the **vertical** dimension represents time, the **horizontal** dimension represents different objects. The vertical line is called the **object's lifeline**. The **lifeline** represents the object's existence during the interaction. This form was first popularized by Jacobson. **An object** is shown as a box at the top of a dashed vertical line (see Fig 27). A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles. a sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

An example of a sequence diagram.

Telephone Call

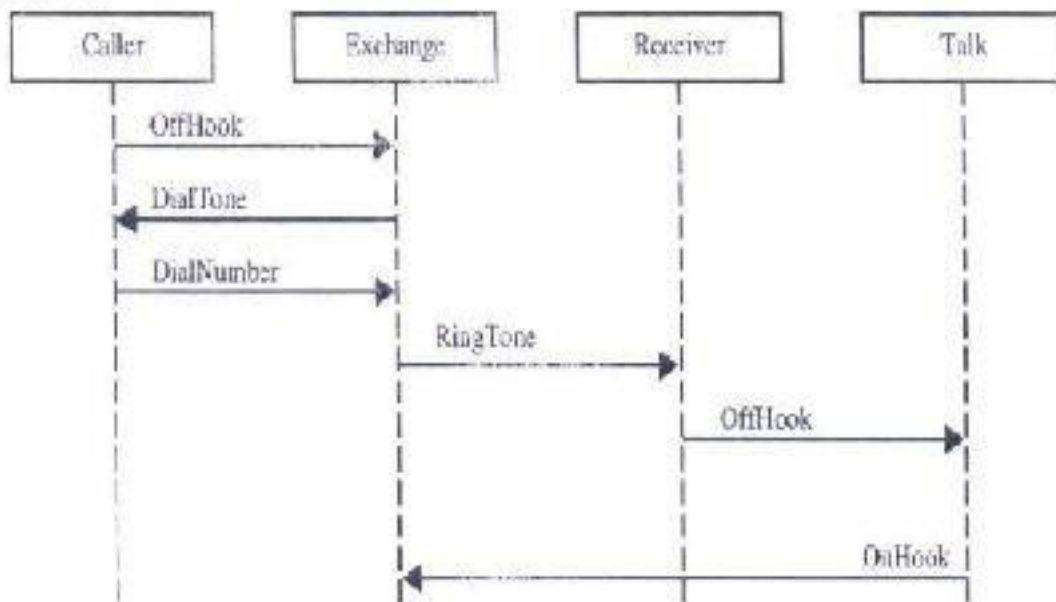


Fig. 2.27. An example of a Sequence Diagram

Each message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary. Often, call arrows are arranged to proceed in one direction across the page, but this is not always possible and the order conveys no information.

The sequence diagram is very simple and has immediate visual appeal-this is its great strength. A sequence diagram is an alternative way to understand the overall flow of the control of a program. Instead of looking at the code and trying to find out the overall sequence of behavior, you can use the sequence diagram to quickly understand that sequence.

UML Collaboration Diagram: Another type of interaction diagram is the collaboration diagram. A **collaboration diagram** represents a collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figs. As in a sequence diagram, arrows indicate the messagesent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages.

A collaboration diagram provides several numbering schemes. The simplest is illustrated in Fig 28.

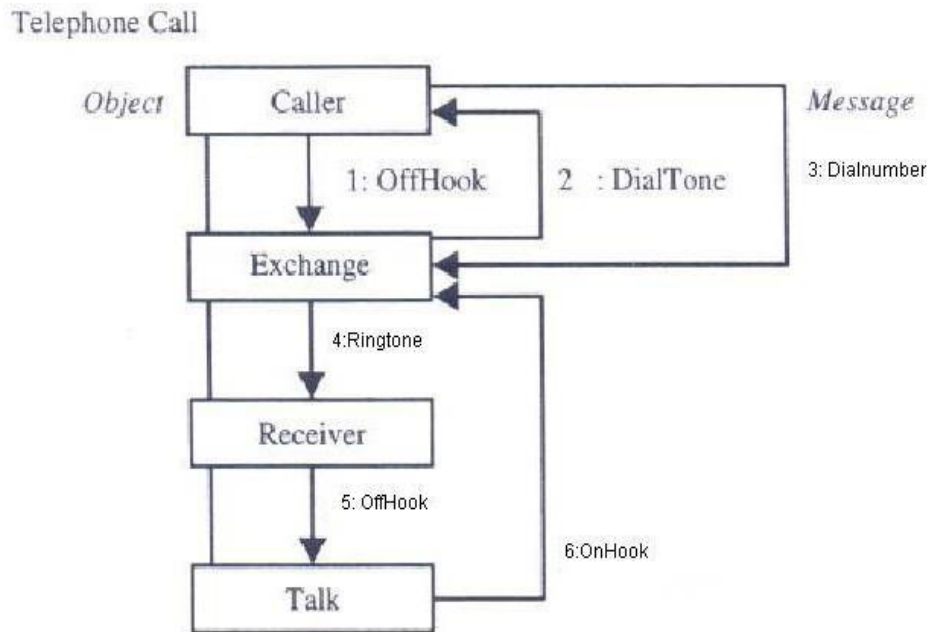


Fig. 2.28. A collaboration diagram with simple numbering.

Fowler and Scott argue that the **main advantage** of interaction diagrams (both collaboration and sequence) is **simplicity**. You easily can see the message by looking at the diagram. The **disadvantage** of interaction diagrams is that they are great only for representing a single sequential process; they begin to break down when you want to represent conditional looping behavior.

Conditional behavior can be represented in sequence or collaboration diagrams through two methods.

*. The preferred method is to use separate diagrams for each scenario. *. Another way is to use conditions on messages to indicate the behavior.

UML STATECHART DIAGRAM

A **state chart diagram** (also called a **state diagram**) shows the sequence of states that an object goes through during its life in response to outside stimuli and messages.

The state is the set of values that describes an object at a specific point in time and is represented by state symbols and the transitions are represented by arrows connecting

the state symbols. A state chart diagram may contain sub diagrams.

A state diagram represents the state of the method execution (that is, the state of the object executing the method), and the activities in the diagram represent the activities of the object that performs the method.

The purpose of the state diagram is to understand the algorithm involved in performing a method. To complete an object-oriented design, the activities within the diagram must be assigned to objects and the control flows assigned to links in the object diagram.

A state is represented as a rounded box, which may contain one or more compartments. The compartments are all optional. The name compartment and the internal transition compartment are two such compartments:

- The **name compartment** holds the optional name of the state. States without names are "anonymous" and all are distinct. Do not show the same named state twice in the same diagram, since it will be very confusing.
- The **internal transition compartment** holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing states:

The syntax used is this: event-name argument-list / action-expression; for example, help / display help.

Two special events are entry and exit, which are reserved words and cannot be used for event names. These terms are used in the following ways: entry I action expression (the action is to be performed on entry to the state) and exit I action expressed (the action is to be performed on exit from the state).

The statechart supports nested state machines; to activate a substate machine use the keyword do: do I machine-name (argument-list). If this state is entered, after the entry action is completed, the nested (sub)state machine will be executed with its initial state. When the nested state machine reaches its final state, it will exit the action of the current state, and the current state will be considered completed. An initial state is shown as a small dot, and the transition from the initial state may be labeled with the event that creates the objects; otherwise, it is unlabeled. If unlabeled, it represents any transition to the enclosing state.

A final state is shown as a circle surrounding a small dot, a bull's-eye. This represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition (see Fig 29).

The transition can be simple or complex. A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain actions when a specific event occurs; if the specified conditions are satisfied, the transition is said to "fire." Events are processed one at a time. An event that triggers no transition is simply ignored.

A complex transition may have multiple source and target states. It represents a synchronization or a splitting of control into concurrent threads. A complex transition is enabled when all the source states are changed, after a complex transition "fires" all its destination states. A complex transition is shown as a short heavy bar. The bar may have one or more solid arrows from states to the bar (these are source states); the bar also may have one or more solid arrows from the bar to states (these are the destination states). A transition string

may be shown near the bar. Individual arrows do not have their own transition strings (see Fig 5-30).

A simple state, Idle, and a nested state. The dialing state contains substates, which consist of start and dial states.

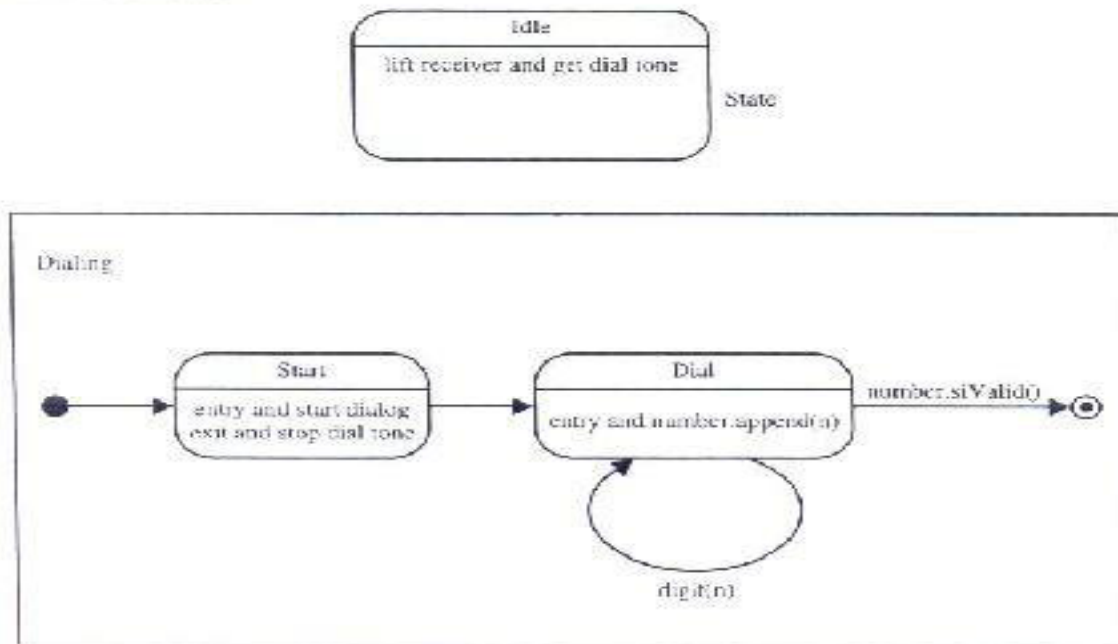


Fig. 2.29. A simple ideal and a nested state.

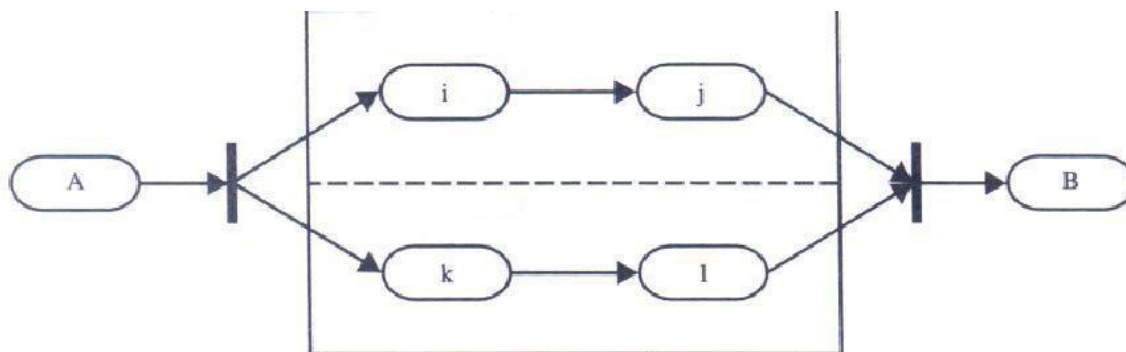


Fig. 2.30. A complex Transition
UML ACTIVITY DIAGRAM

An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. Unlike state diagrams that focus on the events occurring to a single object as it responds to messages, an activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. An activity diagram can also be used to represent a class's method implementation.

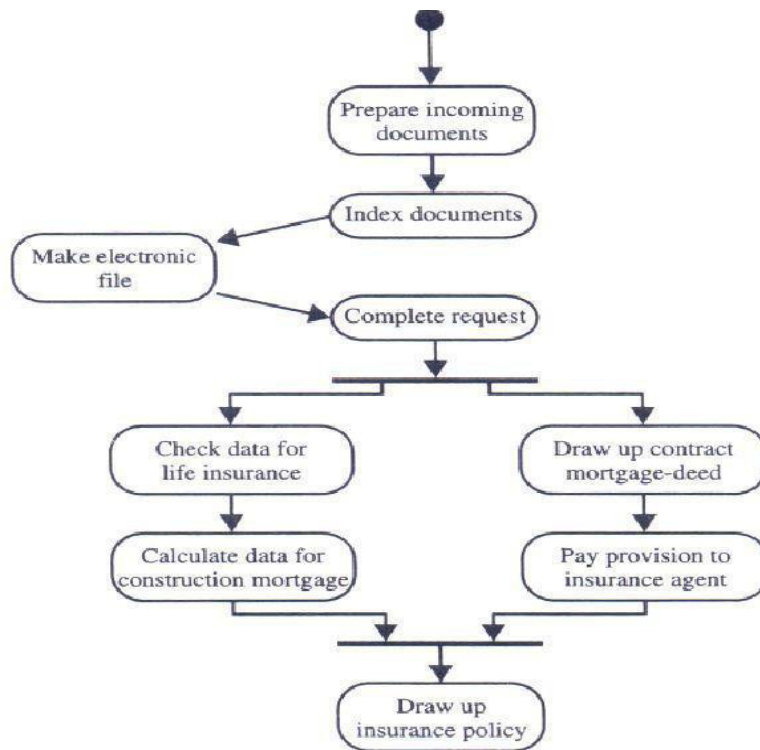


Fig. 2.31. An activity diagram for processing mortgage requests
(Loan: ProcessingMortgage Request).

An activity model is similar to a state chart diagram, where a token (shown by a black dot) represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation.

Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in different phases.

An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but the conditions that depend on the result of the activity or other values may be included (Fig. 31).

Several transitions with different conditions imply a branching off of control. If conditions are not disjoint, then the branch is nondeterministic. The concurrent control is represented by multiple arrows leaving a synchronization bar, which is represented by a short thick bar with incoming and outgoing arrows. Joining concurrent control is expressed by multiple arrows entering the synchronization bar.

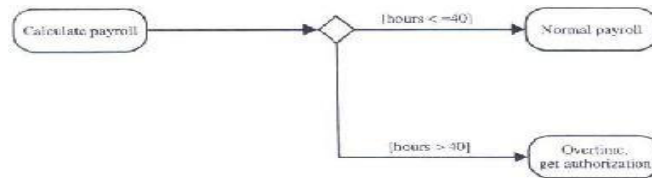


Fig. 2.32. A decision.

An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "waitstate," a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result of an activity by another object (such as a user input or some other signal). **The two states are wait state and activity state.** More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Activity and state diagrams express a decision when conditions (the UML calls them **guard conditions**) are used to indicate different possible transitions that depend on Boolean conditions of container object. The fig 2.32 provided for a decision is the traditional diamond shape, with one or more incoming arrows and two or more outgoing arrows, each labeled by a distinct guard condition. All possible outcomes should appear on one of the outgoing transitions (see Fig 2.32).

Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each **swimlane** represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path (see Fig 2.33).

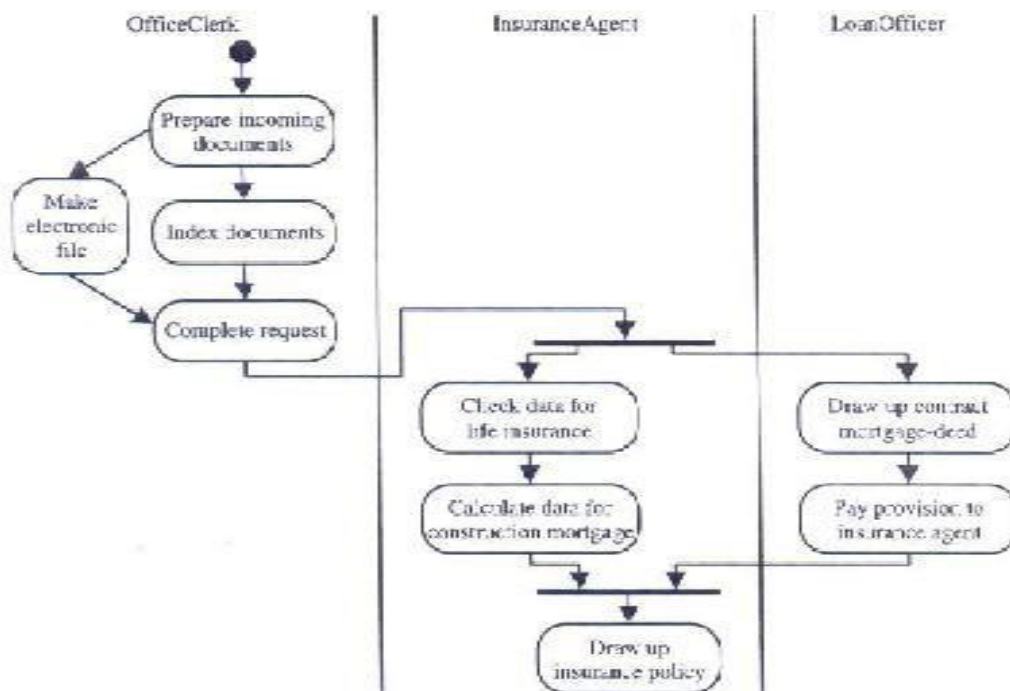


Fig. 2.33. Swimlanes in an activity diagram.

UML IMPLEMENTATION DIAGRAMS

Implementation diagrams show the implementation phase of systems development, such as the source code structure and the run-time implementation structure. There are 2 types of implementation diagrams:

- *. Component diagrams – Its Show the structure of the code itself.
- *. Deployment Diagrams – Its show the structure of the runtime system. These are relatively simple, high-level diagrams compared with other UML diagrams.

Component Diagram:

Component diagrams model the physical components (such as source code, executable program, user interface) in a design. These high-level physical components may or may not be equivalent to the many smaller components you use in the creation of your application. For example, a user interface may contain many other off-the-shelf components purchased to put together a graphical user interface.

Another way of looking at components is the concept of **packages**. A package is used to show how you can group together classes, which in essence are smaller scale components. A package usually will be used to group logical components of the application, such as classes, and not necessarily physical components. However, the package could be a first approximation of what eventually will turn into physical grouping. In that case, the package will become a component.

A component diagram is a graph of the design's components connected by dependency relationships. A component is represented by the boxed fig shown in Fig 34. Dependency is shown as a dashed arrow.

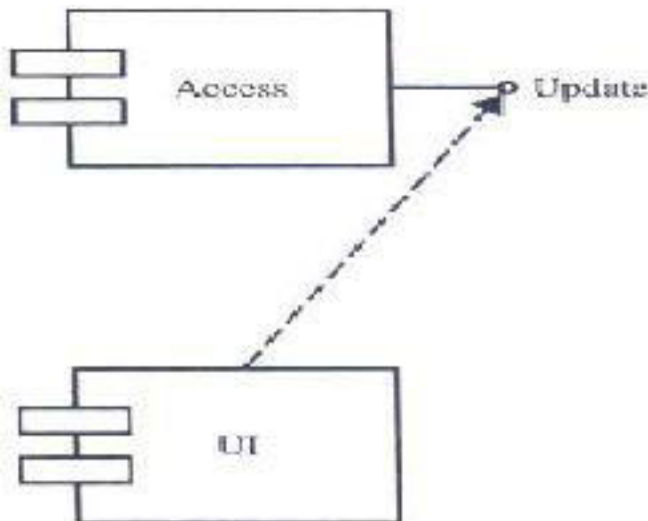


Fig. 2.34. A Component Diagram

Deployment Diagram

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live in them. Software component instances represent run-time manifestations of code units. In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributed on various hardware platforms. In many cases, component and deployment diagrams can be combined.

A deployment diagram is a graph of nodes connected by communication association. Nodes may contain component instances, which mean that the component lives or runs at that node. Components may contain objects; this indicates that the object is part of the component. Components are connected to other components by dashed arrow dependencies, usually through interfaces, which indicate one component uses the services of another. Each node or processing element in the system is represented by a three- dimensional box. Connections between the nodes (or platforms) themselves are shown by solid lines (see Fig 35).

The basic UML notation for a deployment diagram.

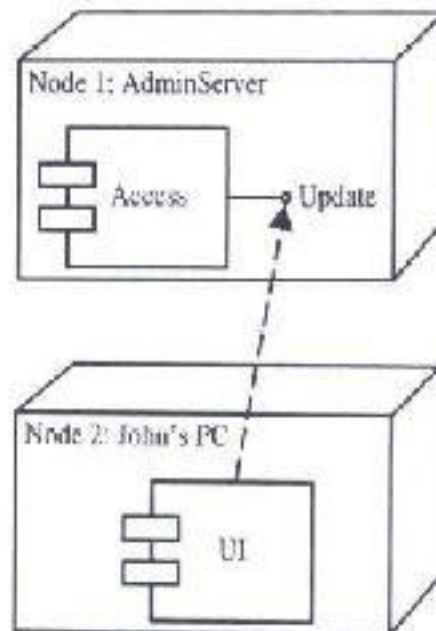


Fig. 2.35. The Basic UML notation for a deployment diagram.

MODEL MANAGEMENT: PACKAGES AND MODEL ORGANIZATION

A **package** is a grouping of model elements. Packages themselves may contain other packages. A package may contain both subordinate packages and ordinary model elements. The entire system can be thought of as a single high-level package with everything else in it. All UML model elements and diagrams can be organized into packages.

A package is represented as a folder, shown as a large rectangle with a tab attached to its upper left corner. If contents of the package are not shown, then the name of the package is placed within the large rectangle. If contents of the package are shown, then the name of the package may be placed on the tab (see Fig 36). The contents of the package are shown within the large rectangle. Fig shows an example of several packages. This fig shows three packages (Clients, Bank, and Customer) and three classes, (Account class, Savings class, and Checking class) inside the Business Model package.

A package and its contents.

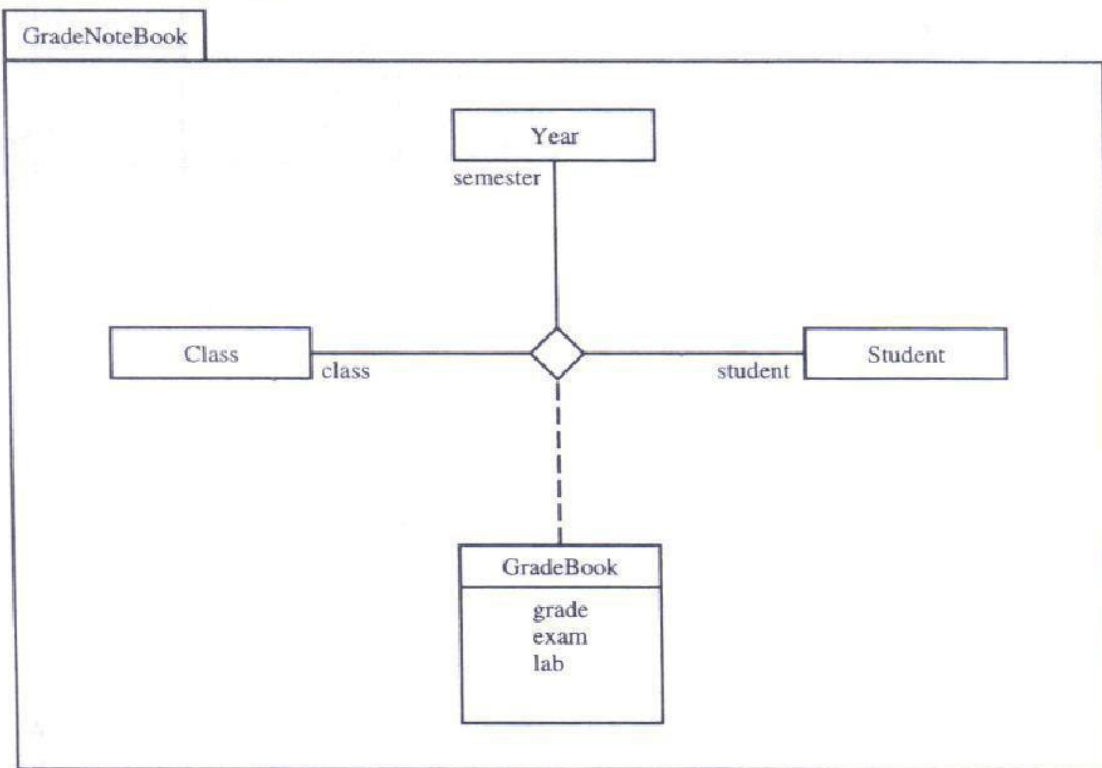


Fig. 2.36. A package and its contents

A real model would have many more classes in each package. The contents might be shown if they are small, or they might be suppressed from higher levels. The entire system is a package. Fig 37 also shows the hierarchical structure, with one package dependent on other packages. For example, the Customer depends on the package Business Model, meaning that one or more elements within Customer depend on one or more elements within the other packages. The package Business Model is shown partially expanded. In this case, we see that the package Business Model owns the classes Bank, Checking, and Savings as well as the packages Clients and Bank. Ownership may be shown by a graphic nesting of the figs or by the expansion of a package in a separate drawing. Packages can be used to designate not only logical and physical groupings but also use-case groups. A use-case group, as the name suggests, is a package of use cases.

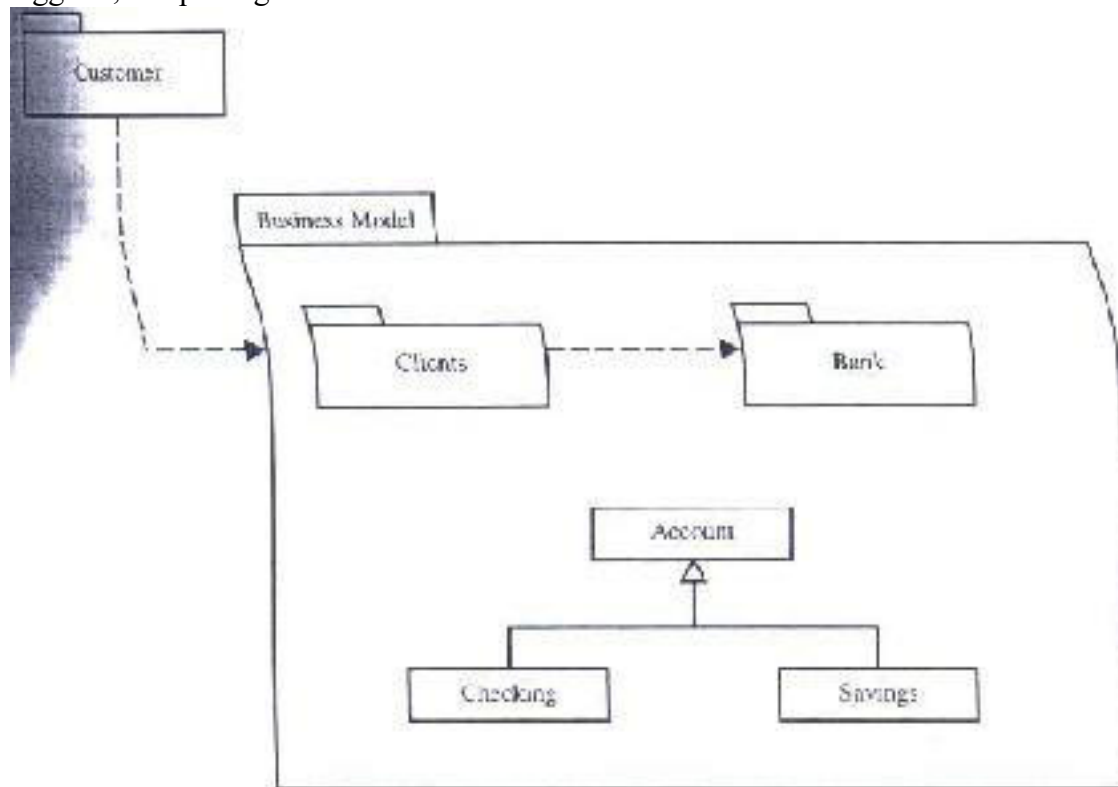


Fig. 2.37. A package and its dependencies

Model dependency represents a situation in which a change to the target element may require a change to the source element in the dependency, thus indicating the relationship between two or more model elements. It relates the model elements themselves and does not require a set of instances for its meaning. A dependency is shown as a dashed arrow from

one model element to another on which the first element is dependent (see Fig 38).

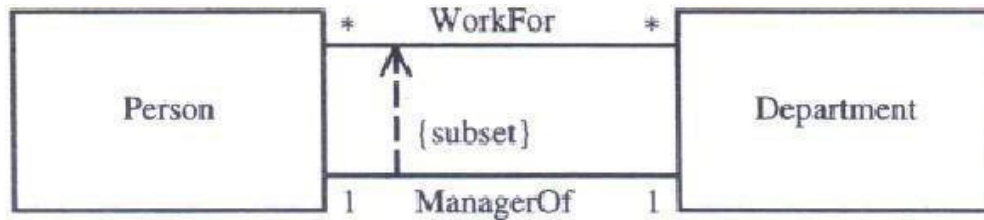


Fig. 2.38. An example of constraints. A person is a manager of people who work for the accounting department.

UML EXTENSIBILITY

1. Model Constraints and comments

Constraints are assumptions or relationship among model elements specifying conditions and propositions that must be maintained as true; otherwise, the system described by the model would be invalid. Some constraints, such as association OR constraints are predefined in the UML; others may be defined by users.

Constraints are shown as text in braces (ref. Fig 38). The UML also provides language for writing constraints in the OCL. The constraints may be written in a natural language.

A constraint may be a “Comment”, in which case it is written in text.

For an element whose notation is a text string such as an attribute, the constraint string may follow the element text string. For a list of elements whose notation is a list of text strings, such as the attributes within class, the constraint string may appear as an element in the list. The constraint applies to all succeeding elements of the list until reaching another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraints but may modify individual constraints string maybe placed near the symbol name.

The above example fig 38, shows two classes and two associations. The constraint is shown as a dashed arrow from one element to the other, labeled by the constraints string in brace. The direction of the arrow is relevant information within the constraint.

2. Note

A Note is a graphic symbol containing textual information; it also could contain embedded images. It is attached to the diagram rather than to a model element. A note is shown as a rectangle with “Bent Corner” in the upper right corner. It can contain any length text. (ref. Fig 39).

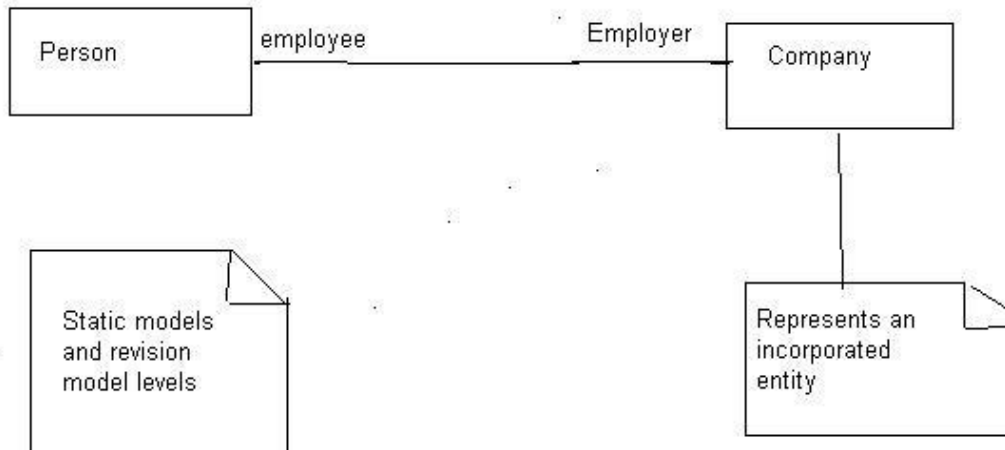


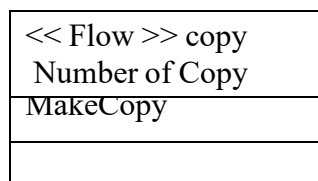
Fig. 2.39. Note

3. Stereotype

Stereotype represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints. A stereotype is a new class of modeling element introduced during modeling, It represents a subclass of an existing modeling element with the same form (attributed and relationships) but a different intent. UML stereotype extend and tailor the UML for a specific domain or process.

The general presentation of a stereotype is to use a figure for the base element but place a keyword string above the name of the element (if used, the keyword string is the name of a stereotype within matched guillemets,

“<<”, ”>>”, such as <<flow>>.. Note that a guillemet looks like a angle-bracket, but it is a single character in most fonts.



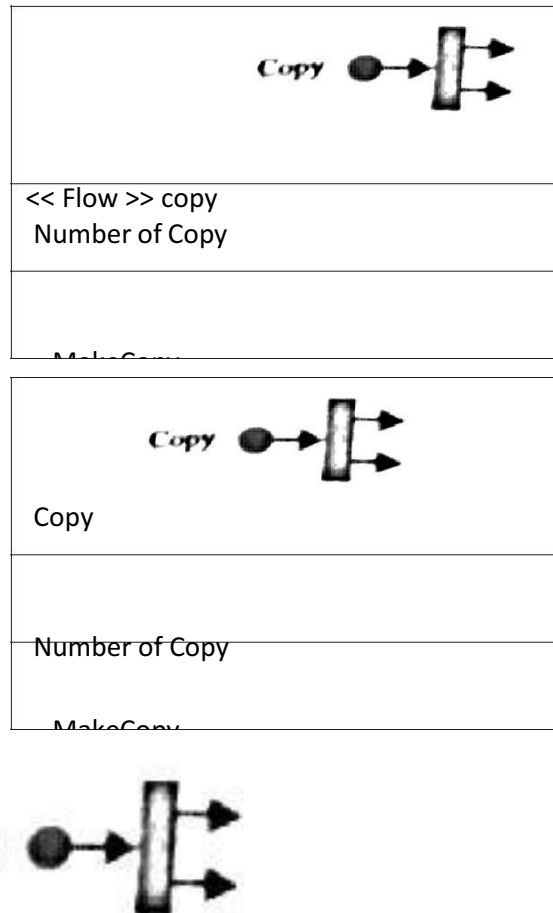


Fig. 2.40. Various forms of Stereotype notation

The stereotype allows extension of UML notation as well as a graphic figure, texture, and color. The figure can be used in one of two ways: (1) instead of or in addition to the stereotype keyword string as part of the symbol for the base model element or (2) as the entire base model element. Other information contained by the base model element symbol is suppressed.

3. UML Meta-model

The UML defined notations as well as a meta model. UML graphic notations can be used not only to describe the system's components but also to describe a model itself. This is known as a **meta-model**.

In other words, a **meta-model** is a model of modeling elements. The purpose of the UML meta-model is to provide a single, common, and definitive statement of the syntax and semantics of the elements of the UML.

The meta model provides us a means to comment different UML diagrams. The connection between the different diagrams is very important, and the UML attempts to make these coupling more explicit through defining the underlying model while imposing no methodology.

The presence of this meta model has made it possible for its developers to agree on semantics and how those semantics would be best rendered. This is an important step forward, since it can assure consistency among diagrams. The meta-model also can serve as a means to exchange data between different case tools. The fig 41 is an example for the UML meta-model that describes relationship with association and generalization; association is depicted as a composition of association roles.

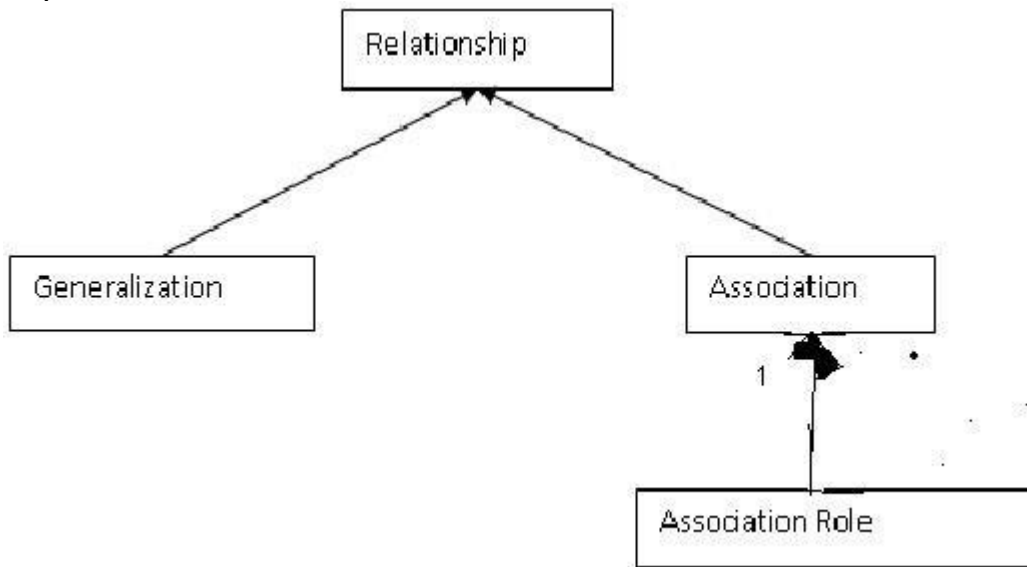


Fig. 2.41. The UML meta-model describing the relationship between association and generalization.

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, "Object oriented systems development using the unified modelling language", 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modelling Language User Guide", 3rd Edition Addison Wesley, 2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modelling Language User Guide", 3rd Edition Addison Wesley.
5. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley.
6. Bernd Oestereich, "Developing Software with UML, Object - Oriented Analysis and Design in Practice", Addison-Wesley

Questions

Part-A			
Q.No	Questions	Competence	BT Level
1.	Compare the difference between a method and a process?	Understand	BTL 2
2.	Define an object model? What are the other OMT models?	Knowledge	BTL 1
3.	Sketch the reason for having abstract use cases?	Apply	BTL 3
4.	Describe the difference between patterns and frameworks.	Knowledge	BTL 1
5.	Sketch multiplicity	Apply	BTL 3
6.	Describe the relationships in a use case diagram.	Knowledge	BTL 1
7.	Compare extends and uses relationship?	Understand	BTL 2
8.	Classify the responsibilities of access and view layer?	Understand	BTL 2
9.	Summarize object diagram, binary association, association role, navigability?	Understand	BTL 2
10.	Defend n-ary association, aggregation, composition, generalization, use case diagram?	Evaluate	BTL 5
Part-B			
Q.No	Questions	Competence	BT Level
1.	Describe Rumbaugh's Object Modelling Technique?	Knowledge	BTL 1
2.	Summarize detailed notes about the Booch Methodology?	Understand	BTL 2
3.	Summarize a detailed account of Jacobson methodology?	Understand	BTL 2
4.	Illustrate in detail about the Unified approach?	Analyze	BTL 4