COURSE FILE

B. Tech III-I

15A05503

OBJECT ORIENTED ANALYSIS AND DESIGN

AMARENDRANATH KOTA

Assistant Professor, CSE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SVR ENGINEERING COLLEGE

NANDYAL

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech III-I Sem. (CSE) **L T P C**
3 1 0 3

15A05503 OBJECT ORIENTED ANALYSIS & DESIGN

Course Objectives

- To understand how to solve complex problems
- Analyze and design solutions to problems using object oriented approach
- Study the notations of Unified Modeling Language

Course Outcomes:

- Ability to find solutions to the complex problems using object oriented
- approach
- Represent classes, responsibilities and states using UML notation
- Identify classes and responsibilities of the problem domain.

Unit-I

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing Complex Systems, Evolution of Object Model, Foundation of Object Model, Elements of Object Model, Applying the Object Model.

Unit-II

Classes and Objects: Nature of object, Relationships among objects, Nature of a Class, Relationship among Classes, Interplay of Classes and Objects, Identifying Classes and Objects, Importance of Proper Classification, Identifying Classes and Objects, Key abstractions and Mechanisms.

Unit-III

Introduction to UML: Why model, Conceptual model of UML, Architecture, Classes, Relationships, Common Mechanisms, Class diagrams, Object diagrams.

Unit-IV

Structural Modeling: Package Diagram, Composite Structure Diagram, Component Diagram, Deployment Diagram, Profile Diagram.

Unit-V

Behavioral Modeling: Use Case Diagram, Activity Diagrams, State Machine Diagrams, Sequence Diagram, Communication Diagram, Timing Diagram, Interaction Overview Diagram.

Text Books:

1. ―Object- Oriented Analysis And Design with Applications‖ , Grady BOOCH, Robert A. Maksimchuk, Michael W. ENGLE, Bobbi J. Young, Jim Conallen, Kellia Houston, PEARSON, 3rd edition, 2013.
2. ―The Unified Modeling Language User Guide‖ , Grady Booch, James Rumbaugh, Ivar Jacobson, PEARSON 12th Impression, 2012.
3. http://www.omg.org/

References:

1. ―Object-oriented analysis and design using UML‖ , Mahesh P. Matha, PHI
2. ―Head first object-oriented analysis and design‖ , Brett D. McLaughlin, Gary Pollice, Dave West, O'Reilly
3. ―Object-oriented analysis and design with the Unified process‖ , John W. Satzinger, Robert B. Jackson, Stephen D. Burd, Cengage Learning
4. ―The Unified modeling language Reference manual‖ , James Rumbaugh, Ivar Jacobson, Grady Booch, Addison-Wesley4. "The Unified modeling language Reference manual", James Rumbaugh, Ivar Jacobson,
Grady Booch, Addison-Wesley

UNIT – I

UNIT I

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing Complex Systems, Evolution of Object Model, Foundation of Object Model, Elements of Object Model, Applying the Object Model.

TOPICS:

INTRODUCTION:

1) THE STRUCTURE OF COMPLEX SYSTEMS
2) THE INHERENT COMPLEXITY OF SOFTWARE
3) ATTRIBUTES OF COMPLEX SYSTEM
4) ORGANIZED AND DISORGANIZED COMPLEXITY
5) BRINGING ORDER TO CHAOS
6) DESIGNING COMPLEX SYSTEMS
7) EVOLUTION OF OBJECT MODEL
8) FOUNDATION OF OBJECT MODEL
9) ELEMENTS OF OBJECT MODEL
10) APPLYING THE OBJECT MODEL

[1]. The Structure of Complex Systems:

Examples of Complex Systems

The Structure of a Personal Computer :

1) A personal computer is a device of moderate complexity. Most of them are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a floppy disk or a hard disk drive.
2) CPU typically encompasses primary memory, an arithmetic/logic unit (ALU), and a bus to which peripheral devices are attached.
3) An ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

1)The Structure of Plants and Animals:

- Plants are complex multi cellular organisms, and from the cooperative activity of various plant organ systems arise such complex behaviors as photosynthesis and transpiration.
- Plants consist of **three major structures** (roots, stems, and leaves), and each of these has its own structure.

For example:

- At the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil.
- Roots interact with stems, which transport these raw materials up to the leaves.
- The leaves in turn use the water and minerals provided by the stems to produce food through Photosynthesis.

2)The Structure of Animals:

Animals exhibit a hierarchical structure similar to that of plants: collections of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system), and so on.

3)The Structure of Matter:

- Nuclear physicists are concerned with a structural hierarchy, of matter.
- Atoms are made up of electrons, protons, and neutrons; electrons appear to be elementary particles, but protons, neutrons, and other particles are formed from more basic components called *quark*s.

4)**The Structure of Social Institutions:**

- The structure of social institutions, Groups of people join together to accomplish tasks that cannot be done by individuals.
- Some organizations are transitory, and some endure beyond many lifetimes.
- If the organization endures, the boundaries among these parts may change, and over time, a new, more stable hierarchy may emerge.
- The relationships among the various parts of a large organization are just like those found among the components of a computer, or a plant, or even a galaxy.

[2] The inherent complexity of software:

The Properties of Simple and Complex Software Systems:

Software may also involve elements of great complexity; however, the complexity we find here is of a fundamentally different kind.

- **Some software systems are simple:**

These are the largely forgettable applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation.

Such systems tend to have a very limited purpose and a very short life span.

- **Some software systems are complex:**

we find applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information.

Although such applications are generally products of research and development they are no less complex, for they are the means and artifacts of incremental and exploratory development.

Why Software Is Inherently Complex:

"The complexity of software is an essential property, not an accidental one" . We observe that this inherent complexity derives from **four elements**:

- The complexity of the problem domain.
- The difficulty of managing the developmental process.
- The flexibility possible through software.
- The problems of characterizing the behavior of discrete systems.

The Complexity of the Problem Domain:

In which we find a myriad of competing, perhaps even contradictory, requirements. Consider the requirements for the electronic system of a **multi-engine aircraft, a cellular phone switching system, or an autonomous robot.**

The raw functionality of such systems is difficult enough to comprehend, but now add all of the (often implicit) nonfunctional requirements such as usability, performance, cost, survivability, and reliability.

The Difficulty of Managing the Development Process:

The fundamental task of the software development team is Lo engineer the illusion of simplicity - to shield users from this vast and often arbitrary external complexity

More developers means more complex communication and hence more difficult coordination, particularly if the team isgeographically dispersed, as is often the case in very large projects. With a team of developers, the key management challenge is always to maintain a unity and integrity of design.

The Flexibility Possible Through Software:

Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction.

This flexibility turns out to be an incredibly seductive property, however, because it also forces the developer to craft virtually all the primitive building blocks upon which these higher-level abstractions stand.

While the construction industry has uniform building codes and standards for the quality of raw materials, few such standards exist in the software industry.

The Problems of Characterizing the Behavior of Discrete Systems:

Within a large application, there may be hundreds or even thousands of variables as well asmore than one thread of control.

The entire collection of these variables, their current values, and the current address and calling stack of each process within the system constitute the present state of the application.

Since we have neither the mathematical tools nor the intellectual capacity to model the complete behavior of large discrete systems, we must be content with acceptable levels of confidence regarding their correctness.

The Consequences of Unrestrained Complexity:

"The more complex the system, the more open it is to total breakdown" .Rarely would abuilder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure

[3] The Five Attributes of a Complex System:

There are five attributes common to all complex systems.

1. "Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached" .

2. The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.

3. "Intracomponent linkages are generally stronger than intercommoning linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components".

4. "Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements ".

5. "A complex system that works is invariably found to have evolved from a simple system that worked.... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system ".

[4]Organized and Disorganized Complexity

The Canonical Form of a Complex System :

The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex systems.

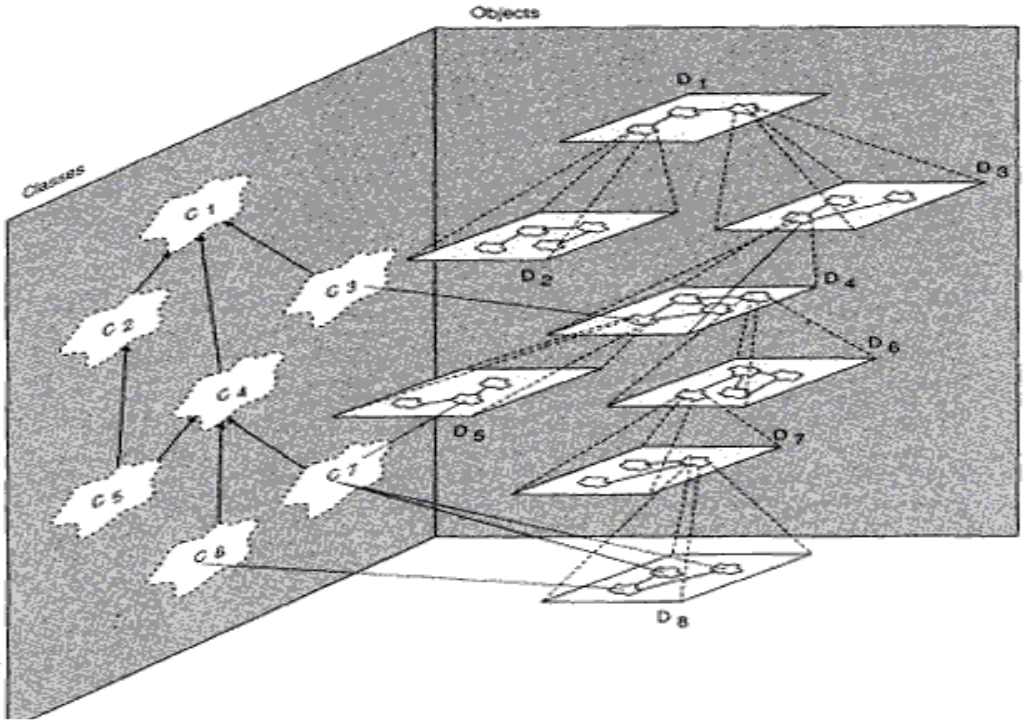


Fig:The Canonical Form of a Complex System.

This decomposition represents a structural, or "part of" hierarchy. Alternately, we can cut across the system in an entirely orthogonal way.

For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or "part of" hierarchy.

The concept of the class and object structure together with the five attributes of a complex system, we find that virtually all complex systems take en the same (canonical) form.

The two orthogonal hierarchies of the system: its class structure and its object structure. Each hierarchy is layered, with the more abstract classes and objects built upon more primitive ones.

As the figure suggests, there are usually many more objects than classes of objects within a complex system.

The most successful complex software systems are those whose designs explicitly encompass a well-engineered class and object structure and whose structure embodies the five attributes of complex systems.

Thus, by showing the "part of" as well as the "is a" hierarchy, we explicitly expose the redundancy of the system under consideration.

If we did not reveal a system's class structure, we would have to duplicate o knowledge about the properties of each individual part.

[5] Bringing Order to Chaos:

The Role of Decomposition:

"The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule)" . When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.

Algorithmic Decomposition:

In this decomposition is a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process.

Figure : is an example of one of the products of structured design, a structure chart that shows the relationships among various functional elements of then solution.

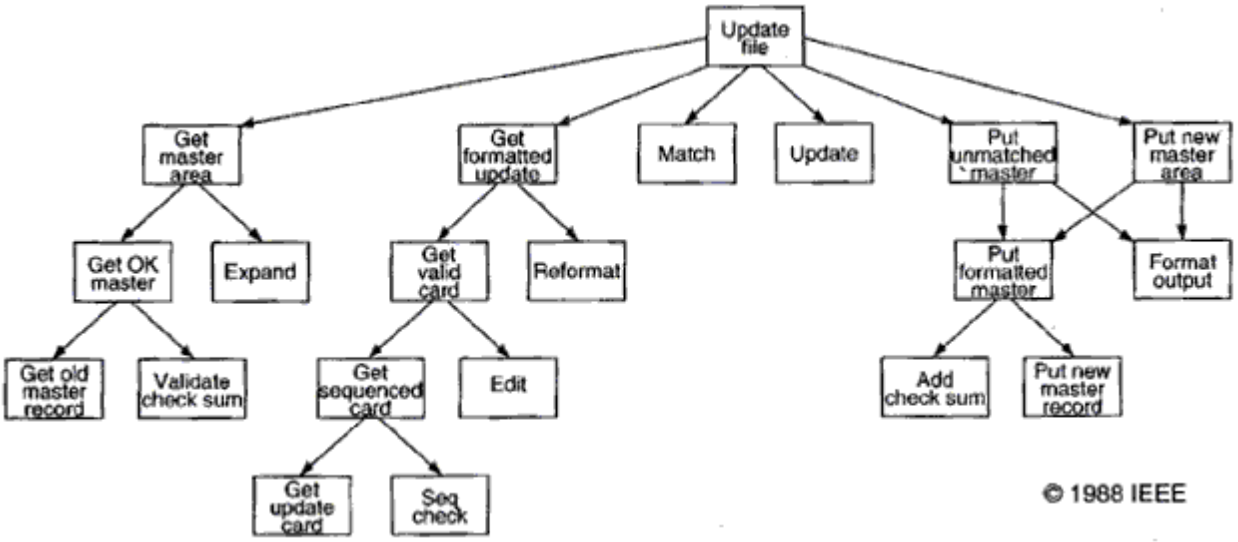


Fig: algorithemic Decomposition

Object-Oriented Decomposition: It was automatically generated from a data flow diagram by an expert system tool that embodies the rules of structured design

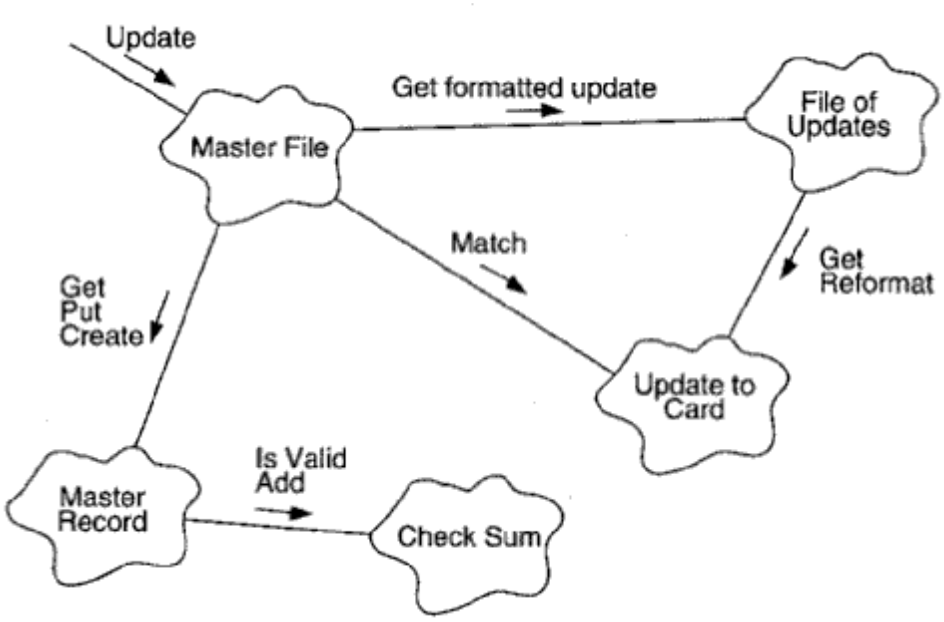


Fig:Object-Oriented Decomposition

Decomposed the system according to the key abstractions in the problem domain.

Rather than decomposing the problem into steps such as ***Get formatted update*** and ***Add check sum*** , we have identified objects such as ***Master File*** and ***Check Sum***, which derive directly from the vocabulary of the problem domain.

Get formatted update thus does not exist as an independent algorithm; rather, it is an operation associated with the object *File of Updates*. Calling this operation creates another object, *Update to Card.*

Algorithmic versus Object-Oriented Decomposition:

- The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act.
- The fact remains that we cannot construct a complex system in both ways simultaneously, for they are completely orthogonal views.
- Object-oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression.
- object-oriented decomposition directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.
- An exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to

ignore its inessential details, dealing instead with the generalized, idealized model of the object.

The Role of Abstraction:

- To increase the semantic content of individual chunks of information is by explicitly recognizing the class and object hierarchies within a complex software system.
- The object structure is important because it illustrates how different objects collaborate with one another through patterns of interaction that we call *mechanisms*.
- The class structure is equally important, because it highlights common structure and behavior within a system.
- Identifying the hierarchies within a complex software system is often not easy, because it requires the discovery of patterns among many objects, each of which may embody some tremendously complicated behavior.

[6] On Designing Complex Systems:

Engineering as a Science and an Art:

- The practice of every engineering discipline - be it civil, mechanical, chemical, electrical, or software engineering - involves elements of both science and art.

- "The conception of a design for a new structure can involve as much a leap of the imagination and as much a synthesis of experience and knowledge as any artist is required to bring to his canvas or paper.

- The role of the engineer as artist is particularly challenging when the task is to design an entirely new system. Frankly, this is the most common circumstance in software engineering.

- In other cases, such as the creation of frameworks, tools for research in artificial intelligence, or even information management systems, we may have a well defined, stable target environment, but our requirements may stress the software technology in one or more dimensions.

For example,

we may be asked to craft systems that are faster, have greater capacity, or have radically improved functionality. In all these situations, we try to use proven abstractions and mechanisms (the "stable intermediate forms," in Simon's words) as a foundation upon which to build new complex systems.

The Meaning of Design: In every engineering discipline, design encompasses the disciplined approach we use to invent a solution for some problem, thus providing

a path from requirements to implementation. In the context of software engineering, Mostow suggests that the purpose of design is to construct a system that:

• "Satisfies a given (perhaps informal) functional specification
• Conforms to limitations of the target medium
• Meets implicit or explicit requirements on performance and resource usage
• Satisfies implicit or explicit design criteria on the form of the artifact
• Satisfies restrictions on the design process itself, such as its length or cost, or the tools available for doing the design"

- As Stroustrup suggests, "the purpose of design is to create a clean and relatively simple internal structure, sometimes also called an architecture.... .
- A design is the end product of the design process" .Design involves balancing a set of competing requirements.
- The products of design are models that enable us to reason about our structures, make trade-offs when requirements conflict, and in general, provide a blueprint for implementation.

The Importance of Model Building :

- The building of models has a broad acceptance among all engineering disciplines, largely because model building appeals to the principles of decomposition, abstraction, and hierarchy.
- Each model within a design describes a specific aspect of the system under consideration.

The Elements of Software Design Methods:

- In fact, the design of complex software systems does not lend itself at all to cookbook approaches.
- Design methods do bring some much-needed discipline to the development process.
- The software engineering community has evolved dozens of, different design methods, which we can loosely classify into three categories (see sidebar).
- Despite their differences, all of these methods have elements in common.

Specifically, each method includes the following:

• Notation The language for expressing each model
• Process The activities leading to the orderly construction of the system's models
• Tools The artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed.

" A sound design method is based upon a solid theoretical foundation, yet offers degrees of freedom for artistic innovation"..

The Models of Object-Oriented Development:

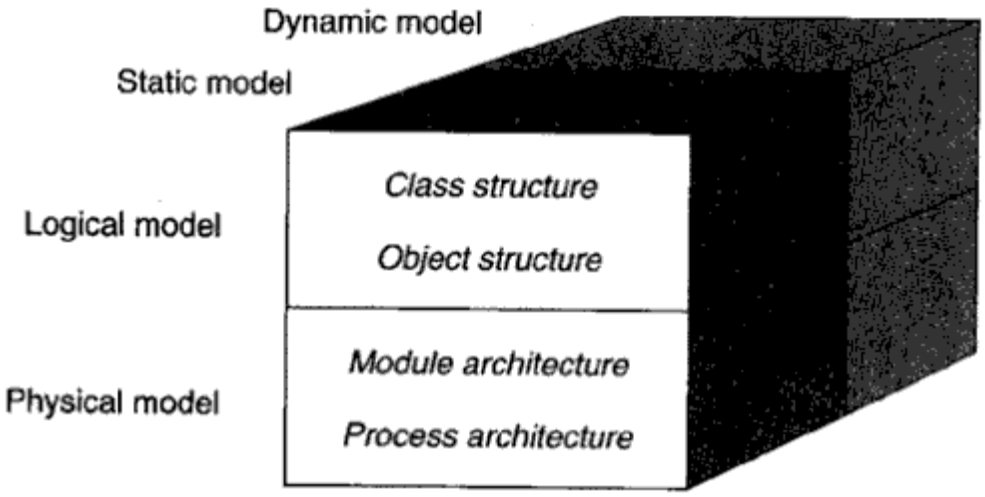


Fig: The Models of object oriented development.

we have found great value in building models that are focused upon the "things" we find, in the problem space, forming what we refer to as an "***object-oriented decomposition".***

- Object-oriented analysis and design is the method that leads us to an object-oriented decomposition.
- By applying object-oriented design, we create software that is resilient to change and written with economy of expression.
- Because model building is so important to the systems, object-oriented development offers a rich describe in Fig.
- The models of object-oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design.
- we have made a case for using object-oriented analysis and design to master the complexity associated with developing software systems.
- Additionally, we have suggested a number of fundamental benefits to be derived from applying this method.
- The object-oriented development is founded, namely, abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.

[7] The Evolution of the Object Model:

- The shift in focus from programming-in-the-small to programming-in-the-large.
- The evolution of high-order programming languages.
- New industrial-strength software systems are larger and more complex than their predecessors.
- Development of more expressive programming languages has complemented these advances.

Wegner has classified some of the more popular high-order programming languages in generations arranged according to the language features they first introduced:

- **First-Generation Languages (1954-1958)**

  FORTRANI Mathematical expressions
  ALGOL 58 Mathematical expressions
  Flowmatic Mathematical expressions
  IPL V Mathematical expressions.

- **Second-Generation Languages (1959~1961)**

  FORTRANII Subroutines, separate compilation
  ALGOL 60 Block structure, data types
  COBOL Data description, file handling
  Lisp List processing, pointers, garbage collection

- **Third-Generation Languages (1962-1970)**

  PL/1 FORTRAN + ALGOL + COBOL
  ALGOL 68 Rigorous successor to ALGOL 60
  Pascal Simple successor to ALGOL 60
  Simula Classes, data abstraction

- **The Generation Gap (1970-1980)**
  Many different languages were invented, but few endured .

The Topology of First- and Early Second-Generation Programming Languages:

- By topology, we mean the basic physical building blocks of the language and how those parts can be connected.

- The arrows in this figure indicate dependencies of the subprograms on various data.

- An error in one part of a program can have a devastating ripple effect across the rest of the system,

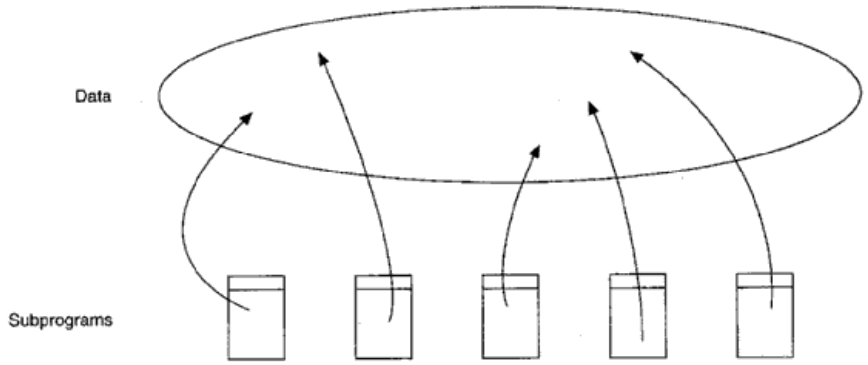


Fig: The Topology of First- and Early Second-Generation Programming Languages

The Topology of Late Second- and Early Third-Generation Programming Languages:

"The first software abstraction, now called the 'procedural' abstraction, grew directly out of this pragmatic view of software.

An abstraction mechanism had three important consequences.

- First, languages invented that supported a variety of parameter passing mechanisms.
- Second, the foundations of structured programming were laid,
- Third, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks.

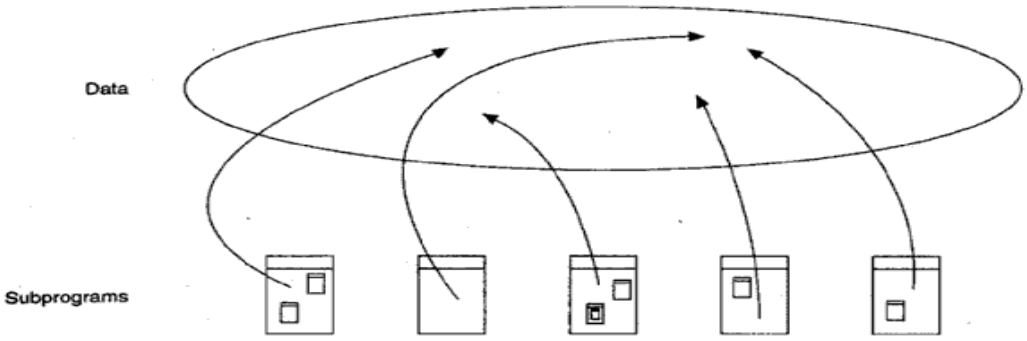
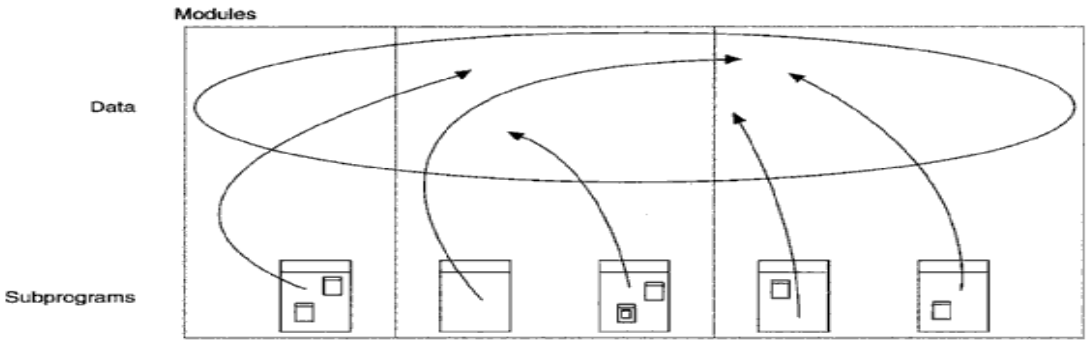


Fig: The Topology of Late Second- and Early Third-Generation Programming Languages

The Topology of Late Second- and Early Third-Generation Programming Languages:



The Topology of Late Third-Generation Programming Languages

Larger projects means- larger development teams and thus the need to develop different parts of the same program independently i.e compiled module.

The Topology of Object-Based and Object-Oriented Programming Languages:

This realization had two important consequences.

First, data-driven design methods emerged, for data abstraction.

Second, theories regarding the concept of a type appeared.which eventually found their realization in languages such as Pascal.

several languages such as -Smalltalk, Object Pascal, C++, CLOS, Ada, and Eiffel. For reasons that we will explain shortly, these languages are called *object-based orobject-oriented.*

The physical building block in these languages is the *module*, which represents a logical collection of classes and objects instead of subprograms,

"*lf procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns".*

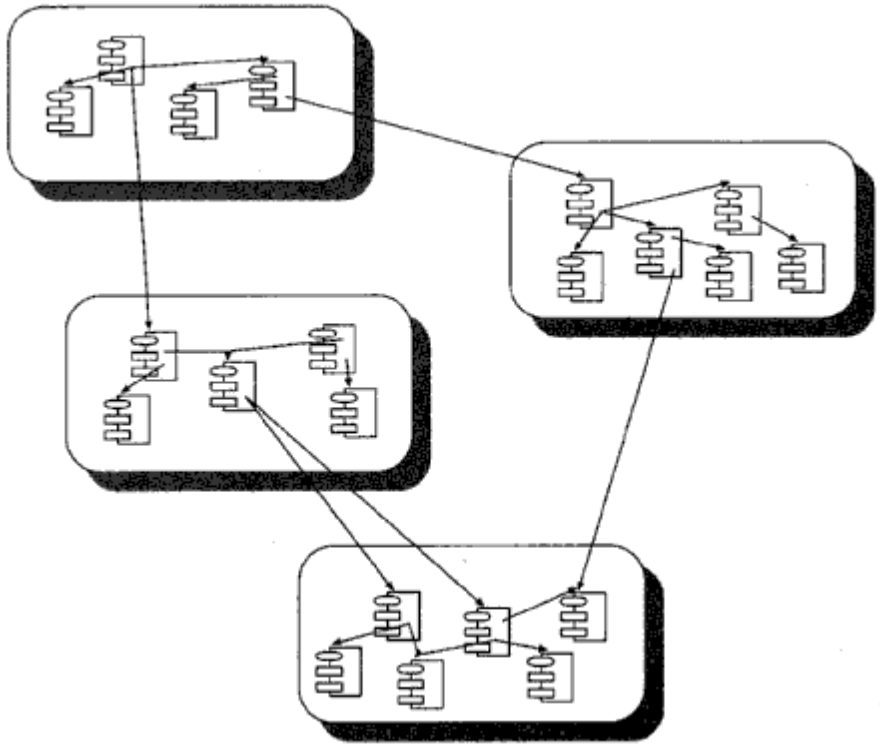


Figure : The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

That the fundamental logical building blocks of our systems are no longer algorithms, but instead are classes and objects. data and operations are united in such a way .

For very complex systems, we find that classes, objects, and modules provide an essential yet insufficient means of abstraction.

[8] Foundations of the Object Model:

Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks.

An object-oriented programming languages, using the class and object as basic building blocks.

Following events have contributed to the evaluation of object oriented concepts:

Advances in computer architecture including capability system and hardware support for operating system concepts.

Advances in programming language has demonstrated in simula,smalltalk, CLU,and ADA.

Advances in programming methodology ,including modularization and information hiding.

We would add to this list three more contributions to the foundation of the object model:

- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science.

Object-Oriented Analysis:

" Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain."

Object-oriented analysis (or OOA, as it is sometimes called) emphasizes the building of real-world models, using an object-oriented view of the world:

Object-Oriented Design:

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design."

There are two important parts to this definition: object-oriented design:
(1) leads to an objectorienteddecomposition and
(2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

Object-Oriented Programming:

"

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships"

A language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and
a hidden local state.

- Objects have an associated type [class].

• Types [classes] may inherit attributes from supertypes [superclasses]".

[9] Elements of the Object Model:

Kinds of Programming Paradigms:
They program in a paradigm enforced by the language they use, Frequently, they have not been exposed to alternate ways of thinking about a problem.

They further suggest that there are five main kinds of programming styles, here listed with the kinds of abstractions they employ:

• Procedure-oriented Algorithms
• Object-oriented Classes and objects
• Logic-oriented Goals, often expressed in a predicate calculus
• Rule-oriented If-then rules
• Constraint-oriented Invariant relationships.

Each of these styles of programming is based upon its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem. For all things object-oriented, the conceptual framework is the object model. There are four major elements of this model:
• Abstraction
• Encapsulation
• Modularity
• Hierarchy

By *major*, we mean that a model without any one of these elements is not object-oriented.
There are three minor elements of the object model:

• Typing
• Concurrency
• Persistence

By *minor*, we mean that each of these elements is a useful, but not essential, part of the object model.
An **abstraction** denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
• **Encapsulation** is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.
• **Modularity** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
• **Hierarchy** is a ranking or ordering of abstractions.

• **Typing** is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, be interchanged only in very restricted ways.
• **Concurrency** is the property that distinguishes an active object from one that is not active.
• **Persistence** is the property of an object through which its existence transcends time and/or space.

Abstraction
The Meaning of Abstraction:
An abstraction as "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.

A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary"

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer".

. From the most to the least useful, these kinds of abstractions include the following:
Entity abstraction: An object that represents auseful model of a problem domain or solution-domain entity

Action abstraction: An object that provides a generalized set of operations, all of which perform the same kind of function

Virtual machine abstraction: An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations.

Coincidental abstraction An object that: packages a set of operations that have no relation to each other.

Modularity:
The Meaning of Modularity

- The act of partitioning a program into individual components can reduce its complexity to some degree.
- A program is that it creates a number of well defined, documented boundaries within the program.
- These boundaries, or interfaces, are invaluable in the comprehension of the program".

"Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules".

[10] Applying the Object Model

Benefits of the Object Model:

- The object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming.
- Thus, the object model offers a number of significant benefits that other models simply do not provide.
- Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems.

There are five other practical benefits to be derived from the application of the object model.

1) The use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages.
2) The use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frame-works.
3) The use of the object model produces systems that are built upon stable intermediate forms, which are more resilient to change.
4) The object model's guidance in designing an intelligent separation of concerns also reduces development risk and increases our confidence in the correctness of our design.
5) Finally, the object model appeals to the workings of human cognition, for as Robson suggests,

"Many people who have no idea how a computer works find the idea of object-orientedsystems quite natural".

Applications of the Object Model:

The object model has proven applicable to a wide variety of problem domains.
Air traffic control :

- Mathematical analysis
- Banking and insurancesoftware
- Chemical process control
- Operating systems
- Computer aided design
- Computer aided education
- Robotics
- Telecommunications

The lists many of the domains for which systems exist that may properly be called object-oriented.

Object-oriented analysis and design may be the only method , that can be employed to attack the complexity inherent in very large systems.

Open Issues

To effectively apply the elements of the object model, we must next address several open issues:

• What exactly are classes and objects?
• How does one properly identify the classes and objects that are relevant to a particular application?
• What is a suitable notation for expressing the design of an object-oriented system?
• What process can lead us to a well-structured object-oriented system?
• What are the management implications of using object-oriented design?.

UNIT-II

Classes and Objects: Nature of object, Relationships among objects, Nature of a Class, Relationship among Classes, Interplay of Classes and Objects, Identifying Classes and Objects, Importance of Proper Classification, Identifying Classes and Objects, Key abstractions and Mechanisms.

1. **Classes and Objects:**
2. **Nature of object**
3. **Relationships among objects**
4. **Nature of a Class**
5. **Relationship among Classes**
6. **Interplay of Classes and Objects**
7. **Identifying Classes and Objects**
8. **Importance of Proper Classification**
9. **Identifying Classes and Objects**
10. **Key abstractions and Mechanisms.**

Classes and objects:

Design a complex software system our basic building blocks are classes and objects.

An object is an abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it,or both

- ✓ Objects have an internal state that is recorded in a set of attributes.
- ✓ Objects have a behavior that is expressed in terms of operations. The execution of operations changes the state of the objects and/or stimulates the execution of operations in other objects.
- ✓ Objects (at least in the analysis phase) have an origin in a real world entity.

Classes represents group of objects which have the same behavior and information structures.

- ✓ Every object is an instance of single class.
- ✓ Class is a kind of type, an ADT(but with data),or an 'entity'(but with methods).
- ✓ Classes are the same in both analysis and design.
- ✓ A class defines the possible behaviors and the information structure of all its object instance.

[1]THE NATURE OF OBJECT:

The ability to recognize physical objects is a skill that humans learn at a very early age. defined an object as a tangible entity that exhibits some well-defined behavior.

What Is and What Isn't an Object

From the perspective of human cognition, an object is any of the following:

- A tangible and/or **visible thing.**
- Something that may be **apprehended intellectually.**
- Something toward which thought or **action is directed.**

Consider for a moment a manufacturing plant that processes composite materials for making such diverse items as bicycle frames and airplane wings. Manufacturing plants are often divided into *separate shops*: mechanical, chemical, electrical, and so forth.
For **example,** we might say that a man (an object) loves his wife (another object), or that a particular cat (yet another object) is gray.

"An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable."

State:
"The state of an object encompasses all of the (usually static) properties of tbe object plus tbe current (usually dynamic) values of each of these properties."

Example: Consider an abstraction of an employee record. Figure 3–1 depicts this abstraction using the Unified Modeling Language notation for a class.

Each part of this abstraction denotes a particular property of our abstraction of an employee. This abstraction is not an object because it does not represent a

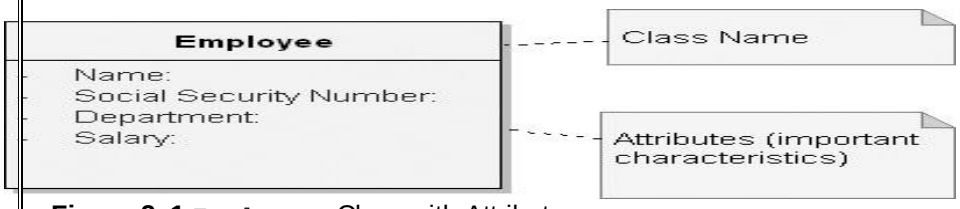


Figure 2–1 `Employee` Class with Attributes

specific instance. When made specific, we may have, for example, two distinct objects: Tom and Kaitlyn, each of which takes up some amount of space in memory (see Figure 2–2).

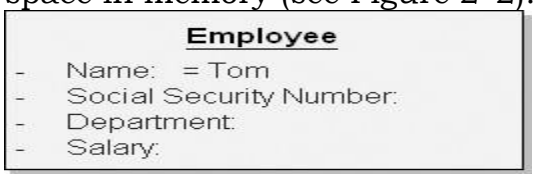


Figure 2–2 Employee Objects Tom and Kaitlyn

It is good engineering practice to encapsulate the state of an object rather than expose it. For example, we might change the abstraction (class) as shown in Figure 3–3.

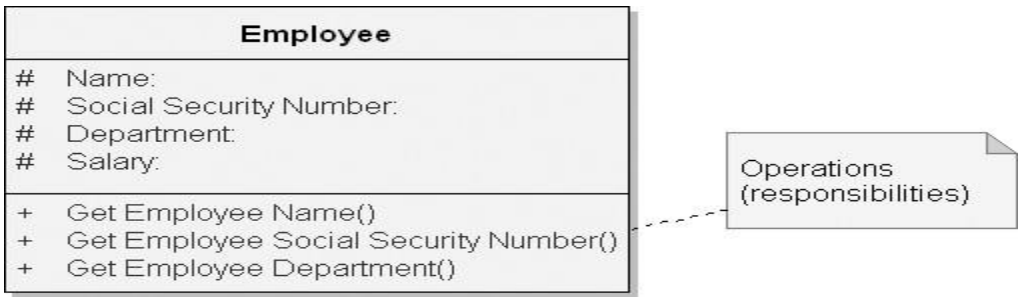


Figure 3–3 Employee Class with Protected Attributes and Public Operations

\+ Public

\- Private

\# Protected

- Class representation is hidden from all other outside clients. Changing class representation will not break outside code.
- All the clients have the right to retrieve the name. social security No and department of an employee.
- Only special clients(subclass) have permission to modify the values of these properties as well as salary. Thus all objects within a system encapsulate some state.

Behavior:

The Meaning of Behavior:

"*Behavior is how an object acts and reacts, in terms of its state changes and message passing".*

*T*he behavior of an object represents its outwardly visible and testable activity.

"The state of an object represents the cumulative results of its bebavior."

Operations : An operation denotes a service that a class offers to its clients. A client typically performs five kinds of operations upon an object.

- **Modifier** - An operation that *alters the state of an object*.
- **Selector** - An operation that *accesses the state of an object*, but does not alter the state.

- **Iterator** - An operation that permits all parts of an object to be accessed in some well-defined order.

Two other kinds of operations are common; they represent the infrastructure necessary to *create* and *destroy* instances of a class.

- **Constructor-** An operation that *creates an object* and/or initializes its state.
- **Destructor** - An operation that frees the state of an object and/or *destroys the object itself*.

- In C++, constructors and destructors are declared as part of the definition of aclass, whereas in Java there are constructors, but no destructors.
- In Smalltalk, such operations are typically part of the protocol of a meta class (i.e., the class of a class).

Roles and Responsibilities:

- Collectively, all of the methods associated with a particular object comprise its protocol.
- The protocol of an object thus defines the envelope of an object's allowable behavior and so comprises the entire static and dynamic view of the object.
- For most nontrivial abstractions, it is useful to divide this larger protocol into logical groupings of behavior.

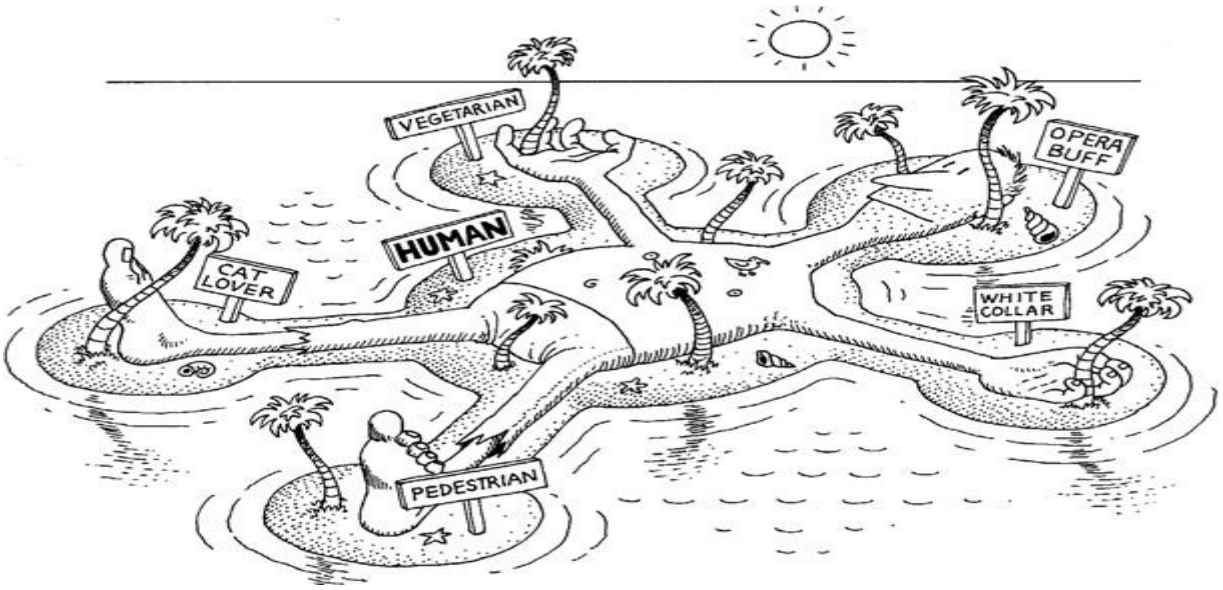


Figure Objects can play many different roles.

"Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports".

Indeed, most interesting objects play many different roles during their lifetime. Consider the following examples

1) A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money.

2) To a trader, a share of stock represents an entity with value that may be bought or sold.
3) In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

Objects as Machines: The existence of state within an object means that the order in which operations are invoked is important. This gives rise to the idea that each object is like a tiny, independent machine.

Identity

Semantics : *"Identity is that property of an object which distinguishes it from all other objects".*

Example 3–2 demonstrates the importance of maintaining the identity of the objects you create and shows how easily the identity can be irrecoverably lost.

state and location. Each display item has a location designated by the coordinates x and y.

- Let us assume we instantiate a number of DisplayItem classes as indicated in Figure 3–4a.
- Specifically, the manner in which we instantiate these classes sets aside four locations in memory whose names are item1,item2, item3, and item4, respectively.
- Here, item1 is the name of a distinctDisplayItem object, but the other three names each denote a pointerto a DisplayItem object.
- Only item2 and item3 actually point to distinctDisplayItem objects (because in their declarations we allocated a new DisplayItem object); item4 designates no such object.
- Furthermore, the names of the objects pointed to by item2 and item3 are anonymous: We can refer to these distinct objects only indirectly, via their pointer value.

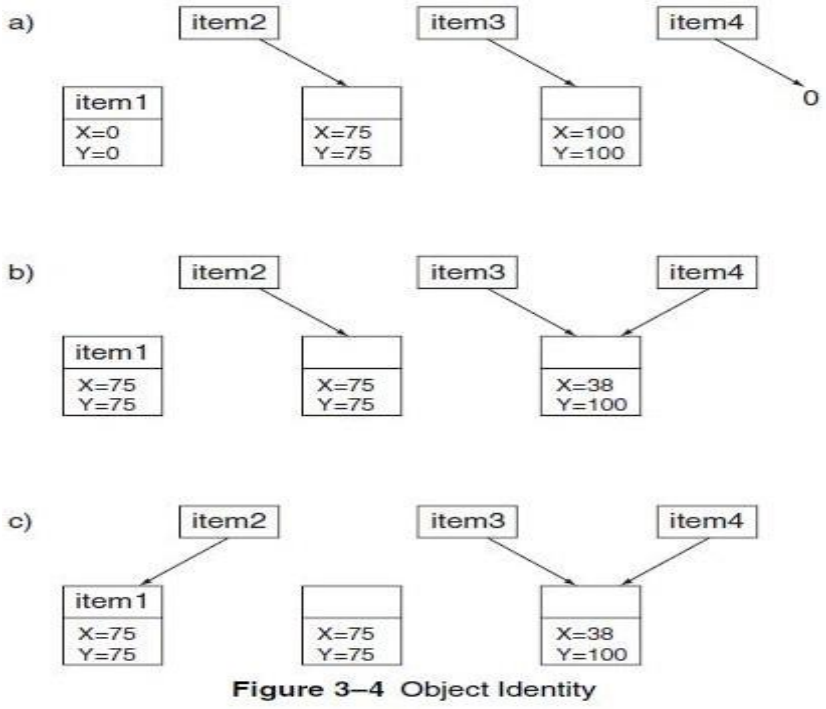


Figure 3–4 Object Identity

The lifetime of an object extends from the time it is first created (and thus first consumes space) until that space is reclaimed . To explicitly create an object, we must either declare it or allocate it.

1. Declaring an object (such as **item1** in our earlier example) creates a new instance on the stack.
2. Allocating an object (such as **item3**) creates a new instance on the heap. In C++, in either case, whenever an object is created, its constructor is automatically invoked.
3. whose purpose is to allocate space for the object and establish an initial stable state. In languages such as Smalltalk, such constructor operations are actually a part of the object's metaclass, not the object's class.

[2] Relationships Among Objects:

"Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires".

"a collection of parts having an inherent tendency to fall to earth, and requiring constant effort and supervision to stave off that outcome"

Kinds of Relationships:

Objects contribute to the behavior of a system by collaborating with one another. The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results. We have found that two kinds of

object hierarchies are of particular interest in object-oriented analysis and design, namely:

- Links
- Aggregation

Links:

- The term *link* derives from Rumbaugh et al., who define it as a "physical or conceptual connection between objects".
- An object collaborates with other objects through its links to these objects.
- Stated another way, a link denotes the specific association through which one object (the client) applies the services of another object (the supplier),

As a participant in a link, an object may play one of three roles:

- **Actor** - An object that can operate upon other objects but is never operated upon by other objects; in some contexts, the terms *active object* and *actor* are interchangeable
- **Server** - An object that never operates upon other objects; it is only operated upon by other objects
- **Agent** - An object that can both operate upon other objects and be operated upon by other objects;

In the context of Figure 3–5, FlowController acts as a controller object,DisplayPanel acts as a server object, and Valve acts as a proxy. Example 3–3illustrates how responsibilities can be properly separated across a group of collaborating objects.

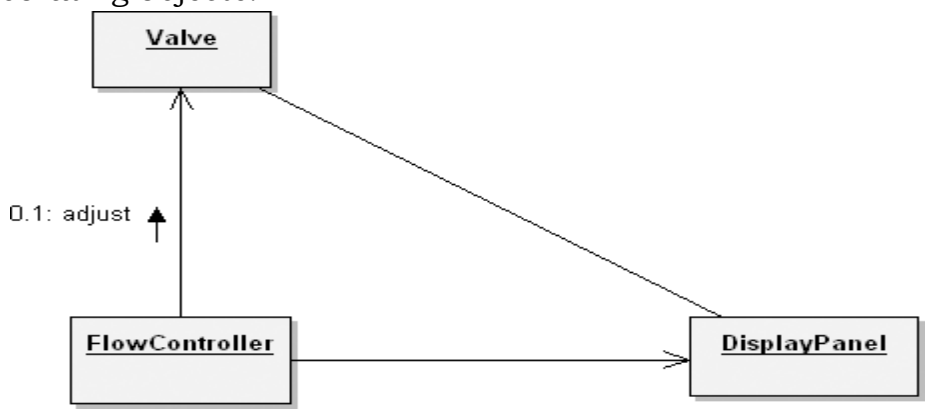


Figure 3–5 Links

Visibility: Consider two objects, **A** and **B**, with a link between the two. In order for **A** to send a message to **B**, **B** must be visible to **A** in some manner.

- The supplier object is *global to the client*.
- The supplier object is a *programmer* to some *operation of the client*.
- The supplier object is a *part of the client* object.

• The supplier object is a *locally declared* object in some operation of the client.

Synchronization :

Whenever one object passes a message to another across a link, the two objects are said to be *synchronized*. For objects in a completely sequential application, this synchronization is usually accomplished by simple method invocation.

However, when one *active object* has a link to a *passive one*, we must choose one of three approaches to synchronization.

• **Sequential** -The semantics of the *passive object are guaranteed* only in the presence of a *single active object* at a time.

• **Guarded** -The semantics of the passive object are guaranteed in the presence of *multiple threads of control*, but the active clients must *collaborate to achieve mutual exclusion.*

• **Synchronous**- The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the *supplier* guarantees mutual exclusion.

Aggregation:

Semantics:

- Where as links denote peer-to-peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the ***aggregate*)** to its parts (also known as its *attributes*), aggregation is a specialized kind of association.

For **example,** as shown in Figure 2–6, the object Temperature-Controller has a link to the object TemperatureRamp as well as to Heater. The object TemperatureController is thus the whole, and Heater is one of its parts.

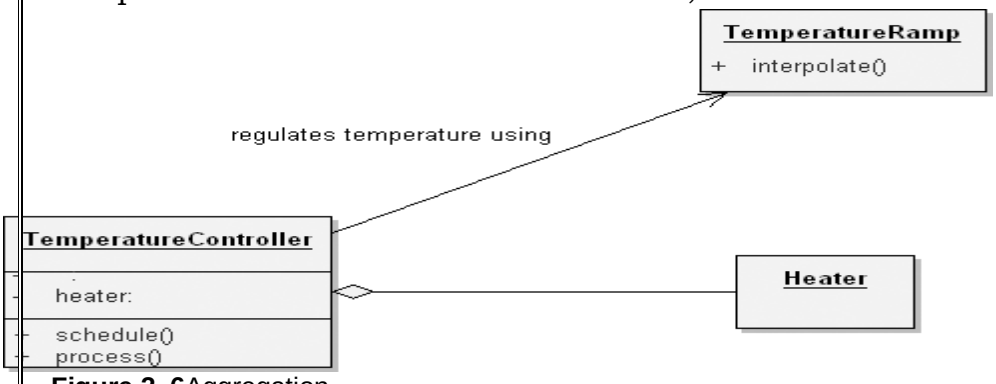


Figure 2–6 Aggregation

- Aggregation may or may not denote physical containment. For **example**, an airplane is composed of wings, engines, landing gear, and so on: This is a

case of physical containment.

- The relationship between a shareholder and his or her shares is an aggregation relationship that does not require physical containment.
- There are clear trade-offs between links and aggregation.
- Aggregation is sometimes better because it encapsulates parts as secrets of the whole.
- Links are sometimes better because they permit looser coupling among objects.

[3] The Nature of a Class:

- A class is a set of objects that share a common structure, common behavior, andcommon semantics.
- A single object is simply an instance of a class.
- Object is a concrete entity that exists in time and space but class represents only an abstraction.A class may be an object is not a class.

Interface and Implementation:

- The ***interface*** of a class provides its **outside view** and therefore *emphasizes* the abstraction while hiding its *structure and the secrets* of its behavior.
- The ***implementation*** of a class is its **inside view**, which encompasses the *secrets o*f its behavior.

The implementation of a class primarily consists of the implementation of all of the operations defined in the interface of the class.

We can further divide the *interface* of a class into three parts:

- **Public** - A declaration that is *accessible* to all clients
- **Protected** - A declaration that is accessible *only to the class* itself, its subclasses, and its friends
- **Private** - A declaration that is accessible only by classes itself .

Package: Adeclaration that is accessible only by classes in the same package

[4] Relationships Among Classes:

Kinds of Relationships:

The similarities and differences among the following classes of objects: flowers, daisies, red roses, yellow roses, petals, and ladybugs. We can make the following observations:

- A daisy is a kind of flower.
- A rose is a (different) kind of flower.
- Red roses and yellow roses are both kinds of roses.
- A petal is a part of both kinds of flowers.
- Ladybugs eat certain pests such as aphids, which may be infesting certain kinds

offlowers.

We establish relationships between *two classes* for one of two reasons:

- First, a class relationship might indicate some sort of sharing
- Second, a class relationship might indicate some kind of semantic connection.

There are three basic kinds of class relationships.:

- The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower , meaning that a rose is a specialized subclass of the more general class, flower.

- The second is whole/part, which denotes a "part of" relationship. Thus, a petal is not a kind of a flower; it is a part of a flower.

- The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers.

Specifically, most object-oriented languages provide direct support for some combination of the following relationships:

- • Association
- • Inheritance
- • Aggregation
- • Using
- • Instantiation
- • Metaclass.

An alternate approach to inheritance involves a language mechanism called *delegation*, in which objects are viewed as prototypes (also called *exemplars*) that delegate their behavior to related objects, thus eliminating the need for classes.

Association:

- Of the different kinds of class relationships, associations are the most general.
- The identification of associations among classes is describing how many classes/objects are taking part in the relationship.

As **example** for a vehicle,two of our key abstractions include the vehicle and wheels .As shown in figure 2.7,we may show a simple association between these two classes:

The class Wheel and the class Vehicle.

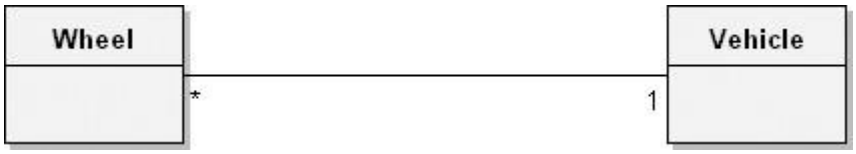


Figure 2–7 Association

Semantic Dependencies: An association only denotes a semantic dependency and does not state the direction of this dependency (unless otherwise stated, an association implies bidirectional navigation, as in ourexample), nor does it state the exact way in which one class relates to another (we can only imply these semantics by naming the role each class plays in relationship with the other).

Multiplicity*/Cardinality:**A one-to-many association, meaning that for each instance of the class **Sale**, there are zero or more instances of the class **Product**, and for each product, there is exactly one sale. This multiplicity denotes the ***cardinality of the association.

In practice, there are three common kinds of cardinality across an association:

- One-to-one
- One-to-many
- Many-to-many
    - A one-to-one relationship denotes a *very narrow* association.
    - A one-to-one relationship between the class **Sale** and the class **CreditCardTransaction**: each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale
    - Many-to-many relationships are also common.

Inheritance:

- Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships.
- Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.

To capture our decisions by building a hierarchy of classes, in which specialized classes inherit the structure and behavior defined by more generalized classes, as shown in Figure 3–8.

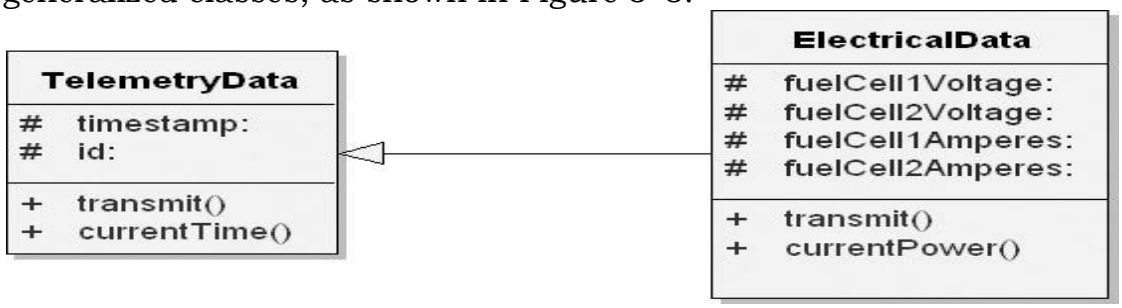


Figure 3–8 ElectricalData Inherits from the Superclass TelemetryData

As for the class ElectricalData, this class inherits the structure and behavior of the class TelemetryData but adds to its structure (the additional voltage data), redefines its behavior (the function transmit) to transmit the additional data, and can even add to its behavior (the function currentPower, a function to provide the current power level).

Single Inheritance: Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (*single inheritance*) or more (*multiple inheritance*) other classes.

our example: **TelemetryData** is a superclass of **ElectricalData**. Similarly, we call a class that inherits from one or more classes a subclass; **ElectricalData** is a subclass of **TelemetryData**. Inheritance therefore defines an "is a" hierarchy among classes, in which a subclass inherits from one or more superclasses.

A subclass typically augments or restricts the existing structure and behavior of its
superclasses. A subclass that augments its superclasses is said to use inheritance for extension.

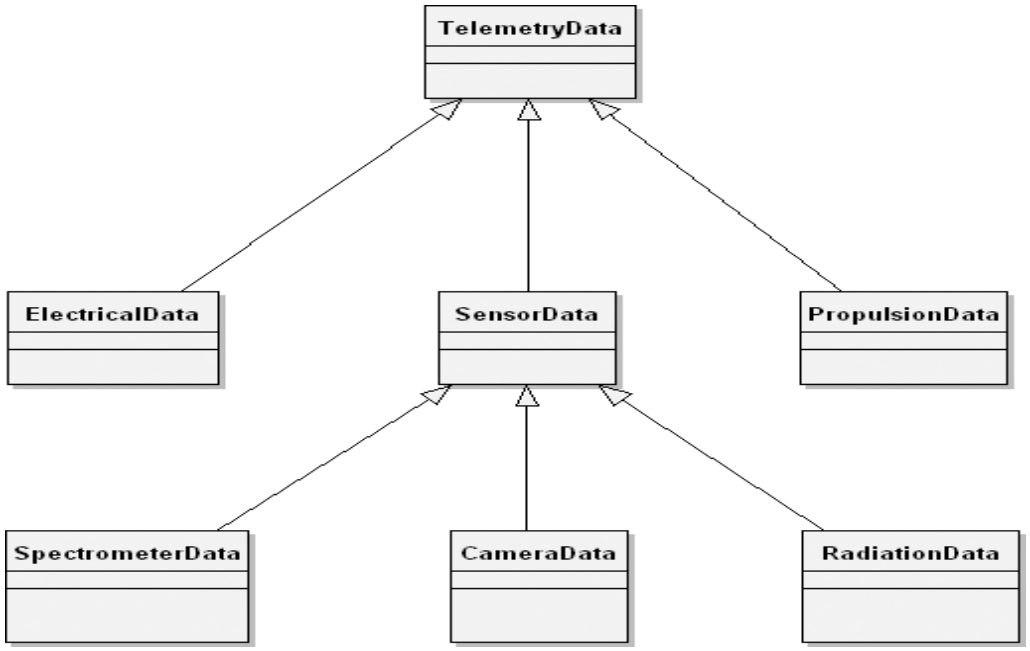


Figure 3–9 illustrates the single inheritance relationships deriving from the superclass TelemetryData.
Each directed line denotes an “is a” relationship.
For example, CameraData “is a” kind of SensorData, which in turn “is a” kind of TelemetryData.

Polymorphism:

- Polymorphism is a concept in type theory wherein a name may denote *instances of many different classes as long as they are related by some common superclass*.
- Any object denoted by this name is thus able to respond to some common set of operations in different ways.
- With polymorphism, an operation can be implemented differently by the classes in the hierarchy.

Figure 3–10 DisplayItem Class Diagram

Consider the class hierarchy in Figure 3–10, which shows the base class DisplayItem along with three subclasses named Circle, Triangle, and Rectangle. Rectangle also has one subclass, named SolidRectangle. In the class DisplayItem, suppose that we define the instance variable the Center (denoting the coordinates for the center of the displayed item), along with the following operations **.**

Multiple Inheritance:

- ✓ Consider for a moment how one might organize various assets such as savings accounts, real estate, stocks, and bonds. Savings accounts and checking accounts are both kinds of assets typically managed by a bank, so we might classify both of them as kinds of bank accounts, which in turn are kinds of assets.
- ✓ Stocks and bonds are managed quite differently than bank accounts, so we might classifystocks, bonds, mutual funds, and the like as kinds of securities, which in turn are also kinds of assets.

Figure 3–11 Multiple Inheritance

- ✓ Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance. Figure 3–11 illustrates such a class structure.
- ✓ Here we see that the class Security is a kind of Asset as well as a kind of InterestBearingItem.
- ✓ Similarly, the class BankAccount is a kind of Asset, as well as a kind of InsurableItem and InterestBearingItem.

Aggregation: We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

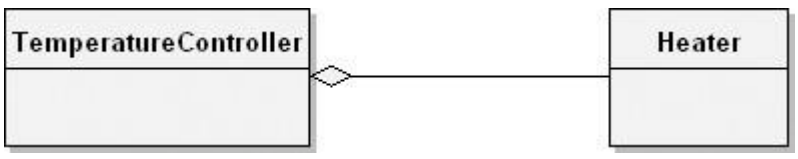


Figure 3–12 Aggregation
As we show in Figure 3–12, the class TemperatureController denotes the whole, and the class Heater is one of its parts.

Physical Containment:

- In the case of the class TemperatureController, we have aggregation as containment by value, a kind of physical containment meaning that the Heater object does not exist independently of its enclosing TemperatureController instance.
- Rather, the lifetimes of these two objects are intimately connected:
  i. When we create an instance of TemperatureController, we also create an instance of the class Heater.
  ii. When we destroy our TemperatureController object, by implication we also destroy the corresponding Heater object.

Dependencies:

- Aside from inheritance, aggregation, and association, there is another group of relationships called *dependencies*.
- A dependency indicates that an element on one end of the relationship, in some manner, depends on the element on the other end of the relationship.
- This alerts the designer that if one of these elements changes, there could be an impact to the other.

[5] The Interplay of Classes and Objects:

- ✓ Clases and object are separate yet *intimately related concepts.*
- ✓ Every object is the *instance of some class*, and *every class has zero or more instances.*
- ✓ The class of most objects is static, meaning that once an *object is created*, its *class is fixed*.

Example: The classes and objects in the implementation of an traffic control system. Some of the more important abstractions include *planes, flight plans, runways,* and air spaces.

Role of Classes and Objects in Analysis and Design:

During analysis and the early stages of design, the developer has two primary tasks:

1. Identify the classes and objects that form the *vocabulary of the problem domain.*
2. Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem.
   I. we call such classes and objects the *key abstractions* of the problem.
   II. we call these cooperative structures the *mechanisms* of the implementation.

- The focus of the developer must be upon the ***outside view*** of these key *abstractions and mechanisms.* This view represents the *logical framework* of the system, and therefore ***encompasses*** the ***class structure*** and ***object structure*** of the system.

- The *focus* is on the ***inside view*** of these *key abstractions and mechanisms*, involving their **physical representation.**

[6] The Importance of Proper Classification:

Classification and Object-Oriented Development

1) The identification of classes and objects is the hardest part of object- oriented analysis and design.
2) To recognize the key abstractions and mechanisms that form the vocabulary of our problem domain. discovery and invention are both problems of classification, and classification is fundamentally a problem of finding sameness.

Discovery: To recognize the key abstractions and mechanisms that form the vocabulary of our problem domain.
Invention:we desire generalized abstractions as well as new mechanisms that specify how objects collaborate discovery and inventions are both problems of classification.

"Classification helps us to identify generalization,specialization, and aggregation hierarchies among classes".

By recognizing the common pattems of interaction among objects, we come to invent the mechanisms that serve as the soul of our implementation.

The Difficulty ofClassification
Examples of Classification:

- ❖ The fact that intelligent classification is difficult is hardly new information. Since there are parallels to the same problems in object- oriented design.

- ❖ consider for a moment the problems of classification in two other scientific disciplines: biology and chemistry.

- ❖ Darwin's theory depended upon an intelligent classification of species and also the contemporary biology, classification denotes "the establishment of a hierarchical system.

- ❖ Classification by DNA is useful in distinguishing organisms that are structurally similar, but genetically very different.

The Incremental and Iterative Nature of Classification:

This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages

The development of models that support automatic implementation and theories that allow the generalization of the solution.

The incremental and iterative nature of classification directly impacts the construction of class and object hierarchies in the design of a complex software system.

In practice, it is common to assert a certain class structure early in a design and then revise this structure over time.

[7] Identifying Classes and Objects:

Classical and modern approaches.There are three general approaches to classification

- Classical categorization
- Conceptual clustering
- Prototype theory

Classical Categorization :

"All the entities that have a given property or collection of properties in common form a category. Such properties are necessary and sufficient to define the category.

Example:

1) Married people constitute a category: One is either married or not, and the value of this property is sufficient to decide to which group a particular person belongs. On the other hand, tall people do not form a category, unless we can agree to some absolute criteria for what distinguishes the property of tall from short.
2) Classical categorization comes to us first from Plato, and then from Aristotle through his classification of plants and animals, in which he uses a technique much akin to the contemporary children's game of Twenty Questions.

"We can name a thing according to the knowledge we have of its nature from its properties and effects"

Conceptual Clustering :

Conceptual clustering is a more modern variation of the classical approach, and largely derives from attempts to explain how knowledge is represented.

Conceptual clustering is closely related to fuzzy (multivalue) set theory,

in which objects may belong to one or more groups, in varying degrees of fitness.

Prototype Theory :

It is more recent approach of classify where a class of objects is represented by a prototypical object, and an object is considered to be a member of this class if and only if it resembles this prototype in significant ways.

1) **Example:** category like games, not in classical since no single common properties shared all games, Classify chairs (beanbag,chairs,barber chairs, in prototypes theory, we group things according to the degree of their relationship to concrete prototypes.

There approaches to classify provide the theoretical foundation of objected analysis by which we identify classes and objects in order to design a complex software ststem(CSS).

Object-Oriented Analysis:

- The boundaries between analysis and design are fuzzy, although the focus of each is quite distinct.
- In analysis, the focus is to fully analyze the problem at hand and to model the world by discovering the classes and objects that form the vocabulary of the problem domain.
- In design, we invent the abstractions and mechanisms in our models that provide the design of the solution to be built.

Classical Approaches :

- A number of methodologists have proposed various sources of classes and objects, derived from the requirements of the problem domain.
- We call these approaches *classical* because they derive primarily from the principles of classical categorization.

For example:

Shlaer and Mellor suggest that candidate classes and objects usually come from one of the following sources.

- **Tangible** -things Cars, telemetry data, pressure sensors.
- **Roles** -Mother, teacher, politician.
- **Events** -Landing, interrupt, request.
- **Interactions**- Loan, meeting, intersection.

From the perspective of database modeling, Ross offers a similar list.

- **People** - Humans who carry out some function
- **Places** - Areas set aside for people or things
- **Things** - Physical objects, or groups of objects, that are tangible
- **Organizations** - Formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals.

• **Concepts** - Principles or ideas not tangible *per se*; used to organize or keep track of business activities and/or communications.
• **Events** - Things that happen, usually to something elsen at a given date and time, or as steps in an ordered sequence.
Coad and Yourdon suggest yet another set of sources of potential objects

- Structure "Is a" and "part of" relationships.
- Other systems External systems with which the application interacts.
- Devices Devices with which the application interacts.
- Events remembered A historical event that must be recorded.
- Roles played The different roles users play in interacting with the application.
- Locations Physical locations, offices, and sites important to the application.
- Organizational units Groups to which users belong.

At a higher level of abstraction, Coad introduces the idea of subject areas, which are basically logical groups of classes that relate to some higher-level system function.

Behavior Analysis :

Dynamic behavior also also be one of the primary source of analysis of classes and objects things can be grouped that have common responsibilities and form hierarchies of classes(including superclasses and subclasses) .

- A function point is *"defined as one end-user business function*".
- A business function represents some kind of output, inquiry, input, file, or interface.
- A function point is any relevant outwardly-visible and testable behavior of the system.

Domain Analysis: Domain analysis seeks to identify the classes and objects that are common to all applications within a given domain, such as patient record tracking, bond trading, compilers, or missile etc.
Domain analysis defined as an attempt to identify the objects,operations and relationships that are important to particular domain.

Moore and Bailin suggest the following steps in domain analysis.

- Construct a strawman generic model of the domain by consulting with domain experts.
- Examine existing systems within the domain and represent this understanding in a common format.
- Identify similarities and differences between the systems by consulting with domain experts.
- Refine the generic model to accommodate existing systems.

Vertical domain analysis: Applied across similar applications.

Horizontal domain analysis: Applied to related parts of the same application domain expert is like doctor in a hospital concerned with conceptual classification.

Use-Case Analysis :

That practice is use-case analysis, first formalized by Jacobson. Jacobson defines a use case as "A behaviourally related sequence of transactions performed by an actor in a dialogue with the system to provide some measurable value to the actor".

we can apply use case analysis as early as requirements analysis, at which time end users, other domain experts, and the development team enumerate the scenarios that are fundamental to the system's operation.

CRC Cards : CRC stands for Class/Responsibilities/Collaborators.

CRC cards have emerged as a simple yet marvelously effective way to analyze scenarios.

A CRC card is nothing more than a 3x5 index card,39 upon which the analyst writes - in pencil - the name of a class (at the top of the card), its responsibilities (on one half of the card) and its collaborators (on the other half of the card).

One card is created for each class identified as relevant to the scenario.

CRC cards can be spatially arranged to represent patterns of collaboration. The cards are arranged to represent generalization/specialization or aggregation hierarchies among the classes.

Informal English Description: Proposed by Abbott

- It is writing an English description of the problem (or a part of a problem) and then underlining the nouns and verbs.
- The nouns represent candidate objects, and the verbs represent candidate operations upon them.
- It is simple and because it forces the developer to work in the vocabulary of the problem space.

Structured Analysis:

A alternative to classical object-oriented analysis uses the products of structured analysis as a front end to object-oriented design.

This technique is appealing only because a large number of analysts are skilled in structured analysis.

The next *two techniques* involve analyzing individual data flow diagrams. A particular data flow diagram (using the terminology of Ward/Mellor .candidate objects may be derived from the following:

- External entities
- Data stores
- Control stores
- Control transformations.

Candidate classes derive from two sources:

- Data flows
- Control flows.

[8] Key Abstractions and Mechanisms:

Identifying Key Abstractions:

Finding Key Abstractions :

"A key abstraction is a class or object that forms part of the vocabulary of the problem domain".

The primary value of identifying such abstractions is that they give boundaries to our problem;

The identification of key abstractions involves two processes: **discovery** and **invention.**

Discovery : Discovery we come to recognize the abstractions used by domain experts if the domain expert.
Invention: Invention, we create new classes and objects that arenot necessarily part of the problem domain.

A developer of such a system uses these same abstractions, but must also introduce new ones, such as databases, screen managers, lists, queues, and so on.

Refining Key Abstractions:

- Once we identify a certain key abstraction as a candidate, we must evaluate it.
- Programmer must focus on questions. How we objects of this class created? What operations can be done on such objects.
- If there are not good answers to such questions, then the problem is to be though again and proposed solution is to be found out instead of immediately starting to code among the problems placing classes and objects at right

levels of abstraction is difficult.

This means that the programmer must focus on the questions: how are objects of this class created? can objects of this class be copied and/or destroyed? What operations can be done on such objects?.

We offer the following suggestions:

• **Object**s -should be named with proper noun phrases, such as **theSensor** or just simply
shape.
• **Classes-** should be named with common noun phrases, such as **Sensors** or **Shapes**.
• **Modifier**- operations should be named with active verb phrases, such as **draw** or **moveLeft.**
• **Selector** - operations should imply a query or be named with verbs the form "to be," such as **extentOf** or **is0pen**.

Identifying Mechanisms:

Finding Mechanisms:

A mechanism is a design decision about how collections of objects cooperate. Mechanisms thus represent patterns of behavior

For example, consider a system requirement for an automobile: pushing the accelerator should cause the engine to run faster, and releasing the accelerator should cause the engine to run slower.

Mechanism is selected is largely a matter of design choice. More specifically, any of the following designs might be considered:

• A mechanical linkage from the accelerator to the carburetor (the most common mechanism).

• An electronic linkage from a pressure sensor below the accelerator to a computer that controls the carburetor (a drive-by-wire mechanism).

• No linkage exists; the gas tank is placed on the roof of the car, and gravity causes fuel to flow to the engine.

Which mechanism a developer chooses from a set of alternatives is most often a result of other factors, such as cost, reliability, manufacturability, and safety. ***Mechanisms as Patterns:***

- ✓ Mechanisms are actually one in a spectrum of patterns we find in well-structured software systems.
- ✓ An idiom is an expression peculiar to a certain programming language or application culture, representing a generally accepted convention for use of

the language.

- ✓ many common programming tasks [are] idiomatic" and therefore identifying such idioms allows.
- ✓ "using C++ constructs to express functionality outside the language proper, while giving the illusion of being part of the language"

Examples of Mechanisms:

Consider the drawing mechanism commonly used in graphical user interfaces. Several objects must collaborate to present an image to a user: a window, a view, the model being viewed, and some client that knows when (but not how) to display this model.

Mechanisms thus represent a level of reuse that is higher than the reuse of individual classes. For **example**, the MVC paradigm is used extensively in the Smalltalk user interface. The MVC paradigm in turn builds on another mechanism, the dependency mechanism, which is embodied in the behavior of the Smalltalk base class Model and thus pervades much of the Smalltalk class library.

UNIT III

Introduction to UML: Why we model, Conceptual model of UML, Architecture, Classes, Relationships, Common Mechanisms, Class diagrams, Object diagrams.

INTRODUCTION TO UML:

1) WHY WE MODEL
2) CONCEPTUAL MODEL OF UML
3) ARCHITECTURE, CLASSES, RELATIONSHIPS
4) COMMON MECHANISMS
5) CLASS DIAGRAMS
6) OBJECT DIAGRAMS

- The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.
- The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.

Model:

A model is a simplification of reality. A model provides the blueprints of a system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

Why do we model:

We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

We build models of complex systems because we cannot comprehend such a system in its entirety.

1.2 Principles of Modeling:

There are four basic principles of model

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

1.2.1 Object Oriented Modeling: In software, there are several ways to approach a model. The two most common ways are

1. Algorithmic perspective
2. Object-oriented perspective

1.2.1.1Algorithmic Perspective

1) The traditional view of software development takes an algorithmic perspective.
2) In this approach, the main building block of all software is the procedure or function.
3) This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.
4) As requirements change and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

1.2.1.2 Object-oriented perspective

The contemporary view of software development takes an object-oriented perspective.
In this approach, the main building block of all software systems is the object or class.
A class is a description of a set of common objects.
Every object has identity, state, and behavior.
Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.

1.3.An Overview of UML:

- The Unified Modeling Language is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.
- The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.
  The UML is a language for
  - Visualizing
  - Specifying
  - Constructing
  - Documenting

- **Visualizing** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this

manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously.

- **Specifying** means building models that are precise, unambiguous, and complete.
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages.
- **Documenting** a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include

  - ✓ Requirements
  - ✓ Architecture
  - ✓ Design
  - ✓ Source code
  - ✓ Project plans
  - ✓ Tests
  - ✓ Prototypes
  - ✓ Releases

To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements:

1. Things
2. Relationships
3. Diagrams

Things in the UML

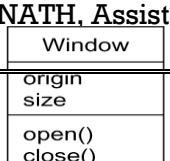
There are four kinds of things in the UML:

1) Structural things
2) Behavioral things
3) Grouping things
4) Annotational things

1)Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

1. Classes
2. Interfaces
3. Collaborations
4. Use cases
5. Active classes
6. Components
7. Node

Class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces.



Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.

Interface

Interface is a collection of operations that specify a service of a class or component.
An interface therefore describes the externally visible behavior of that element.
An interface might represent the complete behavior of a class or component or only a part of that behavior.

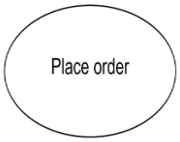
An interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations.

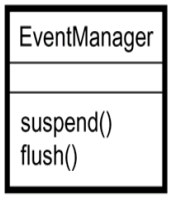
Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its nam

Usecase

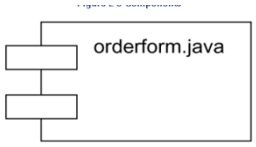
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name



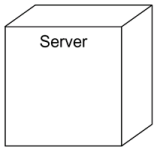
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs



Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube, usually including only its name



2)Behavioral Things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

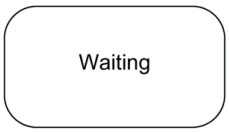
Interaction
state machine
Interaction

Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose
An interaction involves a number of other elements, including messages, action sequences and links
Graphically a message is rendered as a directed line, almost always including the name of its operation

State Machine

State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events

State machine involves a number of other elements, including states, transitions, events and activities
Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

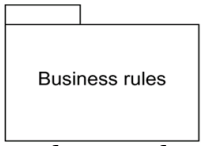


3)Grouping Things:-

1. Are the organizational parts of UML models. These are the boxes into which a model can be decomposed.
2. There is one primary kind of grouping thing, namely, packages.

Package:-

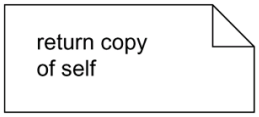
- A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package
- Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



4)Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.



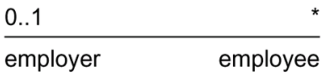
Relationships in the UML: There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

1)Dependency:-
Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing

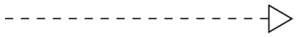
Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.

2)Association is a structural relationship that describes a set of links, a link being a connection among objects.
Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names



3)Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent

4)Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship



Diagrams in the UML

- **Diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- In theory, a diagram may contain any combination of things and relationships.
- For this reason, the UML includes nine such diagrams:
    - Class diagram
    - Object diagram
    - Use case diagram
    - Sequence diagram
    - Collaboration diagram
    - Statechart diagram
    - Activity diagram
    - Component diagram
    - Deployment diagram

Class diagram
A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
Class diagrams that include active classes address the static process view of a system.

Object diagram

- Object diagrams represent static snapshots of instances of the things found in class diagrams
- These diagrams address the static design view or static process view of a system
- An object diagram shows a set of objects and their relationships

Use case diagram

- A use case diagram shows a set of use cases and actors and their relationships
- Use case diagrams address the static use case view of a system.
- These diagrams are especially important in organizing and modeling the behaviors of a system.

Interaction Diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams

Interaction diagrams address the dynamic view of a system

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages.

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

Statechart diagram

- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities
- Statechart diagrams address the dynamic view of a system
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object

Activity diagram

An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system

Activity diagrams address the dynamic view of a system

They are especially important in modeling the function of a system and emphasize the flow of control among objects

Component diagram

- A component diagram shows the organizations and dependencies among a set of components.
- Component diagrams address the static implementation view of a system
- They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations

Deployment diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them
- Deployment diagrams address the static deployment view of an architecture.

Rules of the UML

The UML has semantic rules for

1. Names : What you can call things, relationships, and diagrams
2. Scope : The context that gives specific meaning to a name
3. Visibility : How those names can be seen and used by others
4. Integrity : How things properly and consistently relate to one another
5. Execution :What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

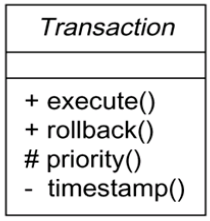
1. Elided Certain elements are hidden to simplify the view
2. Incomplete Certain elements may be missing
3. Inconsistent The integrity of the model is not guaranteed

Common Mechanisms in the UML

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

1) **Specification** that provides a textual statement of the syntax and semantics of that building block. The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion.
2) **Adornments** Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.



Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

Stereotype

- Stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem
- A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification
- A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones

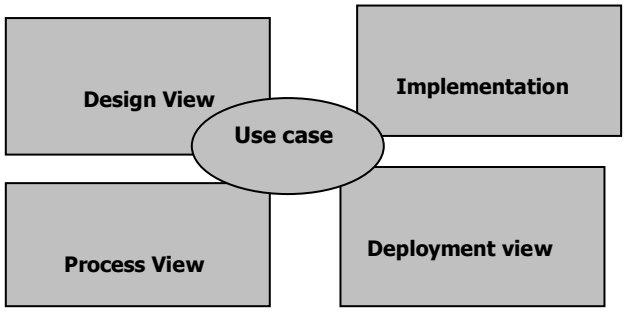
Architecture:

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.
Architecture is the set of significant decisions about.

The organization of a software system
The selection of the structural elements and their interfaces by which the system is composed
Their behavior, as specified in the collaborations among those elements
The composition of these structural and behavioral elements into progressively larger subsystems
The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition.
Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

Vocabulary **System Assembly**

Functionality **Configuration Mgmt**



Behavior
Performance **System topology**
Scalability **distribution delivery**
Throughput **installation**

Modeling a System's Architecture

Use case view

The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.
With the UML, the static aspects of this view are captured in use case diagrams.
The dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Design View

- The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.
- This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users.

Process View

- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system

Implementation View

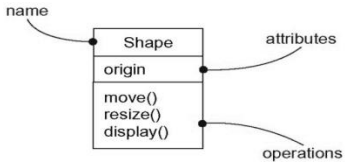
The implementation view of a system encompasses the components and files that are used to assemble and release the physical system.
This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system.

Deployment Diagram

The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes.
This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

1.Class:

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces.
- The UML provides a graphical representation of class



Graphical Representation of Class in UML
Terms and Concepts

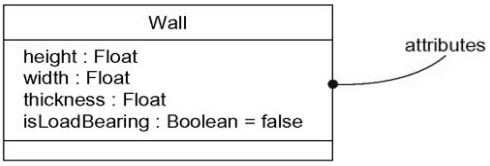
Names

Every class must have a name that distinguishes it from other classes.
A name is a textual string that name alone is known as a simple name;
a path name is the class name prefixed by the name of the package in which that class lives.

Simple Name **Path Name**

Attributes

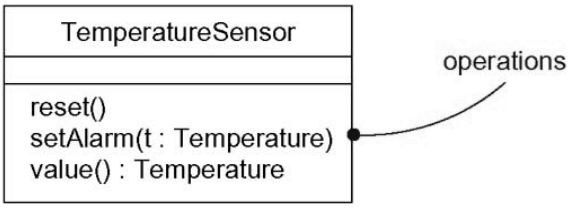
- An attribute is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of thing you are modeling that is shared by all objects of that class
- You can further specify an attribute by stating its class and possibly a default initial value



Attributes and Their Class

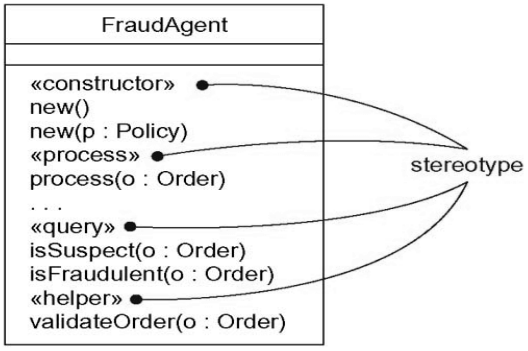
Operations

- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and a return type



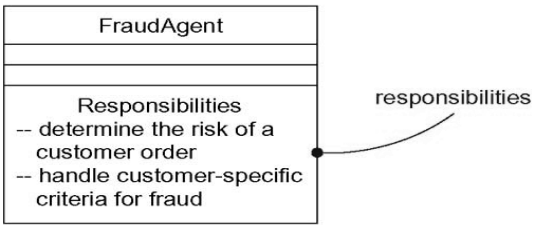
Organizing Attributes and Operations

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes



Responsibilities

- A Responsibility is a contract or an obligation of a class.
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon.



Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- They represent the things that are important to users and to implementers

- To model the vocabulary of a system.
    - Identify those things that users or implementers use to describe the problem or solution.
    - Use CRC cards and use case-based analysis to help find these abstractions.
    - For each abstraction, identify a set of responsibilities.
    - Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Modeling the Distribution of Responsibilities in a System

- Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.
- To model the distribution of responsibilities in a system

    - Identify a set of classes that work together closely to carry out some behavior.
    - Identify a set of responsibilities for each of these classes.
    - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
    - Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Modeling Nonsoftware Things

- Sometimes, the things you model may never have an analog in software
- Your application might not have any software that represents them
- To model nonsoftware things

    - Model the thing you are abstracting as a class.
    - If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
    - If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

Modeling Primitive Types

At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution.
Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types.
To model primitive types.

Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
If you need to specify the range of values associated with this type, use constraints.

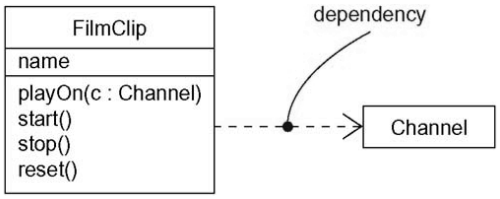
2.Relationships:

In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.
Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships
In object-oriented modeling, there are three kinds of relationships that are most important:

Dependencies
Generalizations
Associations

Dependency

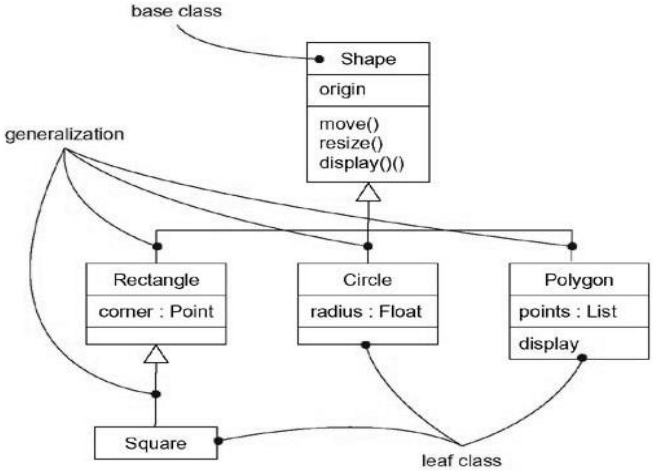
A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse.
Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

Generalization

- A generalization is a relationship between a general thing (called the super class or parent)and a more specific kind of that thing (called the subclass or child).
- generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations.
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent.

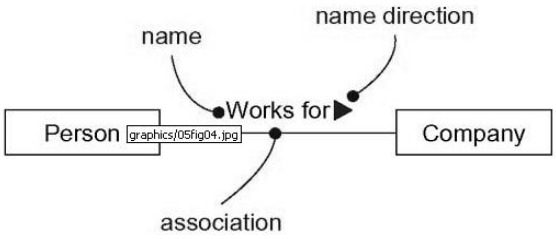


Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
- An association that connects exactly two classes is called a binary association
- An associations that connect more than two classes; these are called n-ary associations.
- Graphically, an association is rendered as a solid line connecting the same or different classes.
- Beyond this basic form, there are four adornments that apply to associations

Name

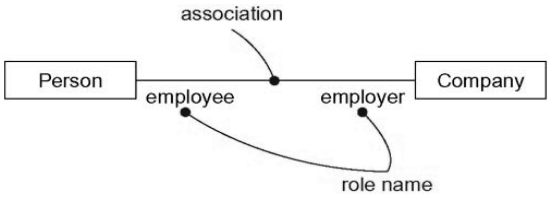
- An association can have a name, and you use that name to describe the nature of the relationship



Association Names

Role

- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link

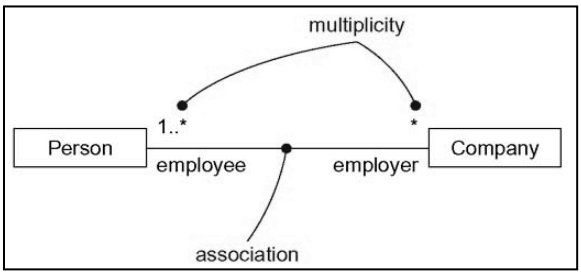


Role Names

Multiplicity

- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- This "how many" is called the multiplicity of an association's role

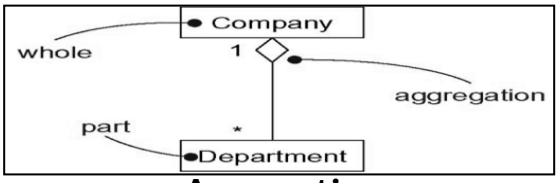
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

Aggregation

- Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").
- This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

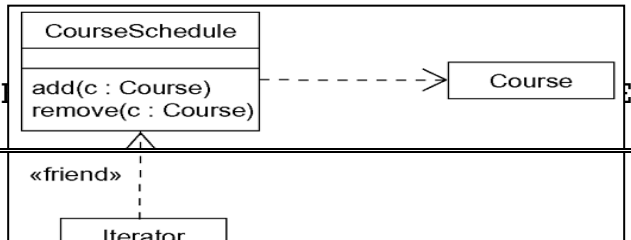
Common Modeling Techniques:

Modeling Simple Dependencies

The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.
To model this using relationship.

Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

The following figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university.
This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.
The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.

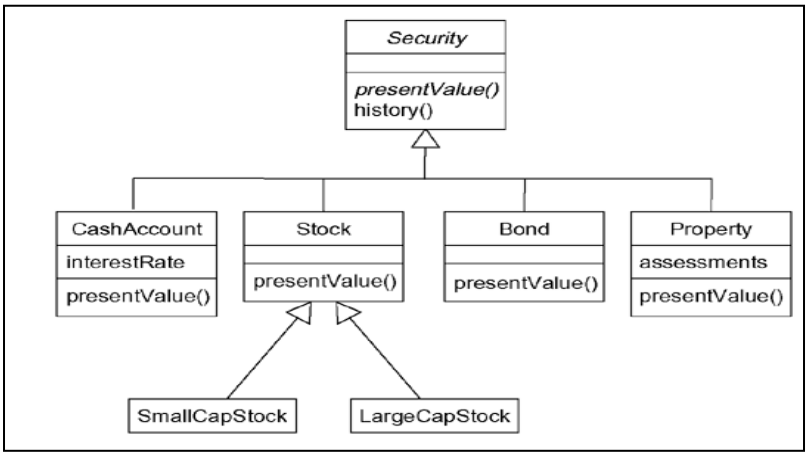


Dependency Relationships

Modeling Single Inheritance

To model inheritance relationships

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



Inheritance Relationships

Modeling Structural Relationships

- When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction
- Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children.
- Dependency and generalization relationships are one-sided.
- Associations are, by default, bidirectional; you can limit their direction
- Given an association between two classes, both rely on the other in some way, and you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

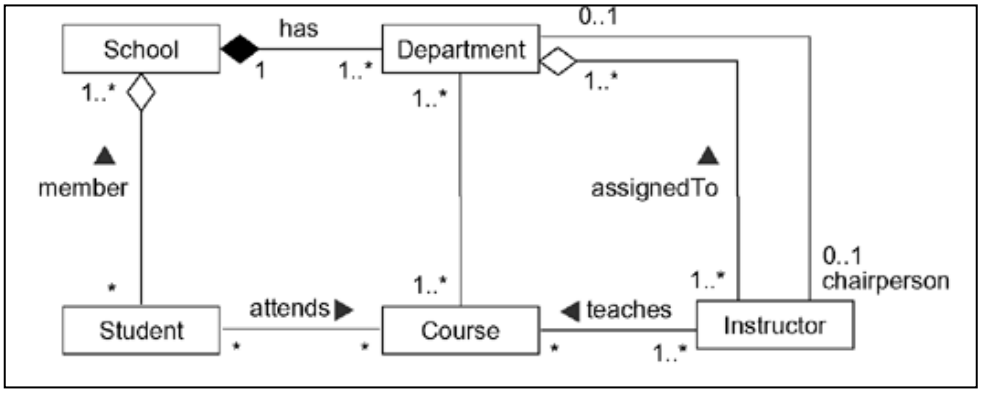
To model structural relationships

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.

- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.

- For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if it helps to explain the model).

- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole



Structural Relationships

Class Diagrams

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

Contents

- Class diagrams commonly contain the following things:
  - Classes
  - Interfaces
  - Collaborations
  - Dependency, generalization, and association relationships

- Like all other diagrams, class diagrams may contain notes and constraints

- Class diagrams may also contain packages or subsystems

Note: Component diagrams and deployment diagrams are similar to class diagrams, except that instead of containing classes, they contain components and nodes

Common Uses

- You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system
- We'll typically use class diagrams in one of three ways:

1. To model the vocabulary of a system
2. To model simple collaborations
3. To model a logical database schema

Modeling the vocabulary of a system

- Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries

Modeling simple collaborations

- A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.

Modeling logical database schema

- We can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

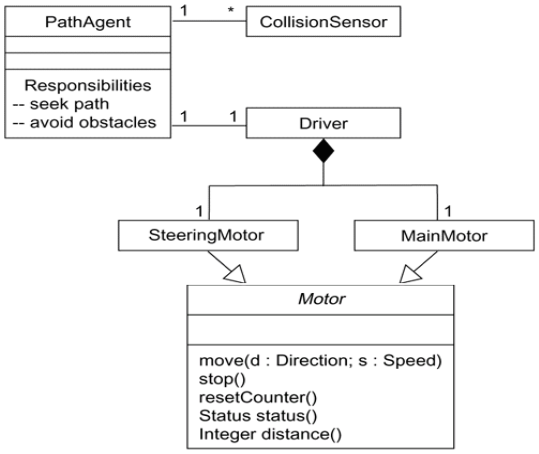
- When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.
- To model a collaboration

  - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

  - For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.

- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.

- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.



Modeling a Logical Database Schema

- The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.
- The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database these logical operations are generally turned into triggers or stored procedures.

- To model a schema,

  - Identify those classes in your model whose state must transcend the lifetime of their applications.

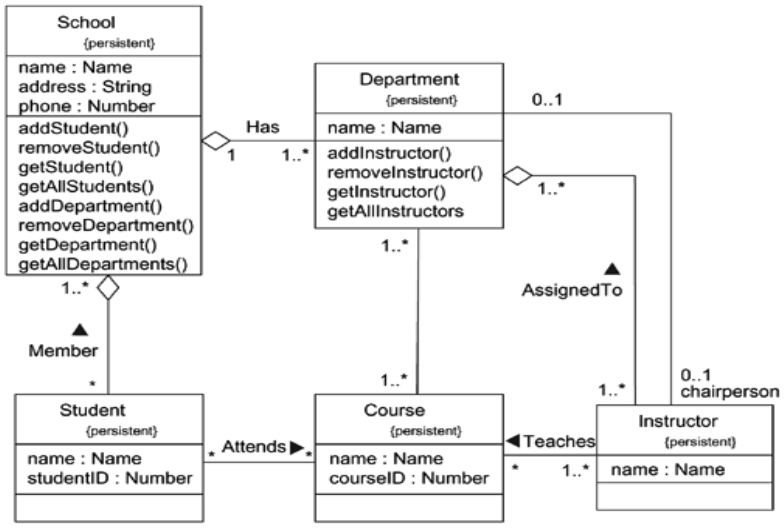
  - Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.

  - Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.

  - Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.

- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.

- Where possible, use tools to help you transform your logical design into a physical design.



Modeling a Schema

Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language
- Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language.
- To forward engineer a class diagram,

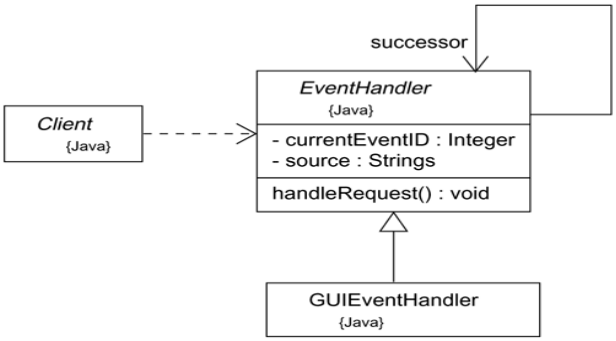
  - Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

  - Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with

multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).

- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.

- Use tools to forward engineer your models.



Forward Engineering

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models.
- Reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.
- To reverse engineer a class diagram,

    - Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

    - Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.

    - Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

Object Diagram

- An object diagram is a diagram that shows a set of objects and their relationships at a point in time.
- Graphically, an object diagram is a collection of vertices and arcs
- An object diagram is a special kind of diagram and shares the same common properties as all other diagrams—that is, a name and graphical contents that are a projection into a model

Contents

- Object diagrams commonly contain
    - Objects
    - Links
- Like all other diagrams, object diagrams may contain notes and constraints.
- Object diagrams may also contain packages or subsystems

Common Uses

- You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams
- When you model the static design view or static process view of a system, you typically use object diagrams in one way:
    - To model object structures

Modeling Object Structures

- Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time.
- An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram.

Common Modeling Techniques

Modeling Object Structures

- An object diagram shows one set of objects in relation to one another at one moment in time.
- To model an object structure,

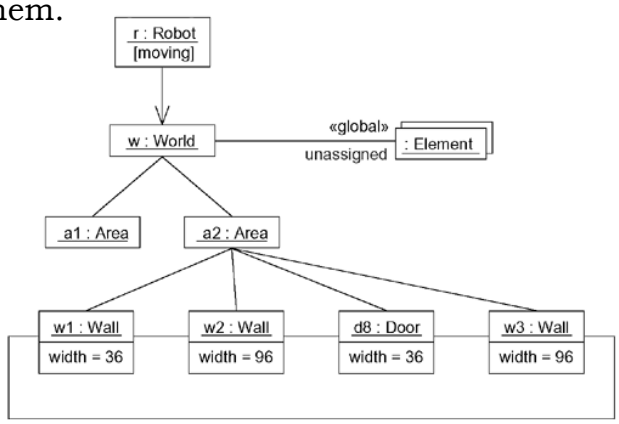
    - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

    - For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.

- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.

- Expose the state and attribute values of each such object, as necessary, to understand the scenario.

- Similarly, expose the links among these objects, representing instances of associations among them.



Modeling Object Structures

Forward and Reverse Engineering

- Forward engineering an object diagram is theoretically possible but pragmatically of limited value
- In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.
- Component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.

- Reverse engineering an object diagram is a very different thing
- To reverse engineer an object diagram,

  - Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.

  - Using a tool or simply walking through a scenario, stop execution at a certain moment in time.

  - Identify the set of interesting objects that collaborate in that context and render them in an object diagram.

  - As necessary to understand their semantics, expose these object's states.

- As necessary to understand their semantics, identify the links that exist among these objects.

- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

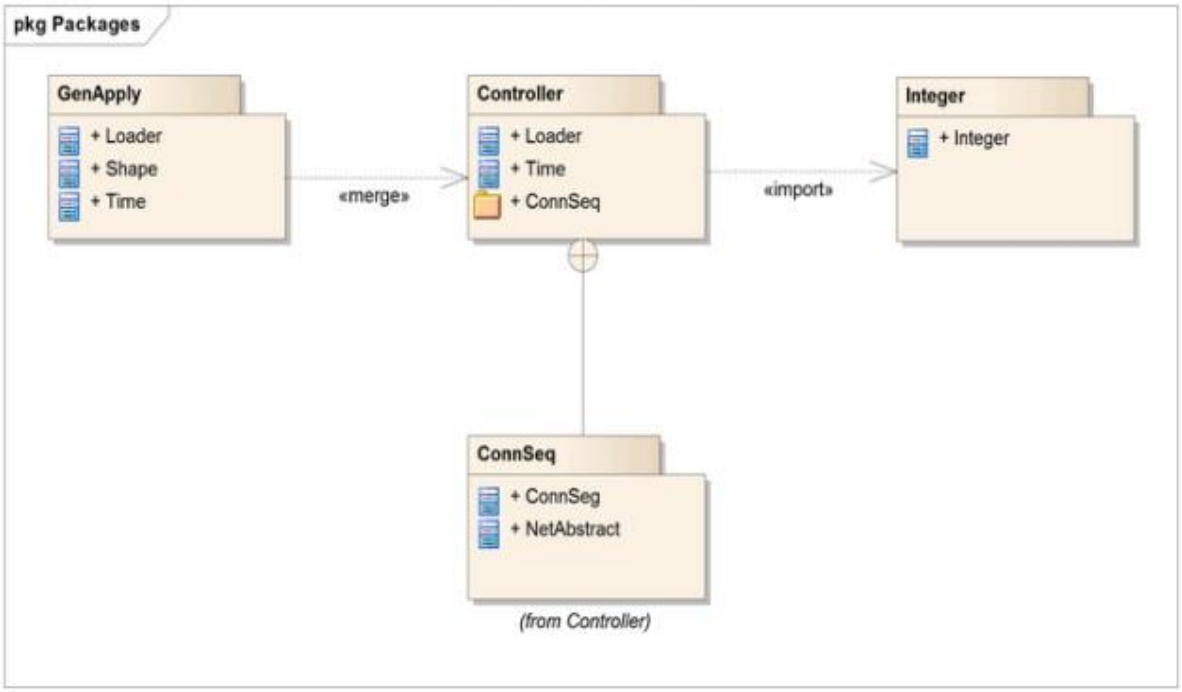
Unit-IV

Structural Modeling: Package Diagram, Composite Structure Diagram, Component Diagram, Deployment Diagram, Profile Diagram.

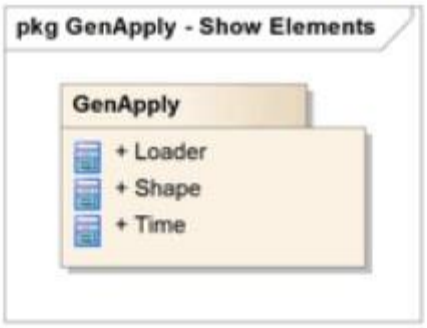
1) Package Diagram
2) Composite Structure Diagram
3) Component Diagram
4) Deployment Diagram
5) Profile Diagram.

[1] Package Diagram:

- Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the namespaces.
- The most common use for package diagrams is to organize use case diagrams and class diagrams, although the use of package diagrams is not limited to these UML elements. The following is an example of a package diagram.



- Elements contained in a package share the same namespace. Therefore, the elements contained in a specific namespace must have unique names.
- Packages can be built to represent either physical or logical relationships. When choosing to include classes in specific packages, it is useful to assign the classes with the same inheritance hierarchy to the same package.
- There is also a strong argument for including classes that are related via composition, and classes that collaborate with them, in the same package.
- Packages are represented in UML 2.1 as folders and contain the elements that share a namespace; all elements within a package must be identifiable, and so has a unique name or type.
- The package must show the package name and can optionally show the elements within the package in extra compartments.



Package Merge:

- A «merge» connector between two packages defines an implicit generalization between elements in the source package, and elements with the same name in the target package.
- The source element definitions are expanded to include the element definitions contained in the target.
- The target element definitions are unaffected, as are the definitions of source package elements that don't match names with any element in the target package.

Package Import:

The «import» connector indicates that the elements within the target package, which in this **example** is a single class, use unqualified names when being referred to from the source package. The source package's namespace gains access to the target classes; the target's namespace is not affected.

Nesting Connectors: The nesting connector between the target package and source packages shows that the source package is fully contained in the target package.

[2]Composite Structure Diagram:

- Composite Structure diagrams explore the internal organization of classes.
- It can be used to explicitly describe a class as a composition of other classes.
- The model can also show how the contained classes interact in the working implementation.

Themes

The diagrams covered in this presentation are used to highlight two main themes in Object Oriented Design:

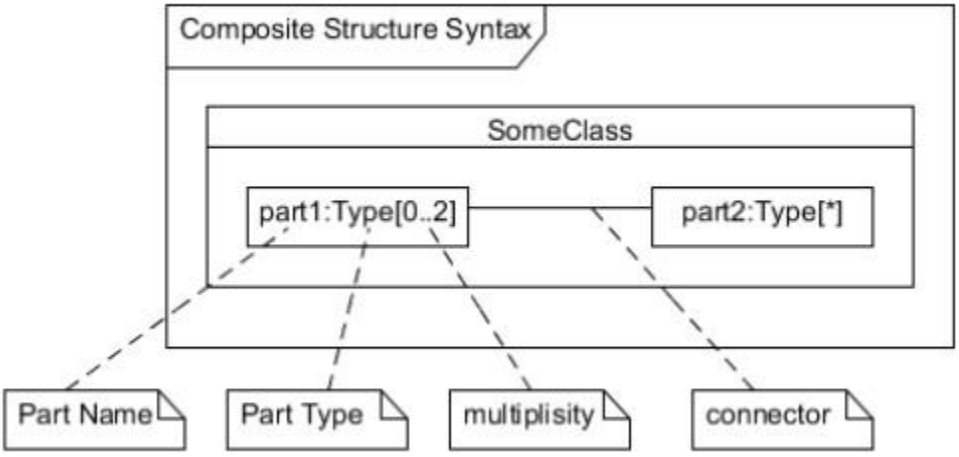
Aggregation: The act or result of forming an object configured from its component parts*

Classification: The act or result of removing certain distinctions between objects, so that we can see commonalities**

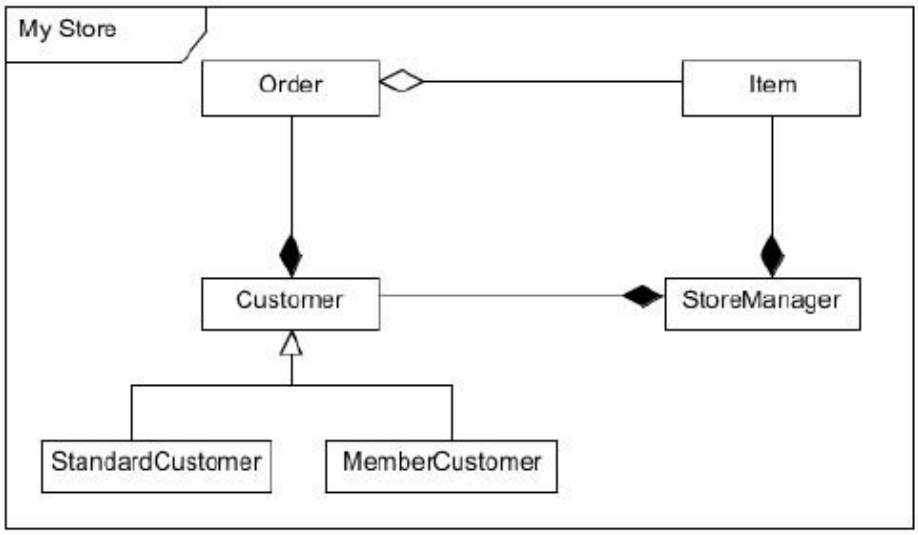
Composite Structure Diagrams and Aggregation:

- Composite Structure Diagrams allow the users to "Peek Inside" an object to see exactly what it is composed of.
- The internal actions of a class, including the relationships of nested classes, can be detailed.
- Objects are shown to be defined as a composition of other classified objects.

Syntax of a Composite Structure Diagram:



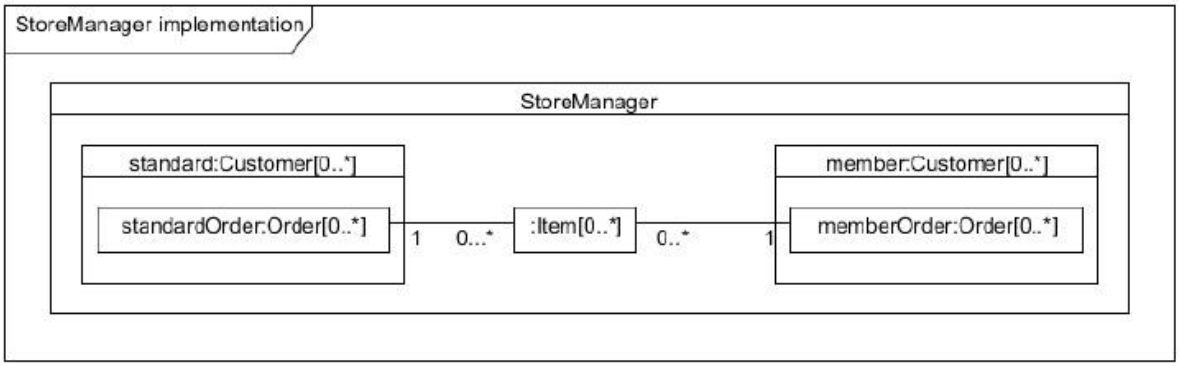
- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class but parts are not necessarily classes, a part is any element that is used to make up the containing class.



We are modeling a system for an online store. The client has told us that customers may join a membership program which will provide them with special offers and discounted shipping, so we have extended the customer object to provide a member and standard option.

We have a class for Item which may be aggregated by the Order class, which is composed by the Customer class which itself is composed by the StoreManager class. We have a lot of objects that end up within other objects.

Everything looks like it ends up inside StoreManager, so we'll create a composite structure diagram to really see what it's made of.

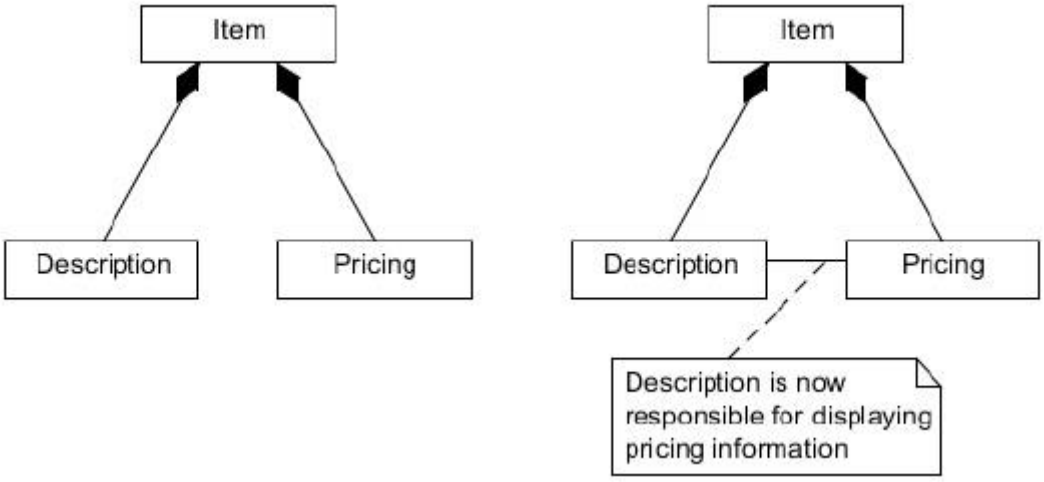


Here we see Store Manager from its own perspective, instead of the system as a whole. Store Manager directly contains two types of objects (Customer and Item) as is indicated by the two composition arrows on the class diagram.

What this diagram shows more explicitly is the inclusion of the subtypes of Customer. Notice that the type of both of these parts is Customer, as the store sees both as Customer objects.

We also see a connector which shows the relation between Item and Order. Order is not directly contained within the Store Manager class but we can show relations to parts nested within the objects it aggregates.

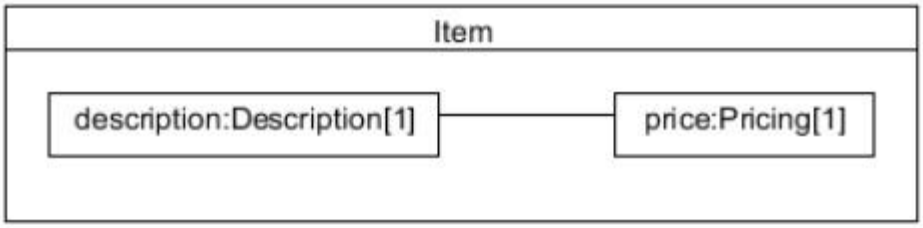
What a Class Diagram can't show:



We now have expanded our model to define the Item object as one which is composed of a Description object and a Pricing object. We then realize the implementation may be simplified if Description can access the pricing information, so we draw a reference to the Pricing object.

The problem is that this diagram is wrong. In a class diagram the reference between Description and Pricing is ambiguous. This does show that Description will have a reference to a Pricing object but this diagram does not specify that it be the Pricing object contained within the same Item object as itself.

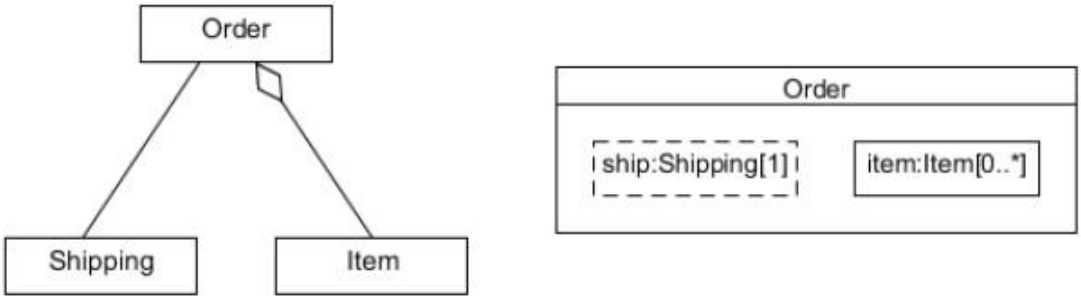
Composite Structure Diagrams are Contained:



The reference between the Description and Pricing objects is contained to objects that are composed by Item.

The specific implementations of an object's activity can be clearly modeled.

References to External parts:



We have seen examples of how Composite Structure diagrams are great at describing aggregation, but your models will also need to contain references to objects outside of the class you are modeling.

References to external objects are shown as a part with a dashed rectangle. Even though they reference object is outside of the class, the reference itself is within the modeled class and is an Important step in showing its implementation.

[3] Component Diagrams :

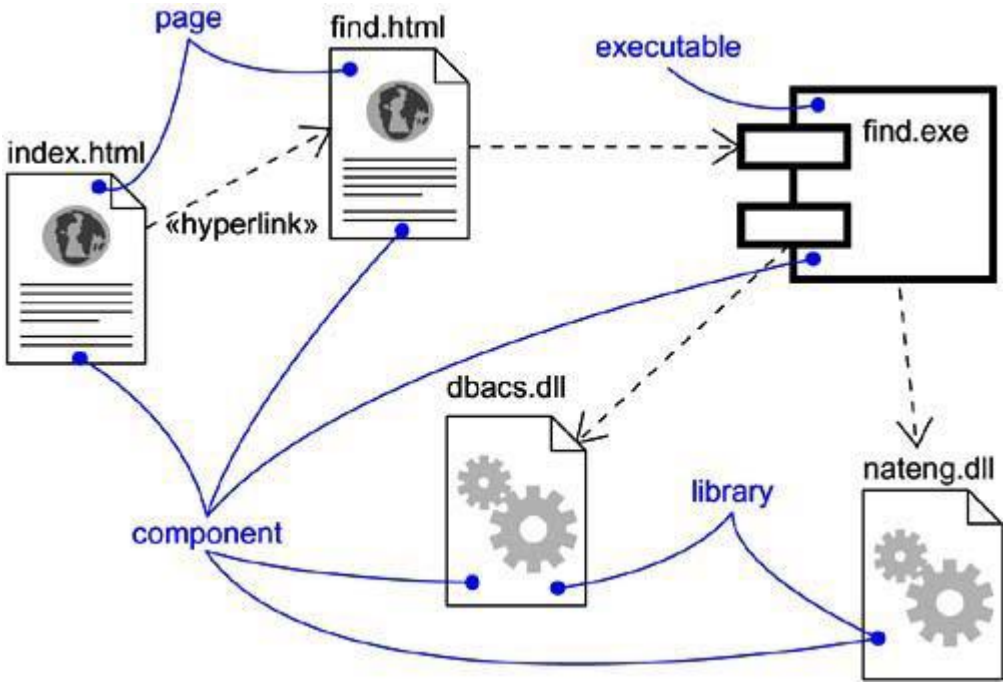


Figure 1: Component Diagram

Component Diagrams

- Component diagrams are used in modeling the physical aspects of object-oriented systems.
- A component diagram shows the organization and dependencies among a set of components.
- Component diagrams are used to model the static implementation view of a system.
- Component diagrams are essentially class diagrams that focus on a system's components.
- Graphically, a Component diagram is a collection of vertices and arcs.
- Component diagrams are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- Component diagrams commonly contain Components, Interfaces and Dependency, generalization, association, and realization relationships. It may also contain notes and constraints

Common Uses:component diagrams are used in one of four ways

1) To model source code
2) To model executable releases
3) To model physical databases
4) To model adaptable systems

1)Modeling Source Code

To model a system's source code,

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

Figure 2 shows five source code files.

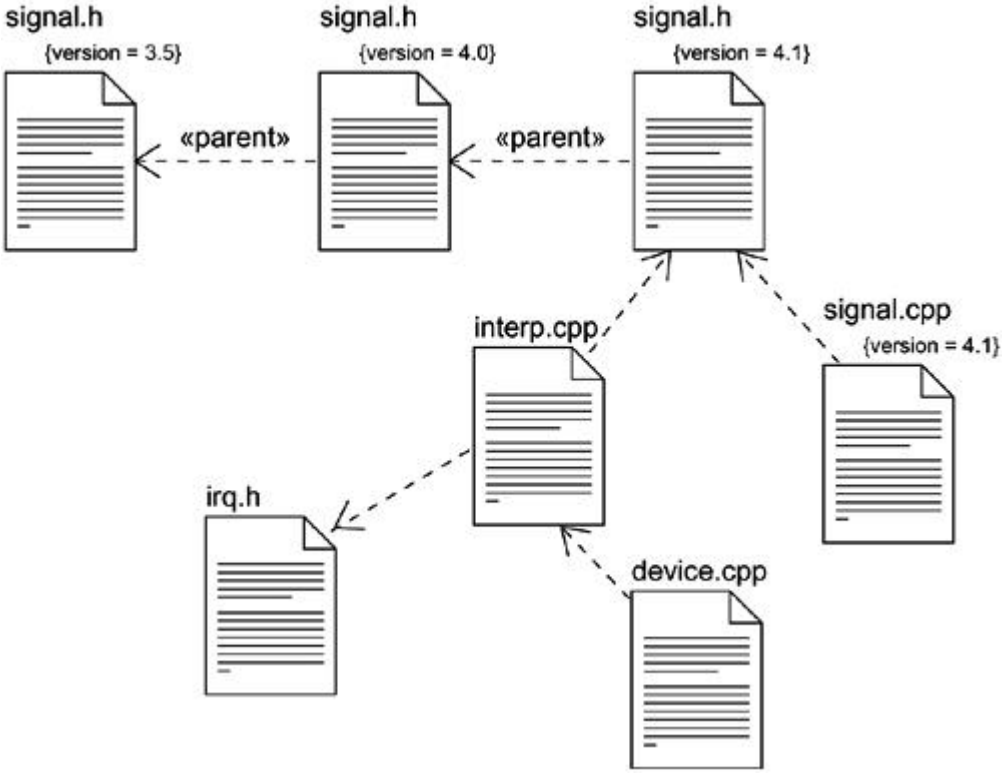


Figure 2: Modeling Source Code

2)Modeling an Executable Release:

To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

Figure 3 models part of the executable release for an autonomous robot.

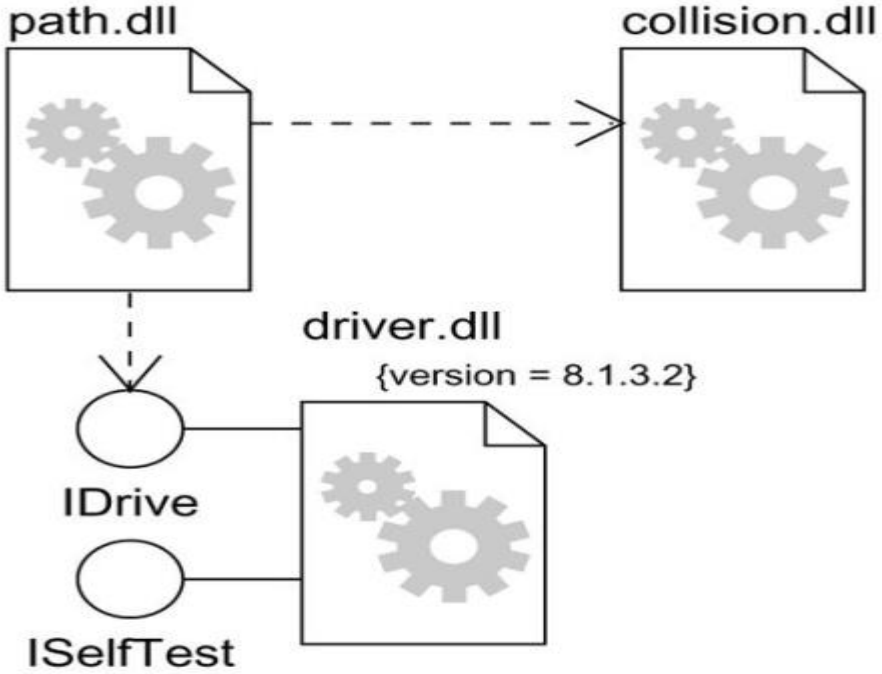


Figure 3: Modeling an Executable Release

3)Modeling a Physical Database

To model a physical database,

- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.

Figure 4 shows a set of database tables drawn from an information system for a school.

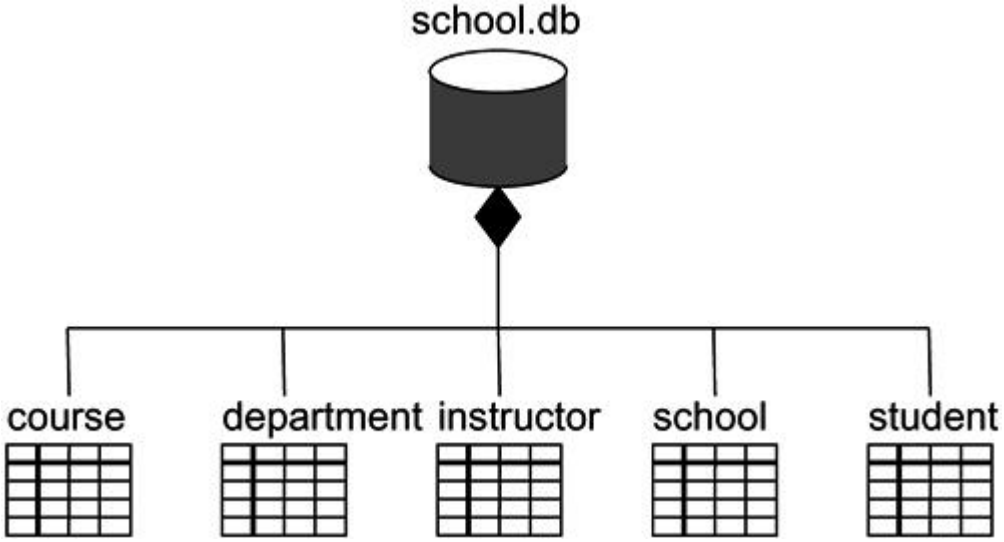


Figure 4: Modeling a Physical Database

4)Modeling Adaptable Systems:

To model an adaptable system,

- Consider the physical distribution of the components that may migrate from node to node.
- You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

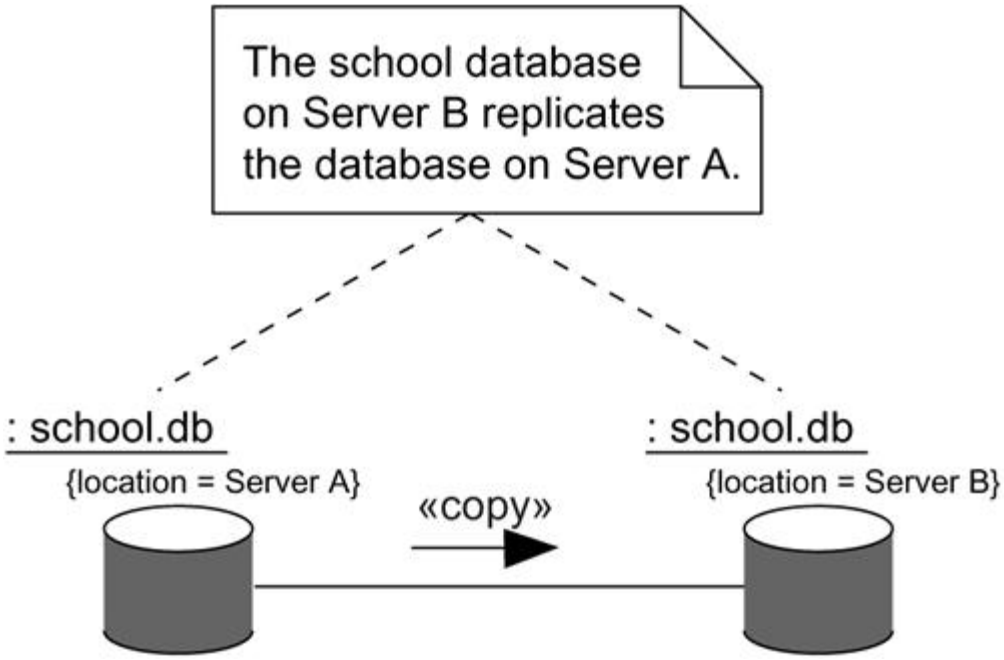


Figure 5: Modeling Adaptable Systems

[4]Deployment Diagrams :

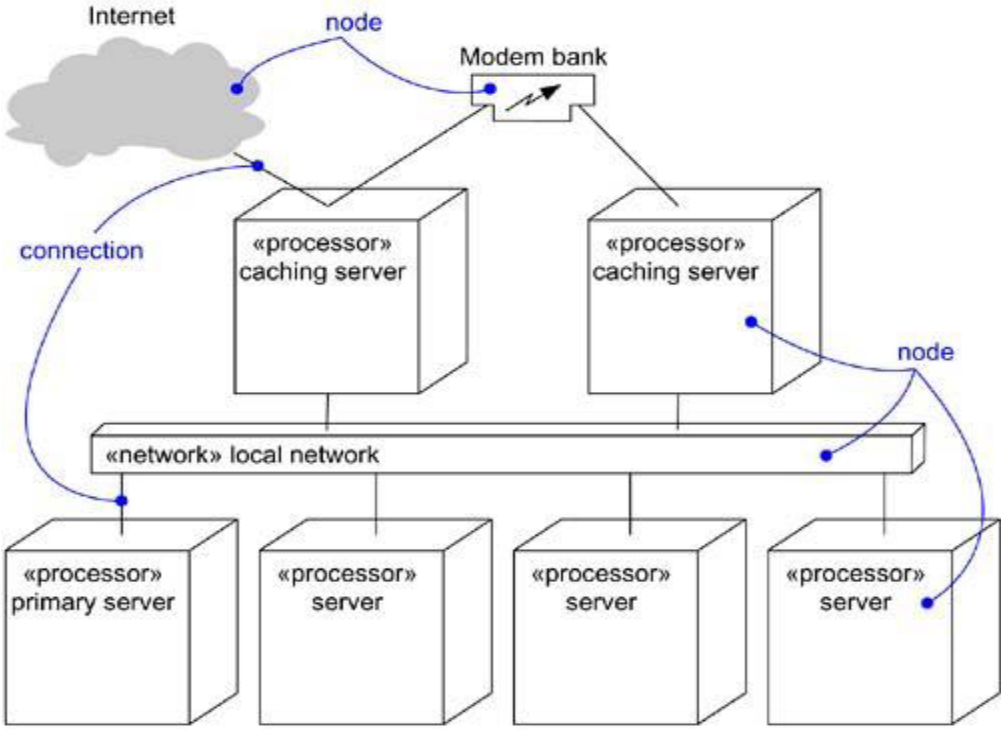


Figure 1: Deployment Diagram

Deployment Diagrams:

1) A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.

2) Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.
3) used to model the static deployment view of a system (topology of the hardware)
4) A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.
5) Graphically, a deployment diagram is a collection of vertices and arcs.
6) Deployment diagrams commonly contain Nodes and Dependency & association relationships. It may also contain notes and constraints.
7) Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

Common Uses: Deployment diagrams are used in one of three ways,

1) To model embedded systems
2) To model client/server systems
3) To model fully distributed systems

1)Modeling an Embedded System

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

Figure 2 shows the hardware for a simple autonomous robot.

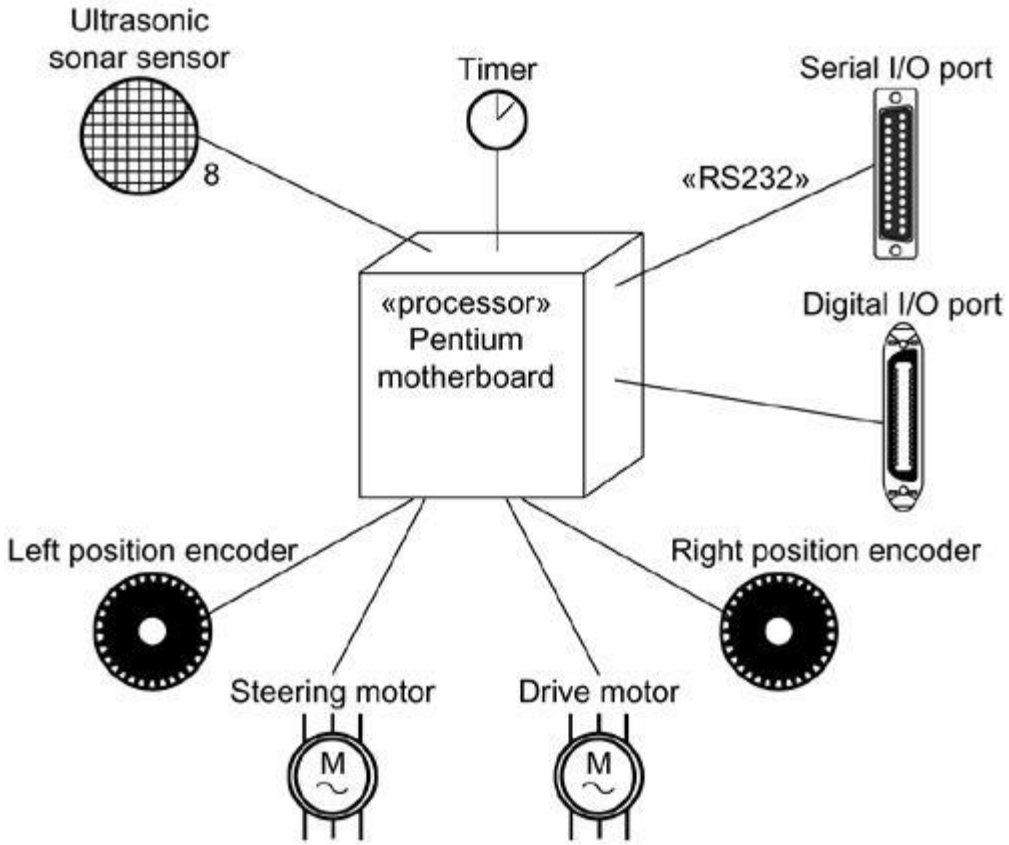


Figure 2: Modeling an Embedded System

2)Modeling a Client/Server System:

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For **example,** you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

Figure 3 shows the topology of a human resources system, which follows a classical client/server architecture.

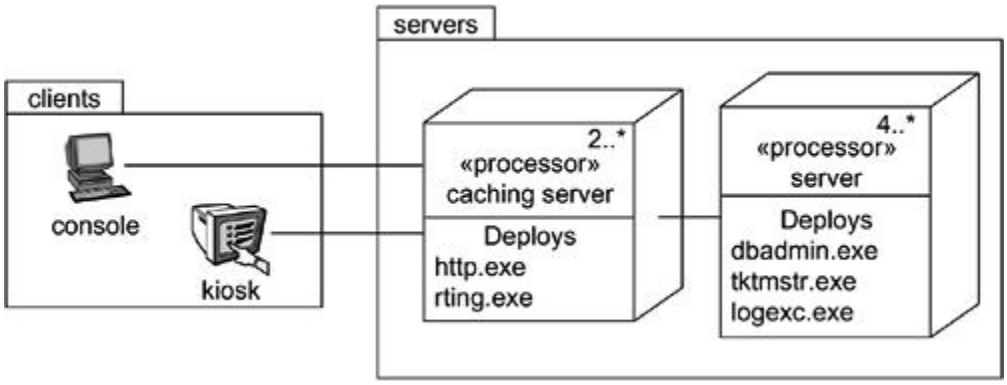


Figure 3: Modeling a Client/ Server System

3)Modeling a Fully Distributed System:

To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.
- When modeling a fully distributed system, it's common to reify the network itself as an node. eg:- Internet, LAN, WAN as nodes

Figure 4 shows the topology of a fully distributed system.

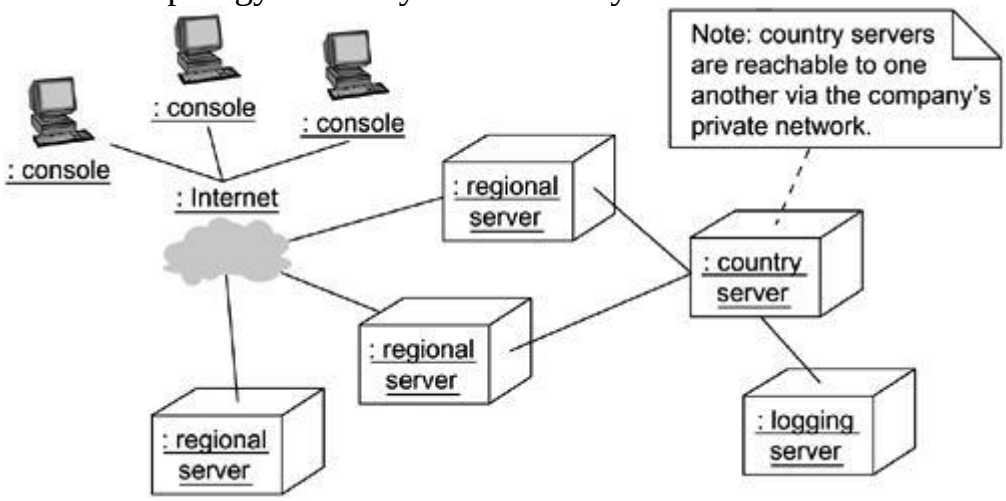
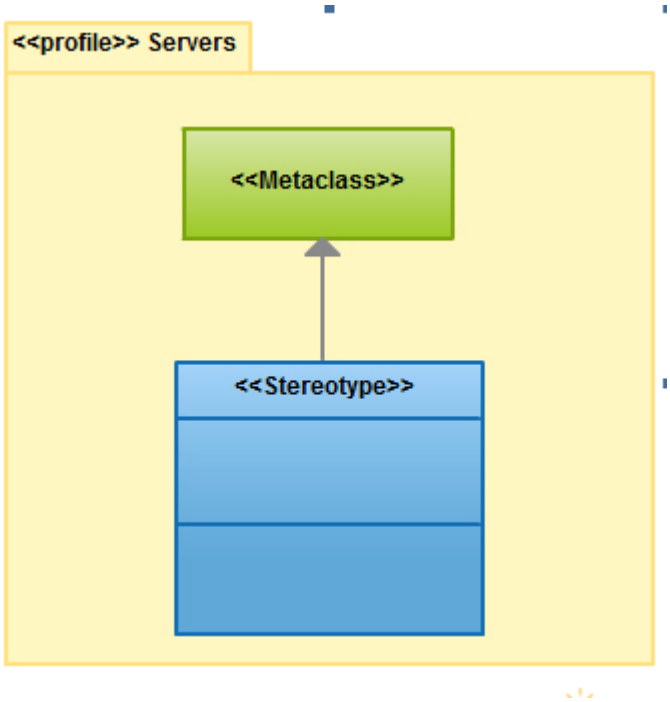


Figure 4: Modeling a Fully Distributed System

[5]Profile Diagram:

Profile diagram is a new diagram type introduced in UML 2. This is a diagram type that is very rarely used in any specification.



Profile diagram is **structure diagram** which describes **lightweight extension mechanism** to the UML by defining custom **stereotypes**, **tagged values**, and constraints. Profiles allow adaptation of the UML metamodel for different:

- ✓ *platforms*, such as Java Platform, Enterprise Edition (Java EE) or Microsoft .NET Framework, or
- ✓ *domains*, such business process modeling, service-oriented architecture, medical applications, etc.

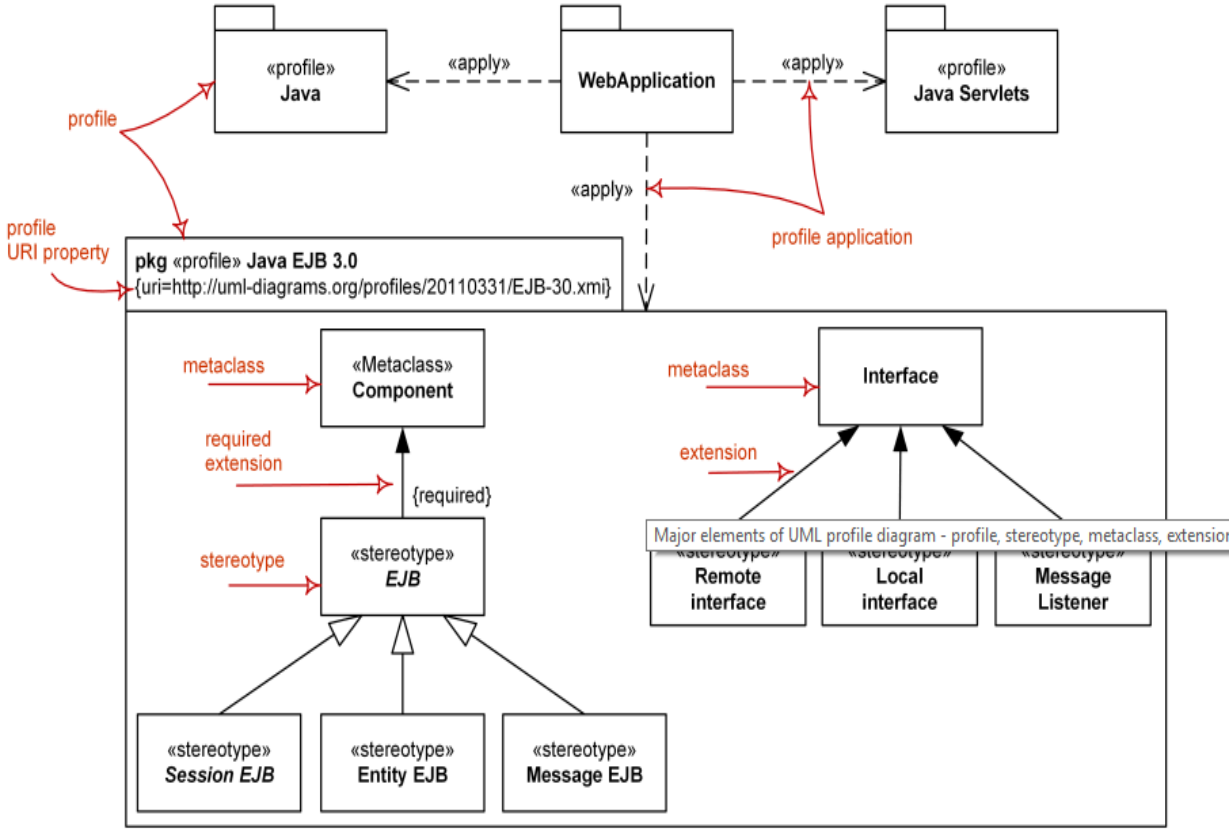
For **example,** semantics of standard UML meta model elements could be specialized in a profile. In a model with the profile "Java model," generalization of classes should be able to be restricted to single inheritance without having to explicitly assign a stereotype «Java class» to each and every class instance.

The profiles mechanism is not a first-class extension mechanism. It does not allow to modify existing meta models or to create a new meta model as MOF does. Profile only allows adaptation or customization of an existing meta model with constructs that are specific to a particular domain, platform, or method. It is not possible to take away any of the constraints that apply to a meta model, but it is possible to **add new constraints** that are specific to the profile.

Metamodel customizations are defined in a profile, which is then applied to a package. **Stereotypes** are specific metaclasses, **tagged values** are standard metaattributes, and **profiles** are specific kinds of packages.

Profiles can be dynamically applied to or retracted from a model. They can also be dynamically combined so that several profiles will be applied at the same time on the same model.

Graphical nodes and edges used on profile diagrams are: **profile**, **metaclass**, **stereotype**, **extension**, **reference**, **profile application**.



Behavioral Modeling: Use Case Diagram, Activity Diagrams, State Machine Diagrams, Sequence Diagram, Communication Diagram, Timing Diagram, Interaction Overview Diagram.

1) Use Case Diagram
2) Activity Diagrams
3) State Machine Diagrams
4) Sequence Diagram
5) Communication Diagram
6) Timing Diagram
7) Interaction Overview Diagram

[1]UseCaseDiagram:

- A use case diagram is a diagram that shows a set of use cases and actors and their relationships.

Contents

- Use case diagrams commonly contain
    - Use cases
    - Actors
    - Dependency, generalization, and association relationships
- Like all other diagrams, use case diagrams may contain notes and constraints.
- Use case diagrams may also contain packages
- Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses

- We apply use case diagrams to model the static use case view of a system. This view primarily supports the behavior of a system
- When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.
    - To model the context of a system
    - To model the requirements of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it.Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

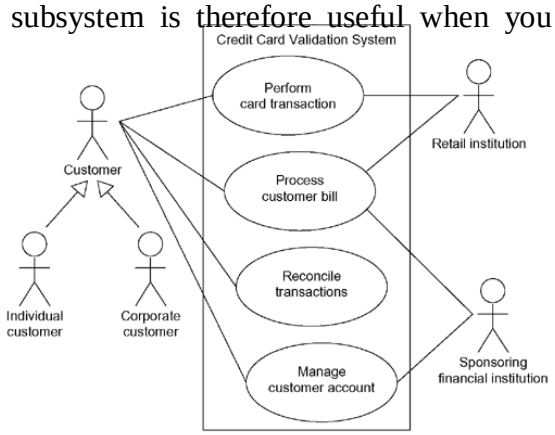
Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system.

Common Modeling Techniques

Modeling the Context of a System

Given a system—any system—some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that interact with the system constitute the system's context. This context defines the environment in which that system lives.

- In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system.
- To model the context of a system
    - Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
    - Organize actors that are similar to one another in a generalization/specialization hierarchy.
    - Where it aids understandability, provide a stereotype for each such actor.
    - Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.
- This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

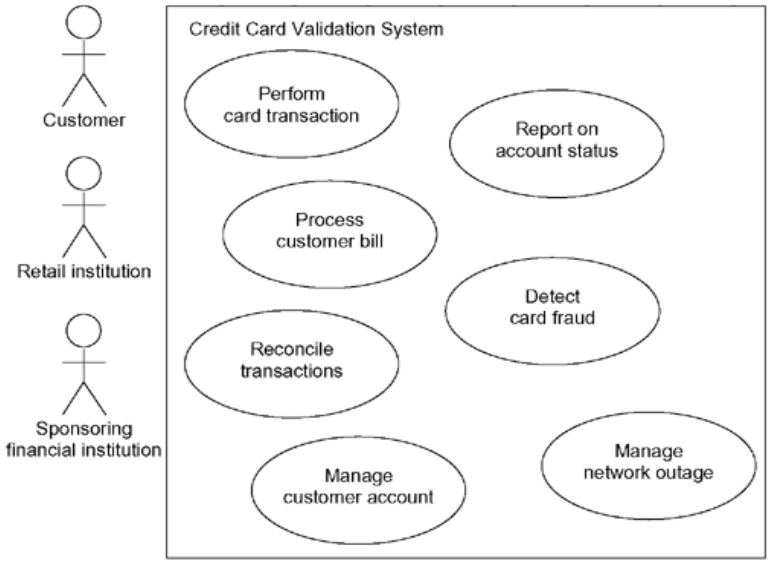


Modeling the Context of a System

Modeling the Requirements of a System

- A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do.

- Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between.
- Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.
- To model the requirements of a system,
  - Establish the context of the system by identifying the actors that surround it.
  - For each actor, consider the behavior that each expects or requires the system to provide.
  - Name these common behaviors as use cases.
  - Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
  - Model these use cases, actors, and their relationships in a use case diagram.
  - Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.
  - This same technique applies to modeling the requirements of a subsystem



Modeling the Requirements of a System

Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language.
- A use case diagram can be forward engineered to form tests for the element to which it applies.
- Each use case in a use case diagram specifies a flow of events and these flows specify how the element is expected to behave
- To forward engineer a use case diagram,
  - For each use case in the diagram, identify its flow of events and its exceptional flow of events.

- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.

- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.

- Use tools to run these tests each time you release the element to which the use case diagram applies.

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

- <u>To reverse engineer a use case diagram</u>

  - Identify each actor that interacts with the system.
  - For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.

  - Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.

  - Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.

Render these actors and use cases in a use case diagram, and establish their relationships.

[2]Activity Diagrams :

- An activity diagram shows the flow from activity to activity
- an activity diagram shows the flow of an object, how its role, state and attribute values changes
- activity diagrams is used to model the dynamic aspects of a system
- Activities result in some action (Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression)
- Figure 1: activity diagram
- an activity diagram is a collection of vertices and arcs
- Activity diagrams commonly contain Activity states and action states, Transitions, Objects
- activity diagrams may contain simple and composite states, branches, forks, and joins
- the initial state is represented as a solid ball and stop state as a solid ball inside a circle.

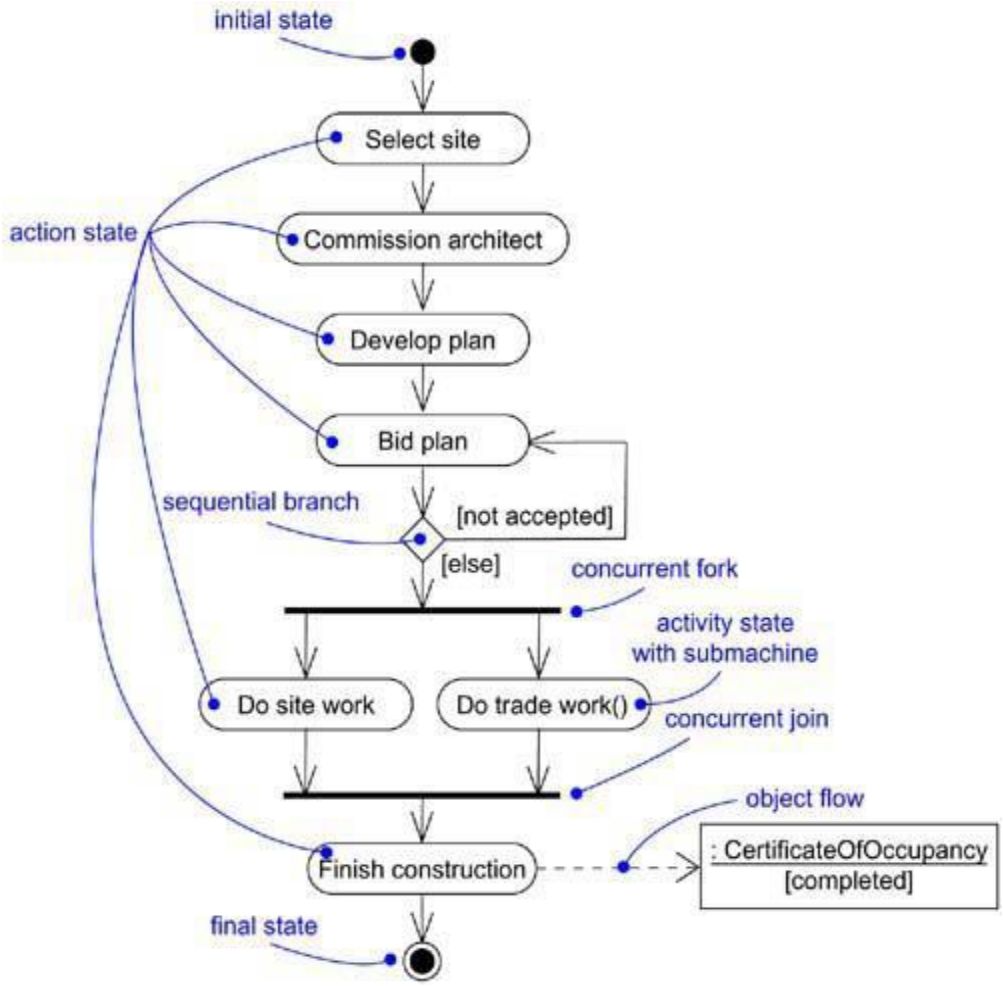


Figure 1: activity diagram

Action States

- The executable, atomic computations are called action states because they are states of the system, each representing the execution of an action
- Figure 2 Action States
- action states can't be decomposed
- action states are atomic, meaning that events may occur, but the work of the action state is not interrupted
- action state is considered to take insignificant execution time
- action states are special kinds of states in a state machine.

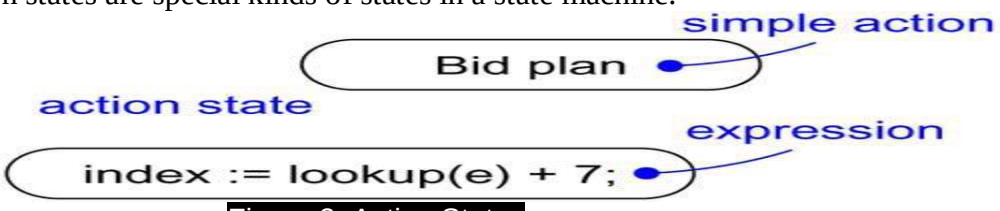


Figure 2: Action States

Activity States:

- activity states can be further decomposed
- activity states are not atomic, meaning that they may be interrupted
- they take some duration to complete
- Figure 3 Activity States
- are just special kinds of states in a state machine.

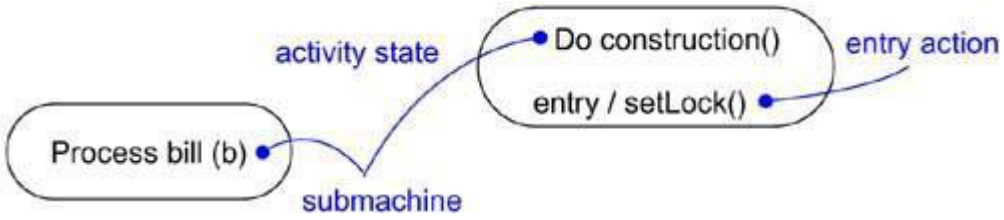


Figure 3: Activity States

Transitions:

- transitions shows the path from one action or activity state to the next action or activity state
- a transition is represented as a simple directed line.

Triggerless Transitions:

- Figure 4
- Triggerless Transitions are transitions where control passes immediately once the work of the source state is done.

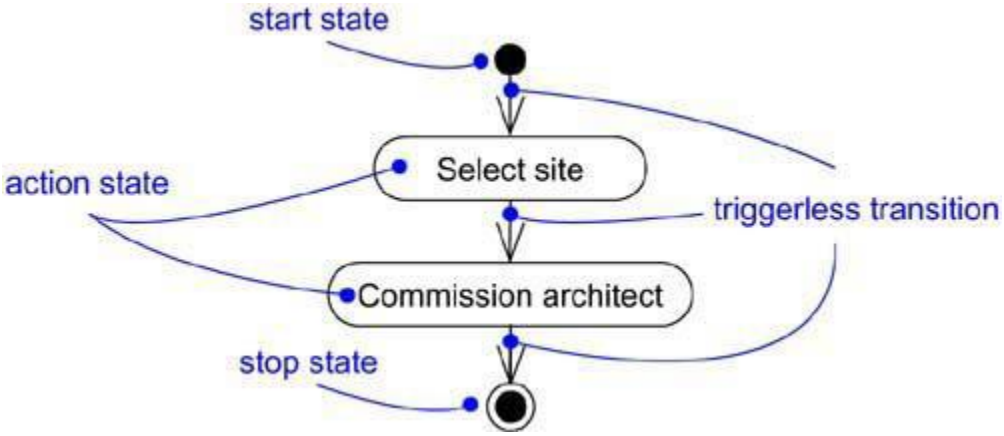


Figure 4: Triggerless Transitions

Branching:

- represent a branch as a diamond
- A branch may have one incoming transition and two or more outgoing ones each .
  - outgoing transition contains a guard expression, which is evaluated only once on entering the branch
- Figure 5: Branching.

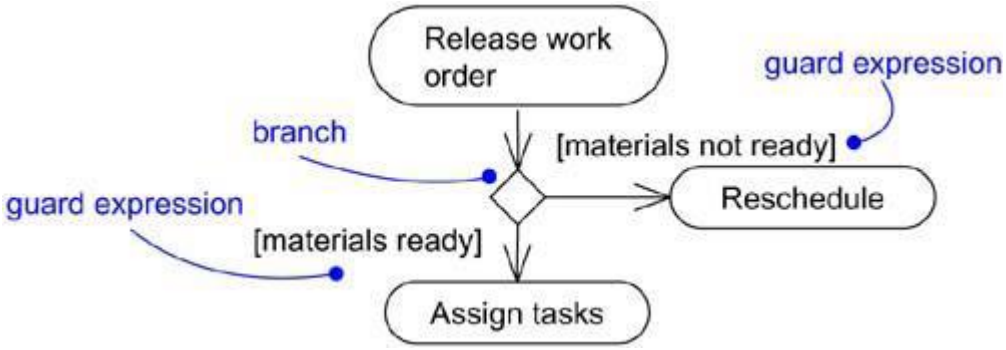


Figure 5: Branching

Forking and Joining:

- A fork may have one incoming transition and two or more outgoing transitions each of which represents an independent flow of control
- a fork represents the splitting of a single flow of control into two or more concurrent flows of control
- Below the fork, the activities associated with each of these paths continues in parallel
- A join may have two or more incoming transitions and one outgoing transition
- Above the join, the activities associated with each of these paths continues in parallel
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join
- the forking and joining of the parallel flows of control are specified by a synchronization bar
- A synchronization bar is rendered as a thick horizontal or vertical line

- Figure 6: Forking and Joining.

Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.

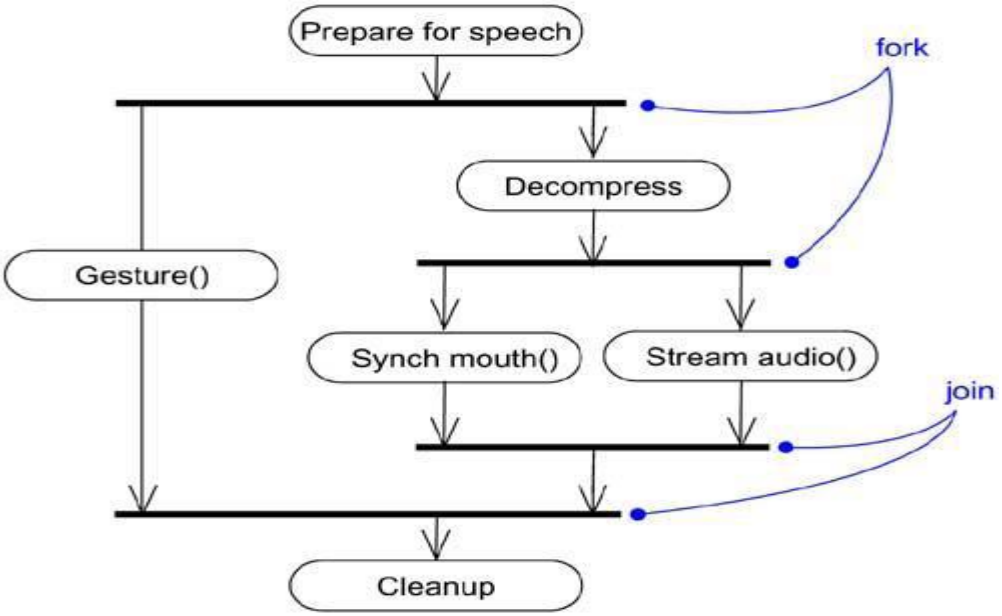


Figure 6: Forking and Joining

Swimlanes:

- swimlanes partitions activity diagrams into groups having activity states where each group represents the business organization responsible for those activities
- Figure 7:Swimlanes
- Each swimlane has a name unique within its diagram
- swimlane represents a high-level responsibility for part of the overall activity of an activity diagram.
- Each swimlane is implemented by one or more classes.

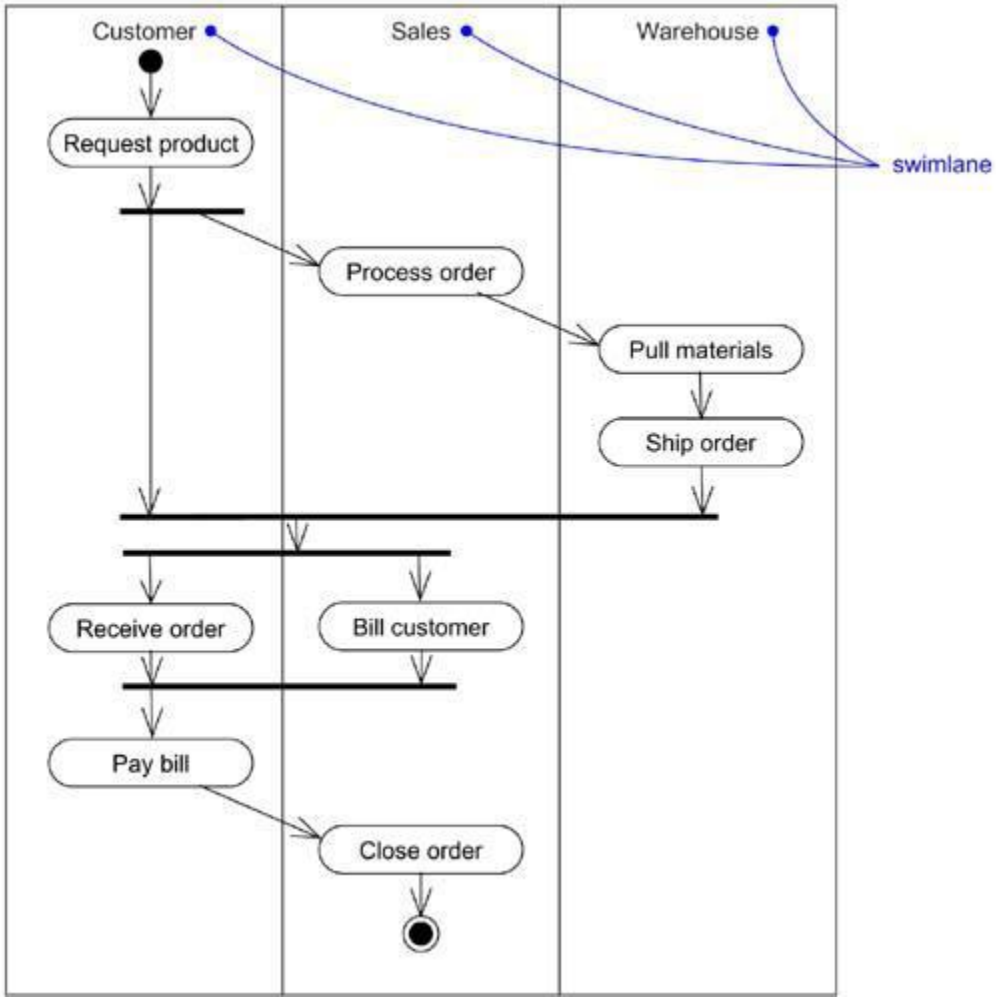


Figure 7: Swimlanes

Object Flow:

- object flow indicates the participation of an object in a flow of control, it is represented with the help of dependency relationships.

- Figure 8: Object Flow.

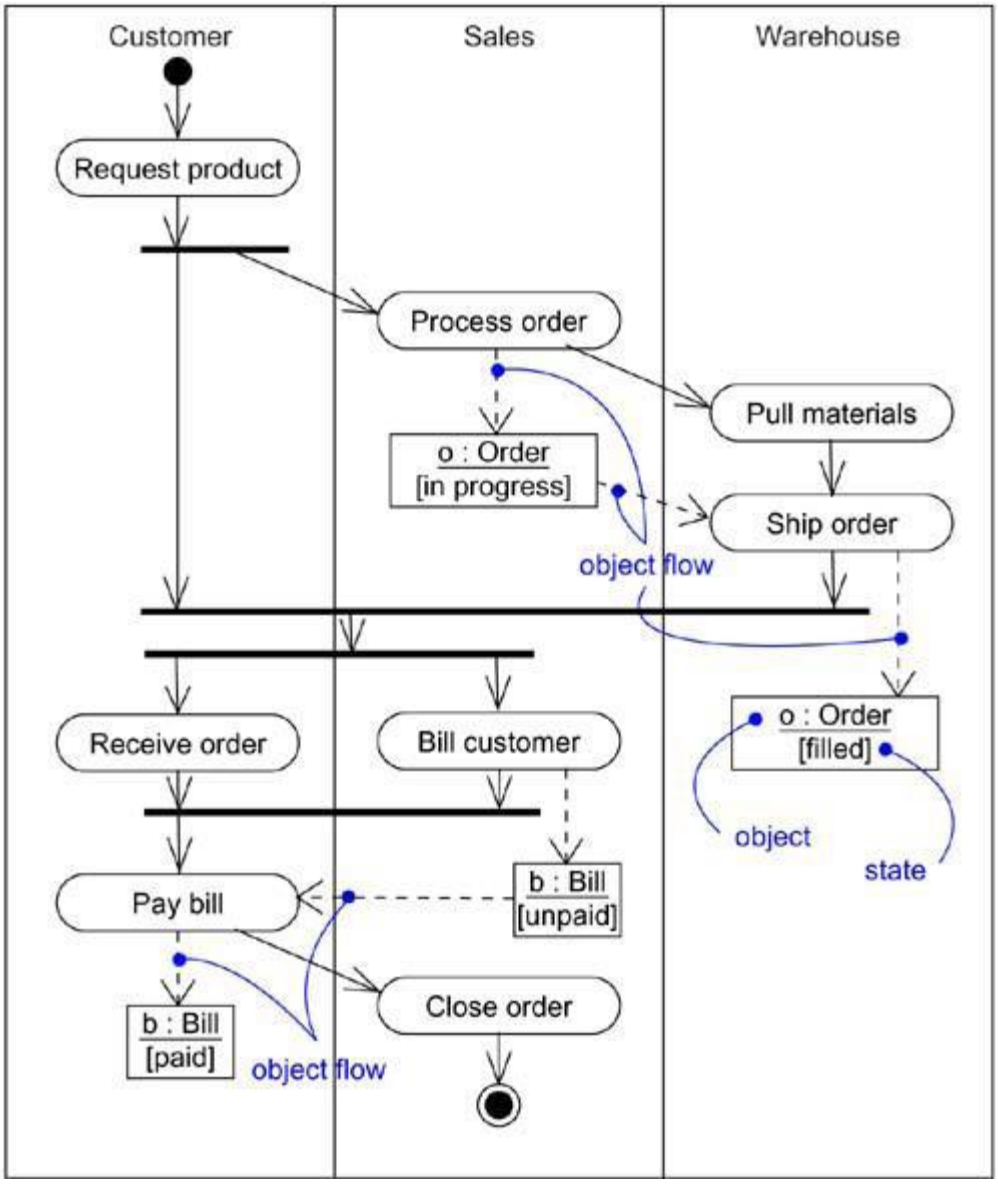


Figure 8: Object Flow

Activity diagrams are used to model dynamic aspects of a system and for this we need activity diagrams,

- To model a workflow
- To model an operation

Modeling a Workflow

To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.

- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure 9 shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.

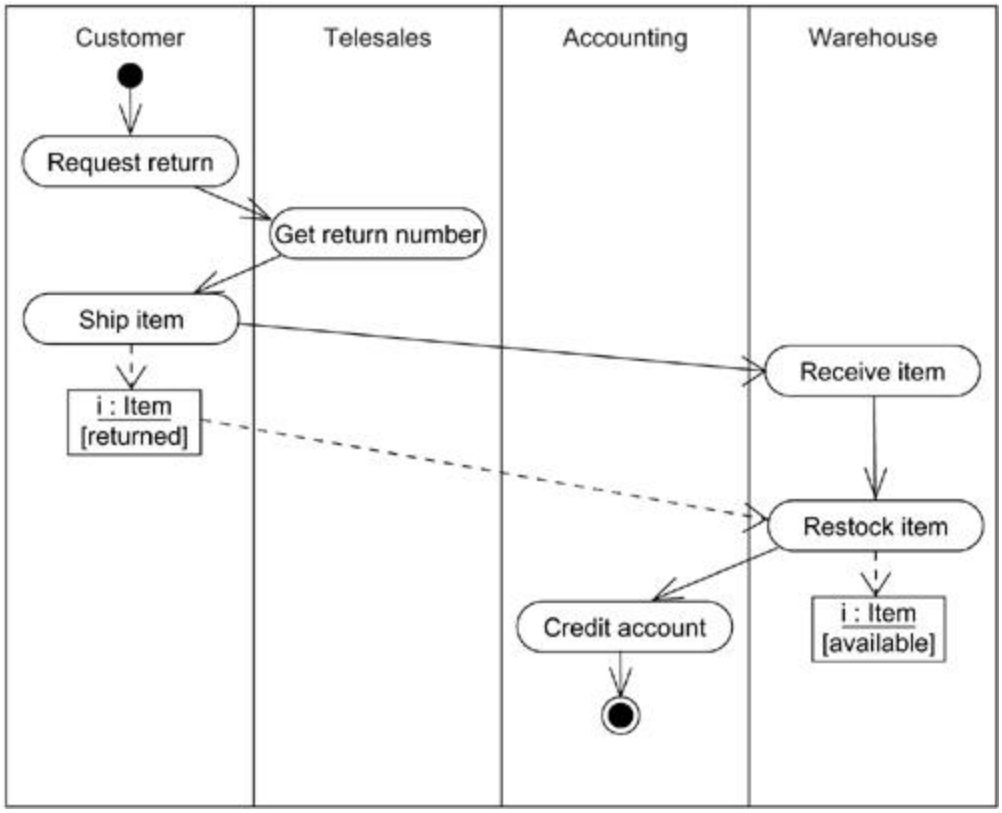


Figure 9: Modeling a Workflow

2)Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.

- Identify the preconditions at the operation's initial state and the post conditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Figure 10 shows an activity diagram that specifies the algorithm of the operation intersection b/w lines.

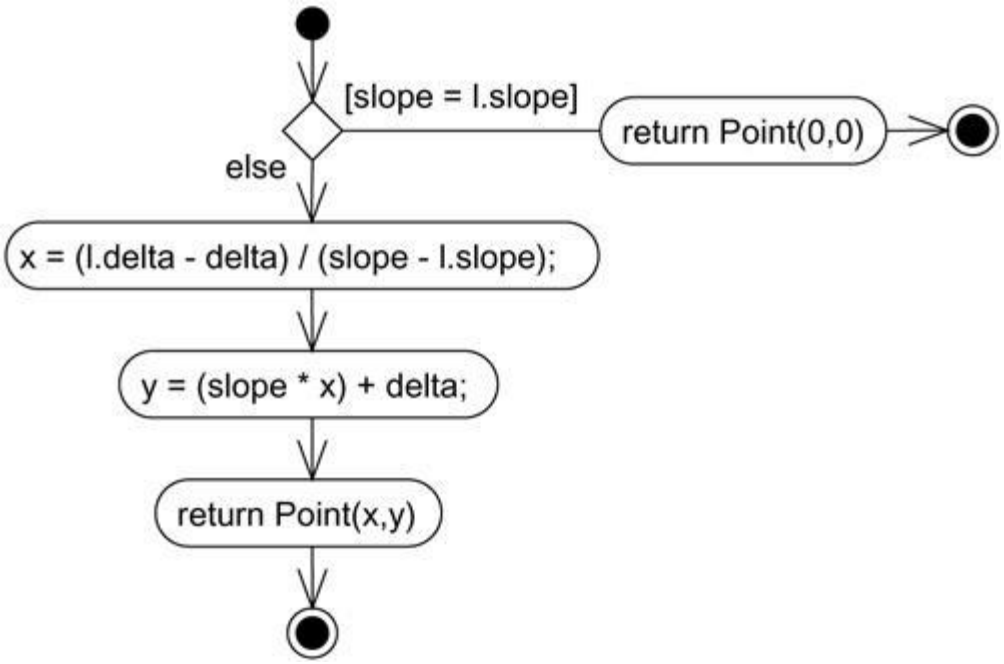


Figure 10: Modeling an Operation

[4]Sequence Diagrams:

Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.

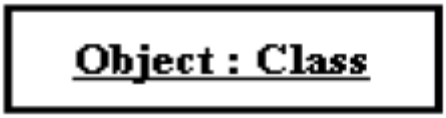
Notations:

These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis, in the order of increasing time from top to bottom.

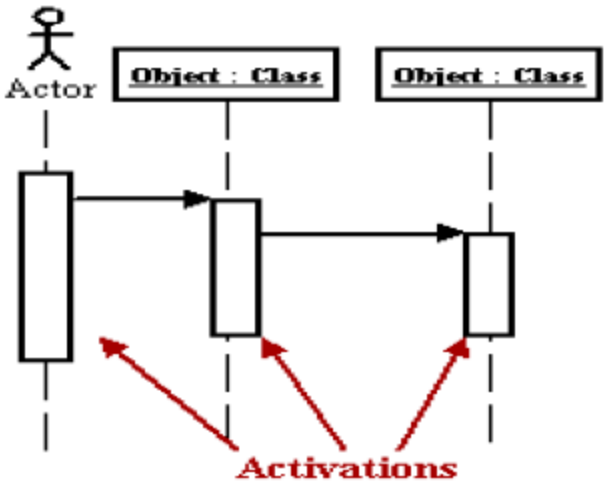
Sequence Diagrams: the basic elements:

Class roles:

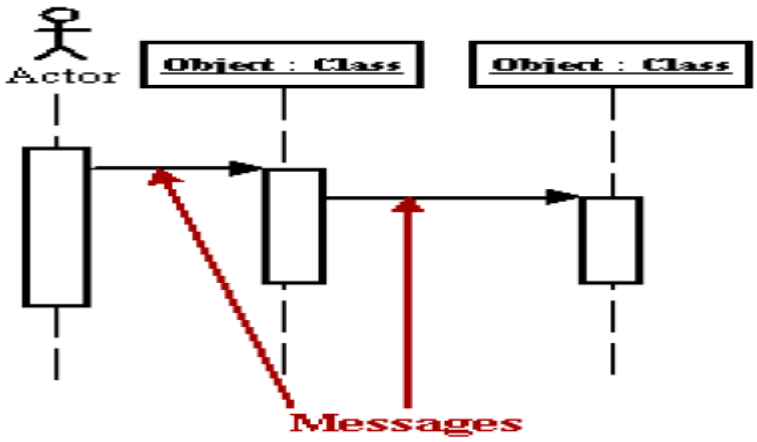
Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.



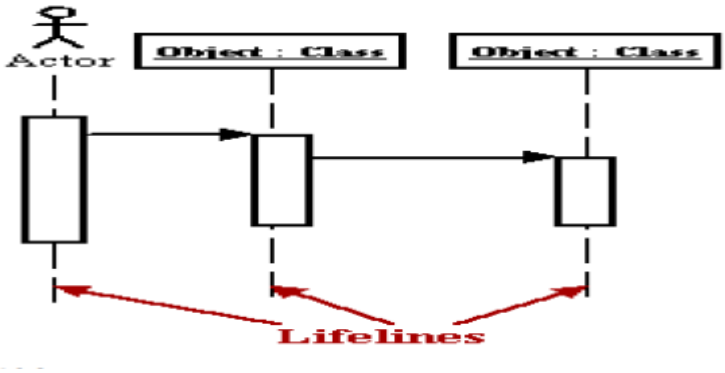
Activation:
Activation boxes represent the time an object needs to complete a task.



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.

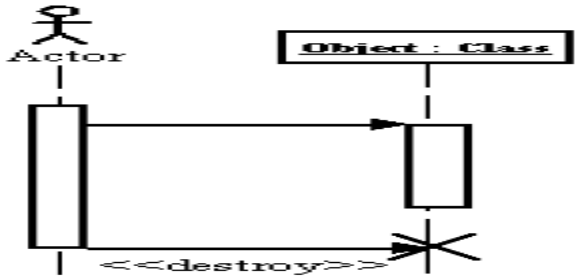


Lifelines:
Lifelines are vertical dashed lines that indicate the object's presence over time.



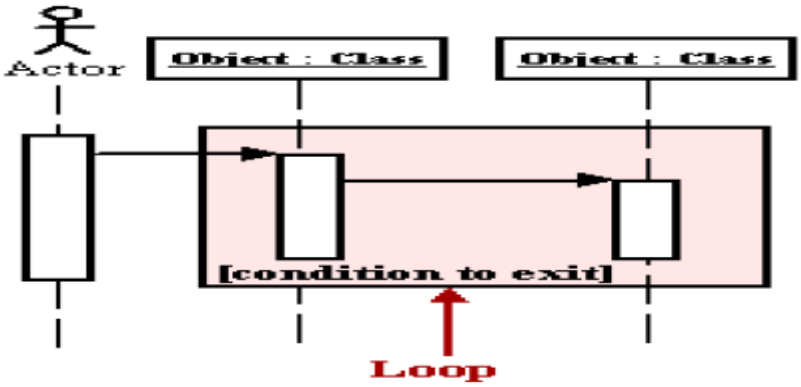
Destroying Objects:

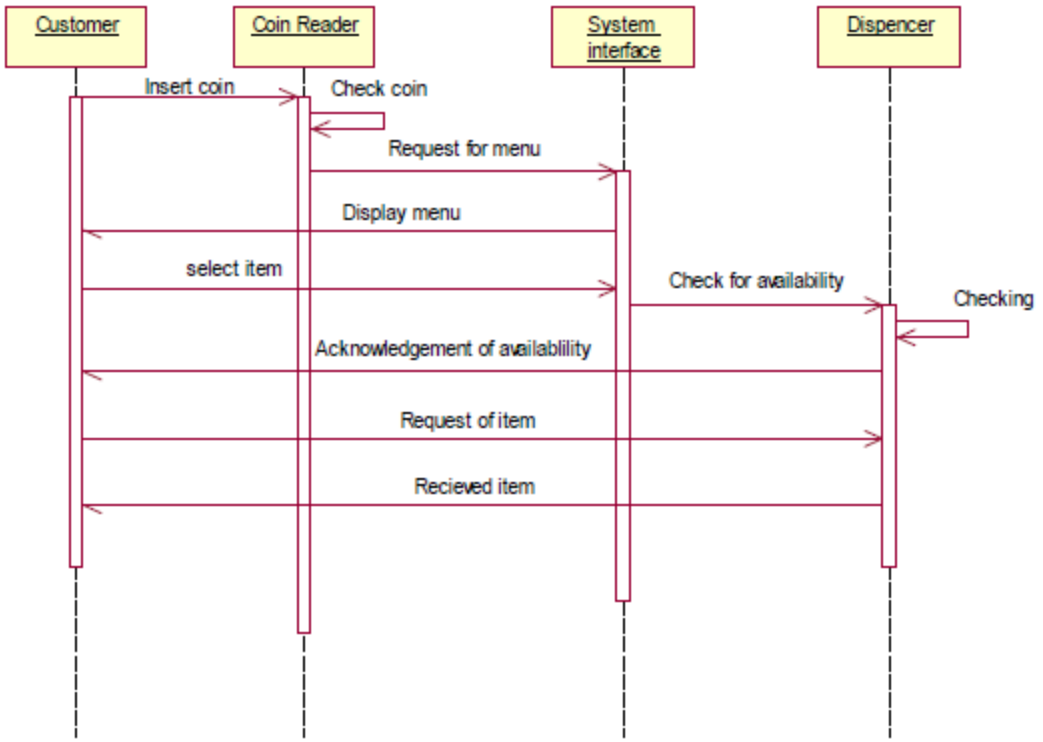
Objects can be terminated early using an arrow labeled "< < destroy > >" that points to an X.



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 [ ].

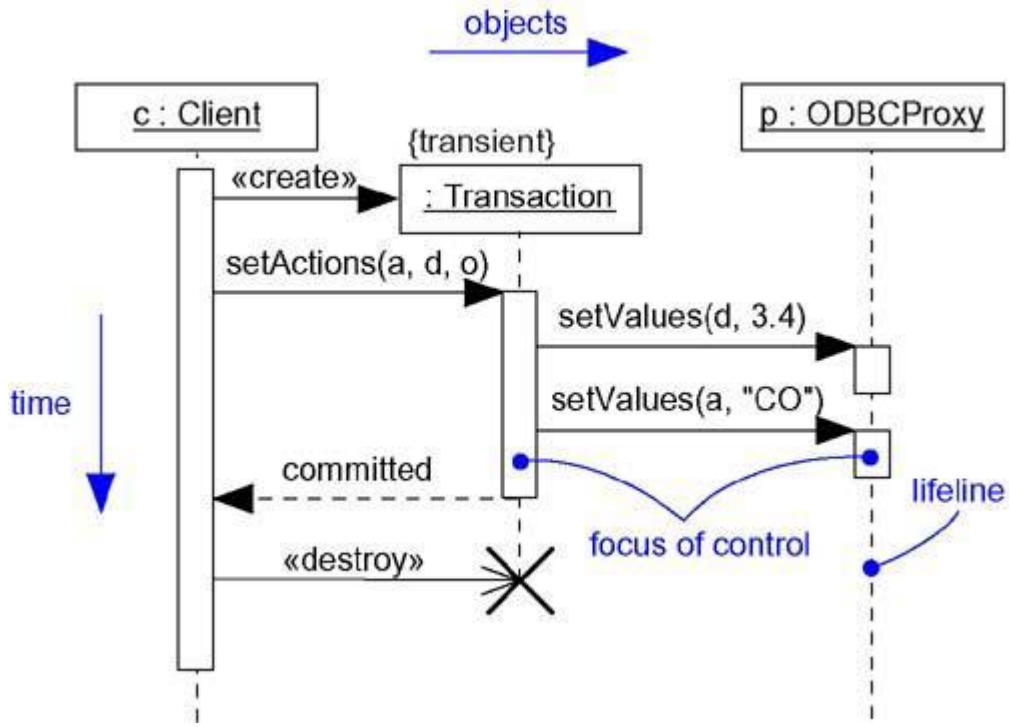




Vending Machine Sequence Diagram

Sequence Diagrams:

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages
- Graphically it is a table that shows objects arranged along the X axis and messages ordered in increasing time along the Y axis.
- Figure 2 Sequence Diagram.
- place the objects that participate in the interaction at the top of your diagram, across the X axis, object that initiates the interaction at the left, and increasingly more subordinate objects to the right.
- place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom.



[5]Communication Diagrams:

4.4 Communication Diagrams

Communication diagrams are the second type of Interaction diagrams. They also show the interaction between objects as they collaborate to realize scenarios within a use case. In addition, they also depict the relationships between objects (that pass messages among themselves) through the paths of communication.

A question may arise as to whether the modeller should depict interaction through sequence or through a communication diagram. The answer depends on what is considered crucial by the modeller. If depicting the interaction and the sequence of events over time is important, then the sequence diagram must be used. If depicting the interaction and the relationship between objects is crucial, then communication diagrams must be used.

4.4.1 Basic Notation for Communication Diagrams

The sequence of messages is indicated by explicitly numbering them. Consider Figure 4.17; there are three objects of the classes Computer, MusicServer and MediaPlayer. The actor initiates the use case by asking the object of the type Computer to execute its play() operation.

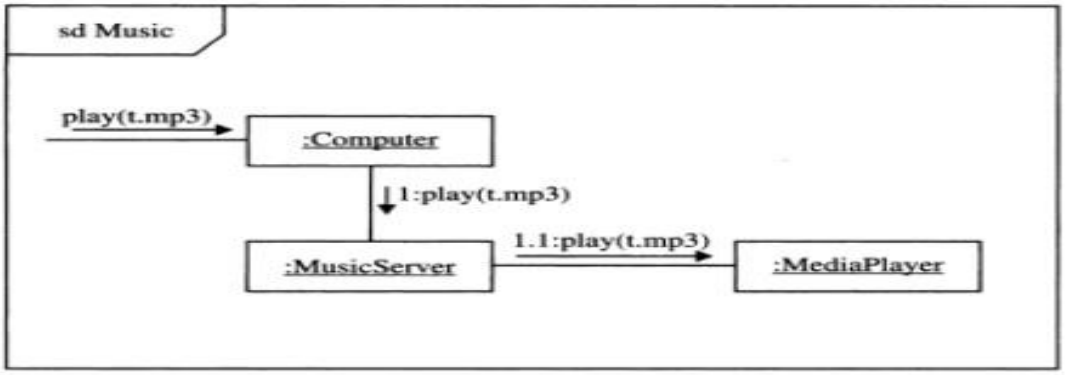


Figure 4.17 Communication diagram.

The object of the type Computer sends a message to the object of the type MusicServer, asking it to execute its play() operation. This message is given the number 1. The receipt of this message prompts the object of the type MusicServer to send a message to the object of the type MediaPlayer to execute its play() operation. Message 1.1 is a nested message. After message 1.1 has been executed, if the object of the class Computer had sent a message to some other object, it would have been numbered 2. If, after message 1.1 has been executed, the MusicServer had sent another message to object, of class MediaPlayer or to any other object, it would have been numbered 1.2. If the object of the class MediaPlayer sent a message to any object after the receipt of massage numbered 1.1, that particular message would have been numbered 1.1.1.

4.4.2 Labelling Messages in Communication Diagrams

The formal syntax for message labelling is as follows:

```
[predecessor] [guard condition] sequence expression
[returnvalue ':='] message-name '(' [argument list] ')'
```

Certain conventions are followed while labelling messages in communication diagrams.

1. Characters in single quotes like := and () are literal values.
2. Terms in [] are not compulsory.
3. Other terms are replaced by actual values.
4. Terms in braces {} are repeated zero or more times.

Predecessor: It is a list of the sequence number of messages that must occur before the current message is sent. This feature is useful if branches are present. If no branch is present, then predecessor can be safely omitted. Note that it is enclosed in []. By default, the message with the immediately preceding sequence number is the predecessor.

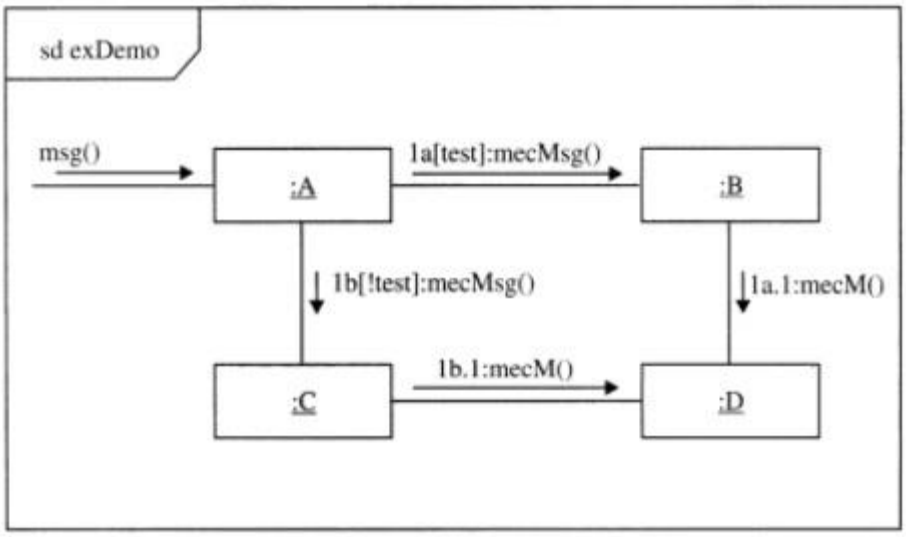


Figure 4.20 Depiction of mutually exclusive conditional messages.

4.4.6 Iteration for a Single Message

On receipt of message msg(), the object of class A sends a message msg2() to the object of class B 'N' times. This is indicated by putting an asterisk after the message number. Suppose the modeller does not want to show the number of times the message is sent, then he can omit the brackets and the entire text within them. See Figure 4.21.

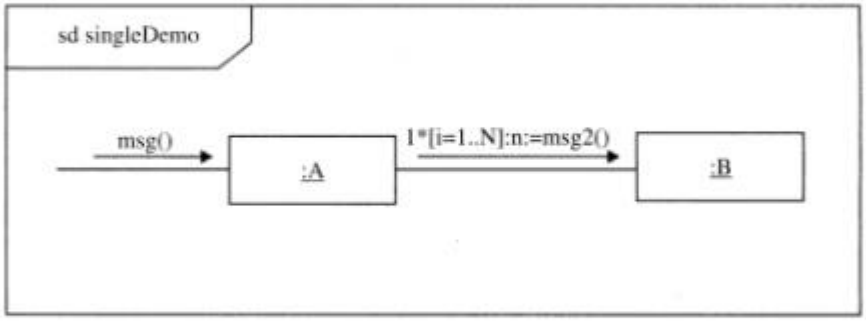


Figure 4.21 Iteration for a single message.

4.4.7 Iteration over a Collection

Suppose there is a collection of objects called B. On receipt of message msg() the object of the class A sends a message msg2() to every object in the collection. It is indicated by showing an asterisk after the message number as well as at the end of the link on the collection side, as shown in Figure 4.22. The symbol double-box indicates the multi-object.

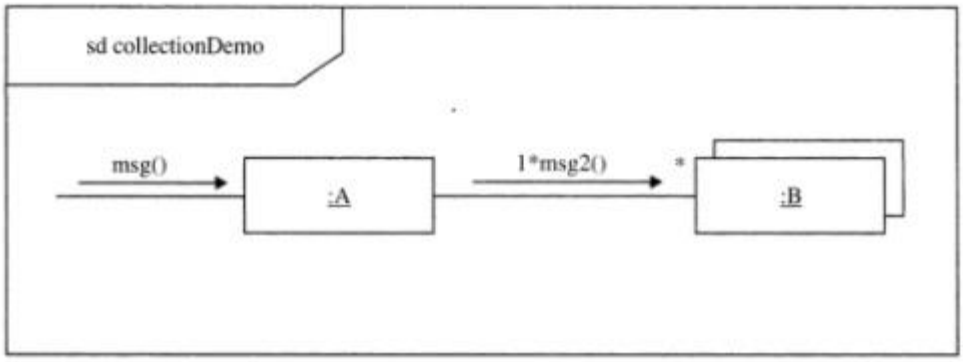


Figure 4.22 Iteration over a collection of objects.

4.4.8 Invoking Static Messages

Suppose an object of the class A wants to invoke the static message msStatic() of class B. It is shown in Figure 4.23.

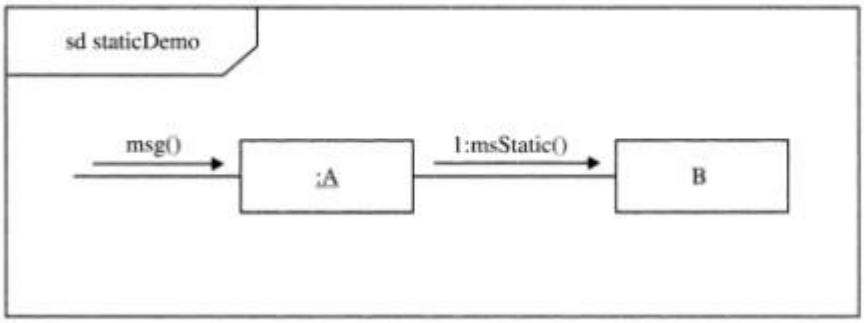


Figure 4.23 Invoking static messages.

Summary

- Interaction diagrams are used to illustrate the dynamic behaviour of a community of objects, which collaborate by passing messages in order to perform a certain task. There are two types of interaction diagrams, namely the sequence and communication diagrams.
- Sequence diagrams are used to depict the sequence of messages that are passed between objects. The diagram is read from top to down. Objects are listed on the top.
- Sequence diagrams can be broken up into chunks called fragments. The alt and opt combination fragments depict conditional behaviour, loop indicates behaviour in which a sequence of instructions is executed many times, break is analogous to break construct in Java, par depicts the concurrent execution of operations, and ref is used to reference another sequence diagram.

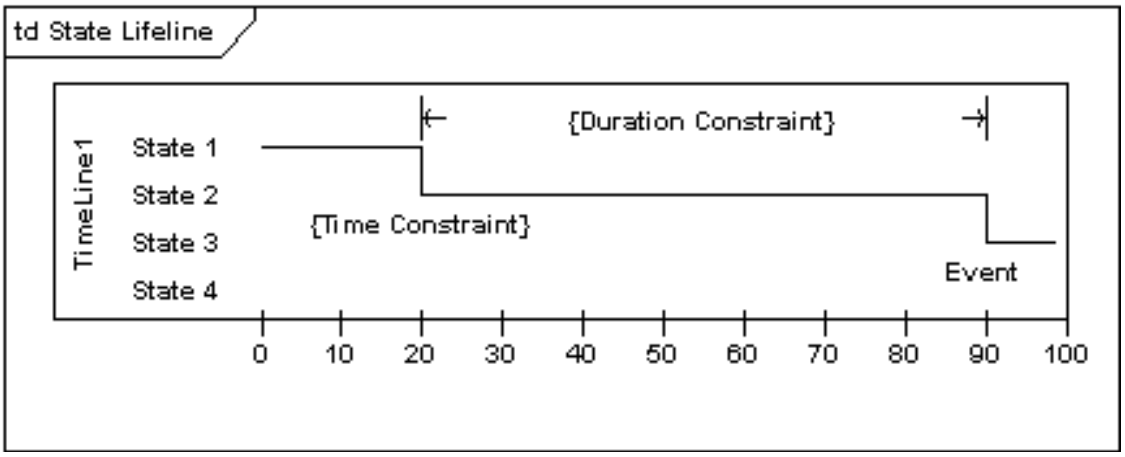
- A communication diagram is another form of interaction diagrams. In this form all messages (other than the first message) are explicitly numbered. An arrow is shown to depict the source and target object of the message.
- Concepts like creation of an object, mutually exclusive conditional messages, looping of a single message, invoking a static message, etc. can be displayed by using communication diagrams.
- There are no lifelines or activation boxes for communication diagrams. The sequence of messages can be found out by reading the numbers of the messages.

[6] Timing Diagrams:

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

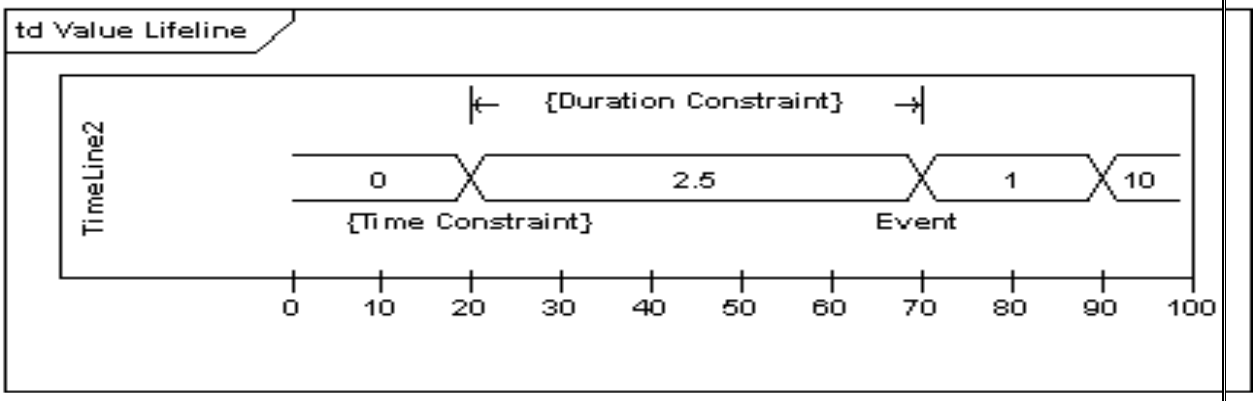
State Lifeline:

- A state lifeline shows the change of state of an item over time.
- The X-axis displays elapsed time in whatever units are chosen, while the Y-axis is labeled with a given list of states.
- A state lifeline is shown below.



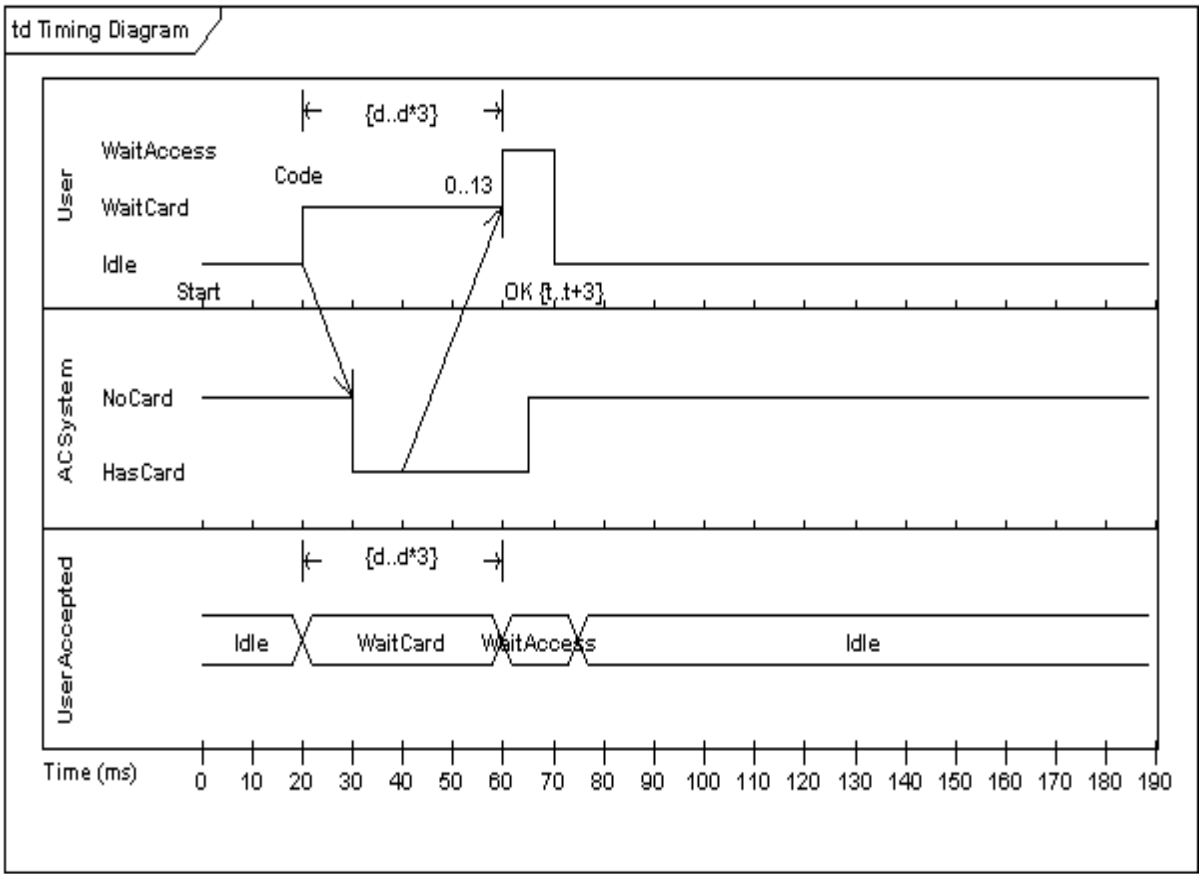
Value Lifeline:

- A value lifeline shows the change of value of an item over time.
- The X-axis displays elapsed time in whatever units are chosen, the same as for the state lifeline.
- The value is shown between the pair of horizontal lines which crosses over at each change in value.
- A value lifeline is shown below.



Putting it all Together:

- State and value Lifelines can be stacked one on top of another in any combination. They must have the same X-axis.
- Messages can be passed from one lifeline to another.
- Each state or value transition can have a defined event, a time constraint which indicates when an event must occur, and a duration constraint which indicates how long a state or value must be in effect for.
- Once these have all been applied, a timing diagram may look like the following.



Pattern:

- Patterns help you to visualize, specify, construct, and document the artifacts of a software intensive system.
- You can forward engineer a system by selecting an appropriate set of patterns and applying them to the abstractions specific to your domain.
- You can also reverse engineer a system by discovering the patterns it embodies, although that's hardly a perfect process. Even better, when you deliver a system, you can specify the patterns it embodies so that when someone later tries to reuse or adapt that system, its patterns will be clearly manifest.

Framework:

- A framework is an architectural pattern that provides an extensible template for applications within a domain.
- For **example,** one common architectural pattern you'll encounter in real time systems is a cyclic executive, which divides time into frames and sub frames, during which Processing takes place under strict deadlines.
- Choosing this pattern versus its alternatives (an evendriven architecture) colors your entire system. Because this pattern (and its alternative) is so common, it makes sense to name it as a framework.

[7] Interaction overview diagram:

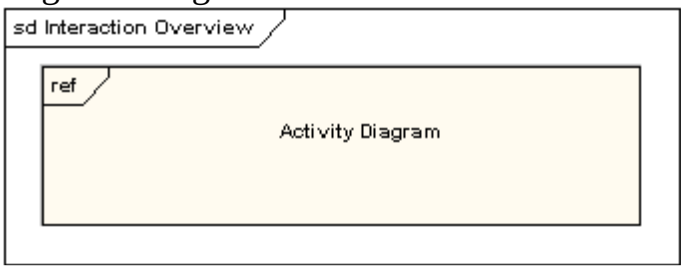
- An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams.
- Interaction diagrams can include sequence, communication, interaction overview and timing diagrams.
- Most of the notation for interaction overview diagrams is the same for activity diagrams.
- For **example,** initial, final, decision, merge, fork and join nodes are all the same. However,

Interaction overview diagrams introduce **two new elements**: interaction occurrences and interaction elements.

a) Interaction occurrences
b) Interaction elements.

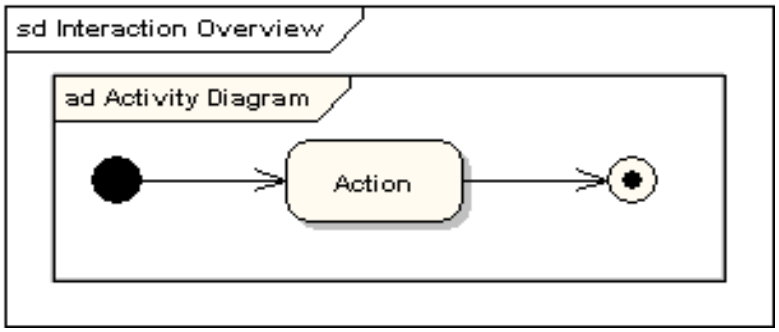
a) Interaction Occurrence:

- Interaction occurrences are references to existing interaction diagrams.
- An interaction occurrence is shown as a reference frame; that is, a frame with "ref" in the top-left corner.
- The name of the diagram being referenced is shown in the center of the frame.



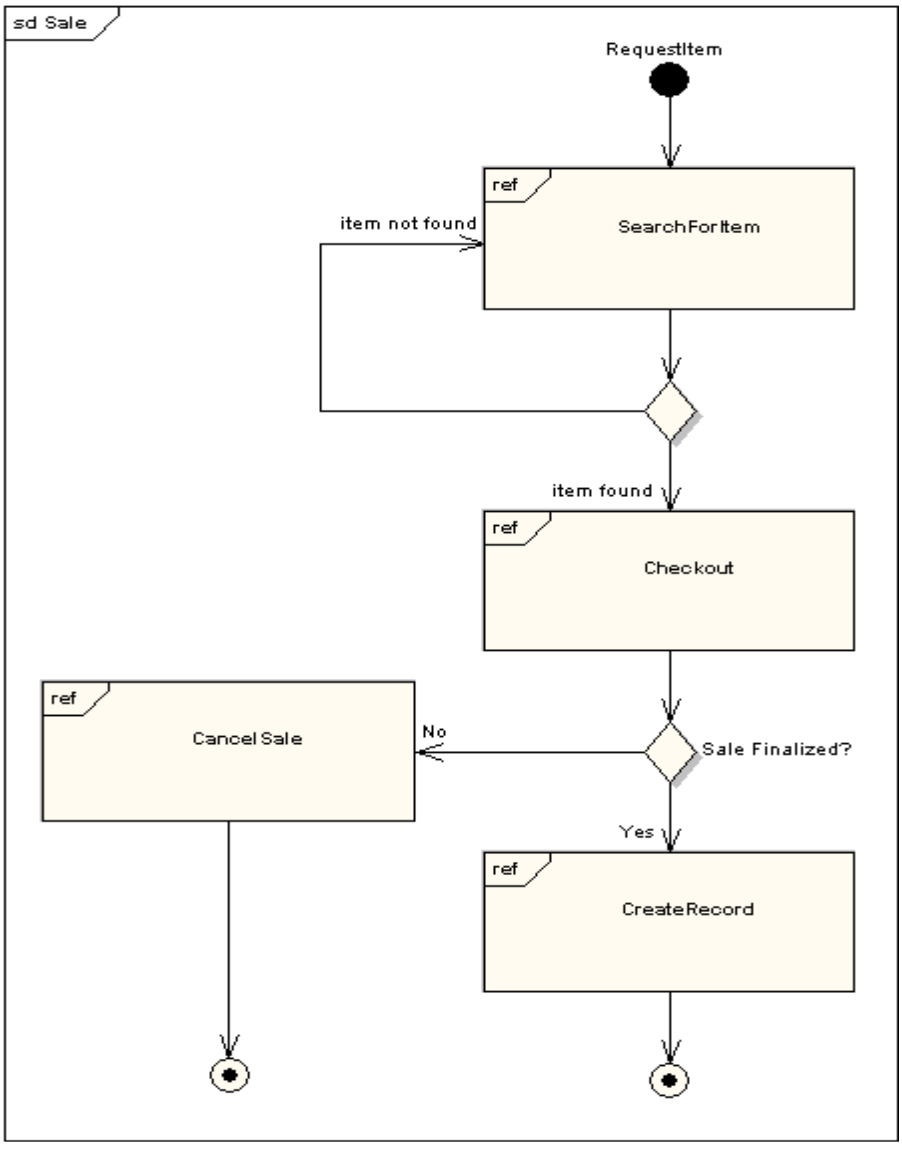
Interaction Element:

- Interaction elements are similar to interaction occurrences, in that they display a representation of existing interaction diagrams within a rectangular frame.
- They differ in that they display the contents of the references diagram inline.



Putting it all Together:

- All the same controls from activity diagrams (fork, join, merge, etc.) can be used on interaction overview diagrams to put the control logic around the lower level diagrams.
- The following example depicts a sample sale process, with sub-processes abstracted within interaction occurrences.



Interaction Diagrams:

- Interaction diagrams depict interactions of objects and their relationships.
- They also include the messages passed between them.

There are two types of interaction diagrams:

- Communication diagram (Collaboration Diagram)
- Sequence Diagram

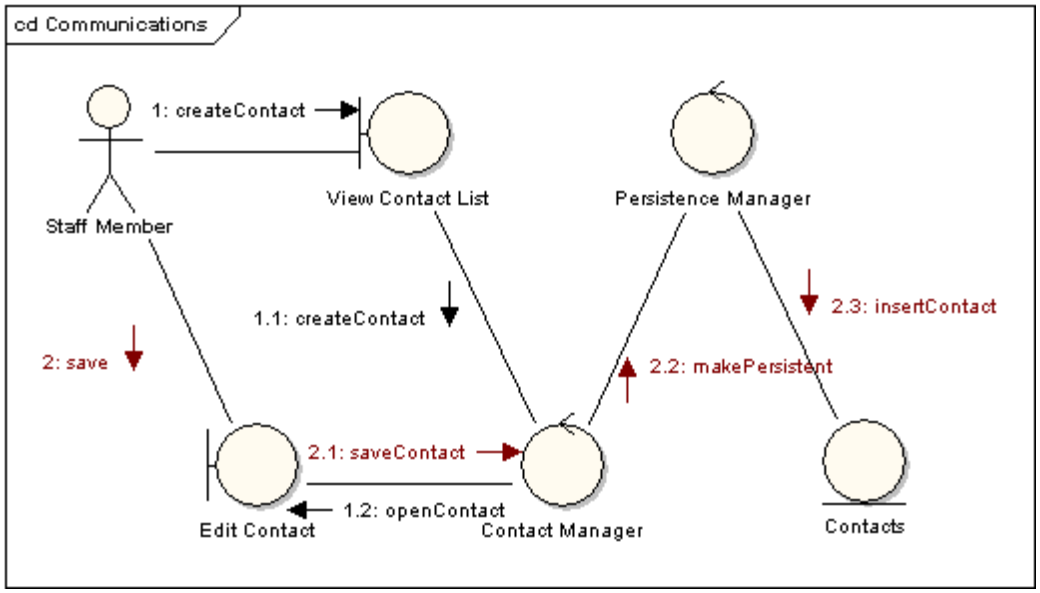
Interaction diagrams are used for modeling:

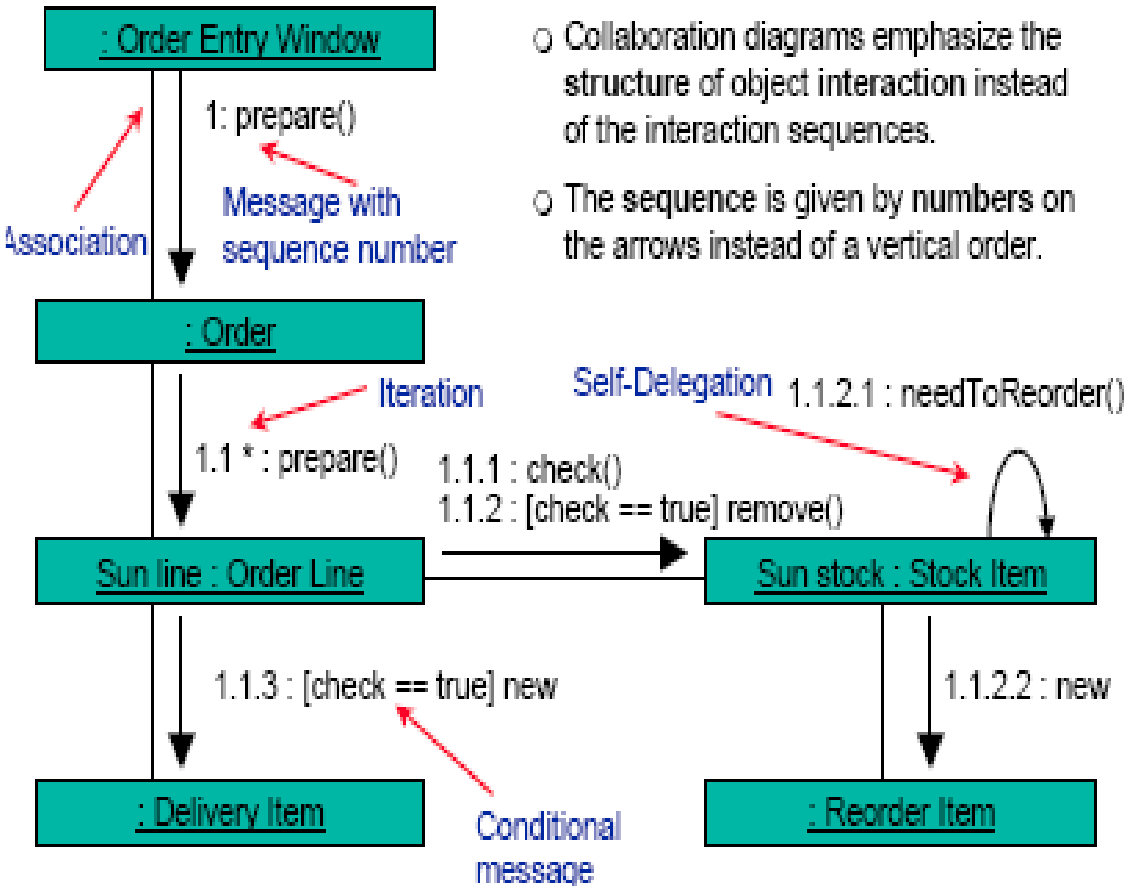
- The control flow by **time ordering** using **sequence diagrams.**
- The control flow of **organization** using **collaboration diagrams.**

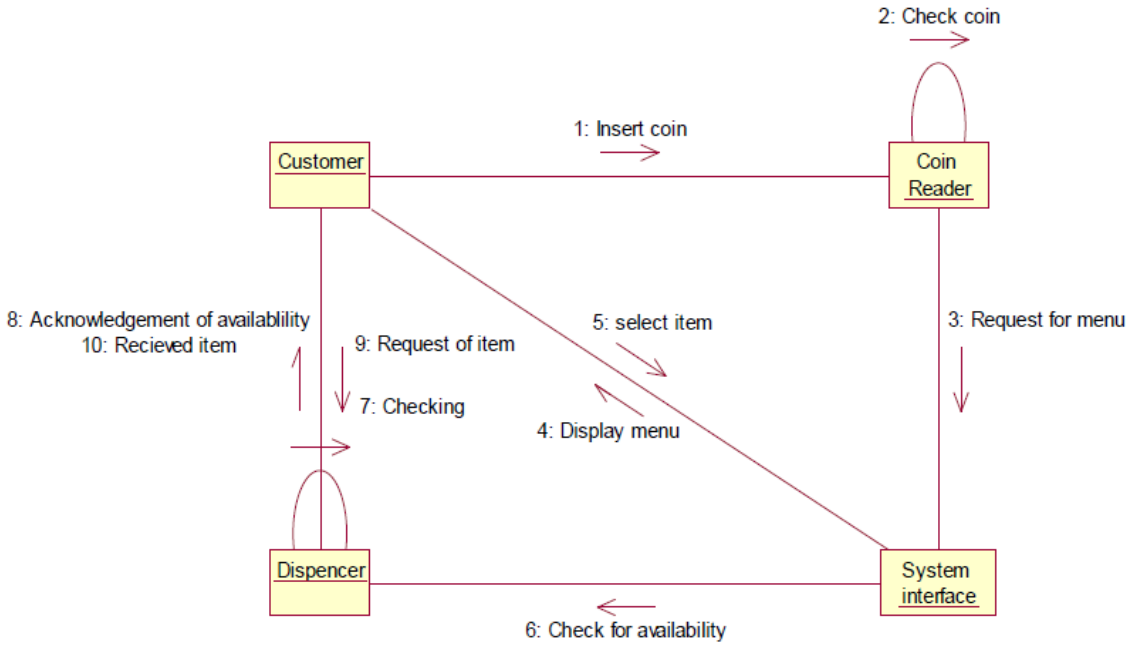
Communication Diagram:

- A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but
- its primary focus is on object relationships.
- In communication diagrams, objects are shown with association connectors between them.
- Messages are added to the associations and show as short arrows pointing in the direction of the message flow.
- The sequence of messages is shown through a numbering scheme.

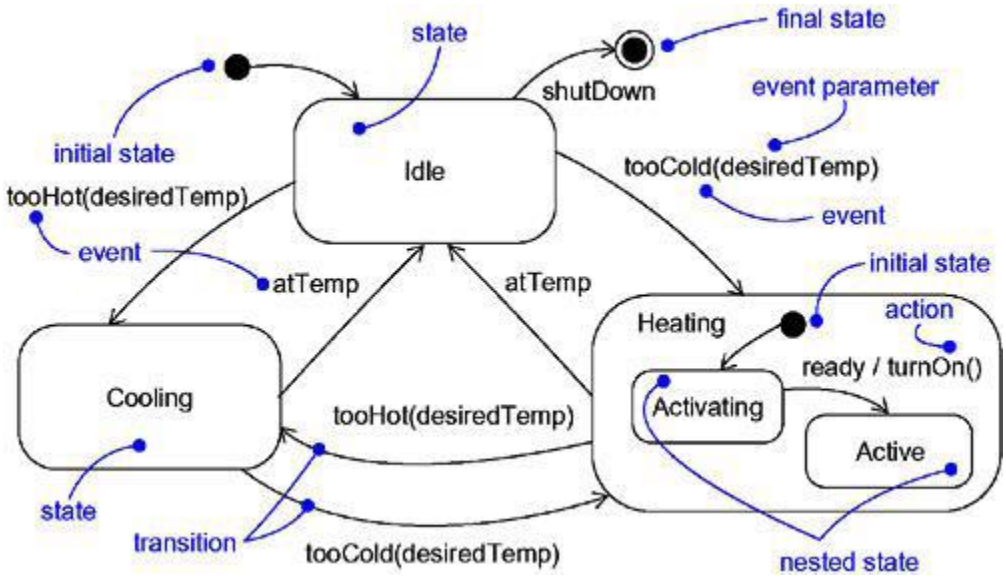
The following diagrams are the examples of communication diagram.




: Order Entry Window
1: prepare()
Association
Message with sequence number
Collaboration diagrams emphasize the structure of object interaction instead of the interaction sequences.
The sequence is given by numbers on the arrows instead of a vertical order.
: Order
Iteration
Self-Delegation
1.1.2.1 : needToReorder()
1.1 * : prepare()
1.1.1 : check()
1.1.2 : [check == true] remove()
Sun line : Order Line
Sun stock : Stock Item
1.1.3 : [check == true] new
1.1.2.2 : new
: Delivery Item
Conditional message
: Reorder Item


2: Check coin
1: Insert coin
Customer
Coin Reader
8: Acknowledgement of availablility
10: Recieved item
9: Request of item
5: select item
3: Request for menu
7: Checking
4: Display menu
Dispencer
System interface
6: Check for availability

State Machine Diagrams:



state machine:

- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events.
- Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.
- Figure 1 shows State Machines
- state machine are used to specific the behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past.
- state machines are used to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle, Activating, Active, and ShuttingDown. a state name must be unique within its enclosing state.
- A state has five parts:
  - ***Name, Entry/exit actions, Internal transitions*** – Transitions that are handled without causing a change in state,
  - ***Substates*** – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates,
  - ***Deferred events*** – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state
- Figure 2 shows States initial state indicates the default starting place for the state machine or substate and is represented as a filled black circle.

- final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle
- Initial and final states are pseudo-states

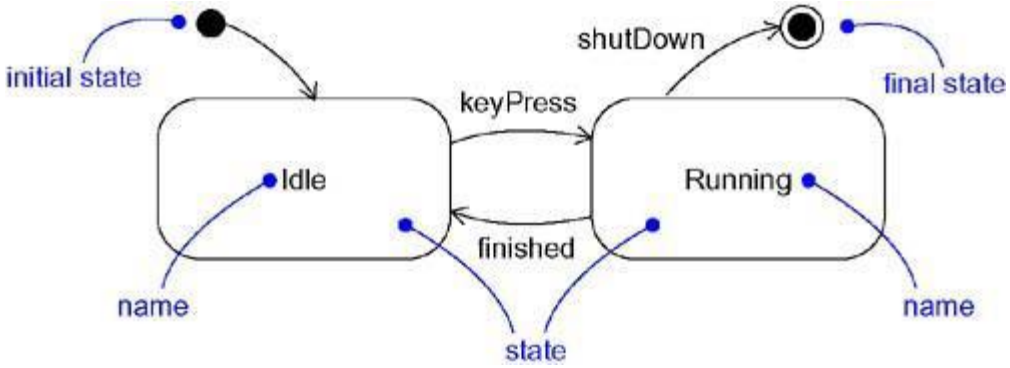


Figure 2: States

Transitions

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.

A transition has five parts:

1. *Source state* – The state affected by the transition,
2. *Event trigger* – a stimulus that can trigger a source state to fire on satisfying guard condition,
3. *Guard condition* – Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger,
4. *Action* – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object,
5. *Target state* – The state that is active after the completion of the transition.

Figure 3 shows transitions

- A transition may have multiple sources as well as multiple targets
- A *self-transition*is a transition whose source and target states are the same

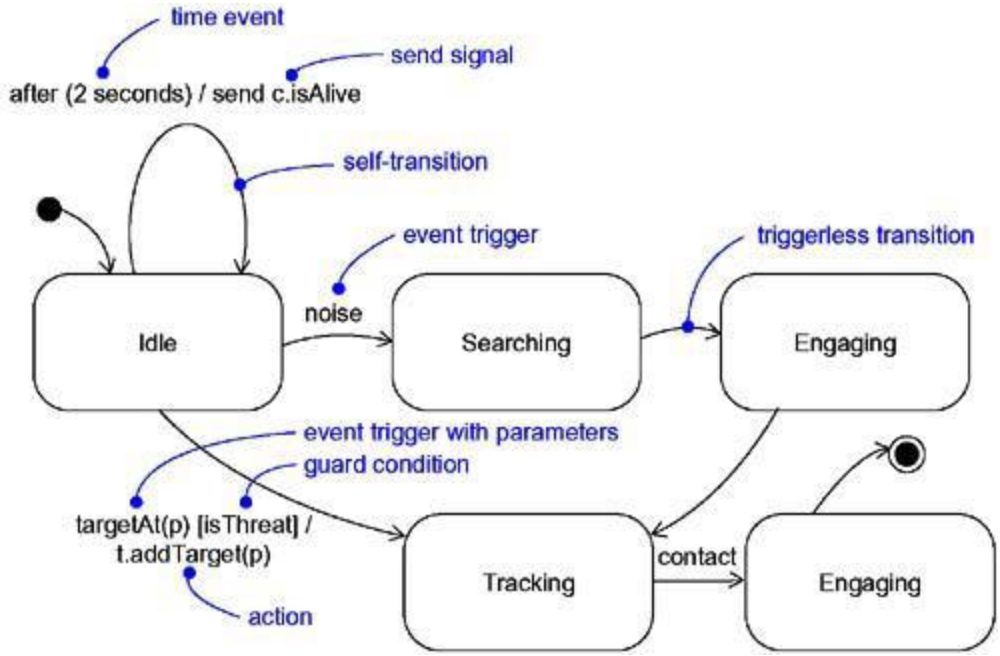


Figure 3:Transitions

Event Trigger

- An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
- events may include signals, calls, the passing of time,or a change in state.
- An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- An event trigger may be polymorphic

Guard condition

- a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event
- A guard condition is evaluated only after the trigger event for its transition occurs
- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

Action

- An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.
- Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object
- *An activity* may be interrupted by other events.

Advanced States and Transitions

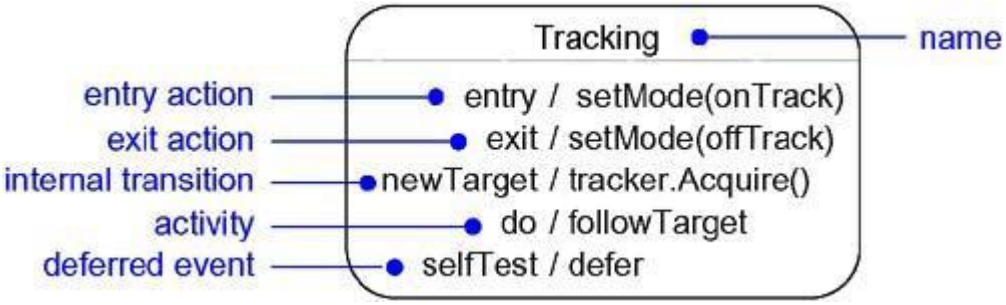


Figure 4: Advanced States and Transitions

Entry and Exit Actions

Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event 'entry' with an appropriate action

Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event 'exit', together with an appropriate action

Internal Transitions

Internal Transitions are events that should be handled internally without leaving the state.

Internal transitions may have events with parameters and guard conditions.

Activities

Activities make use of object's idle time when inside a state. 'do' transition is used to specify the work that's to be done inside a state after the entry action is dispatched.

Deferred:

Events

A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action 'defer'.

Substates

- A substate is a state that's nested inside another one.
- A state that has substates is called a composite state.
- A composite state may contain either *concurrent (orthogonal) or sequential (disjoint) substates*.
- Substates may be nested to any level

Sequential Substates

- Sequential Substates are those substates in which an event common to the composite states can easily be excercised by each states inside it at any time

- sequential substates partition the state space of the composite state into disjoint states

Figure 5: shows Sequential Substates

A nested sequential state machine may have at most one initial state and one final state

History States

- A history state allows composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.
- a shallow history state is represented as a small circle containing the symbol H

Figure 6: shows History State

The first time entry to a composite state doesn't have any history and the process for collecting history is as shown in the figure: 6

the symbol H designates a *shallow history*, which remembers only the history of the immediate nested state machine.

the symbol H* designates *deep history*, which remembers down to the innermost nested state at any depth.

When only one level of nesting, shallow and deep history states are semantically equivalent.

Concurrent Substates

concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object

Figure 7: shows Concurrent Substates

Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow

A nested concurrent state machine does not have an initial, final, or history state

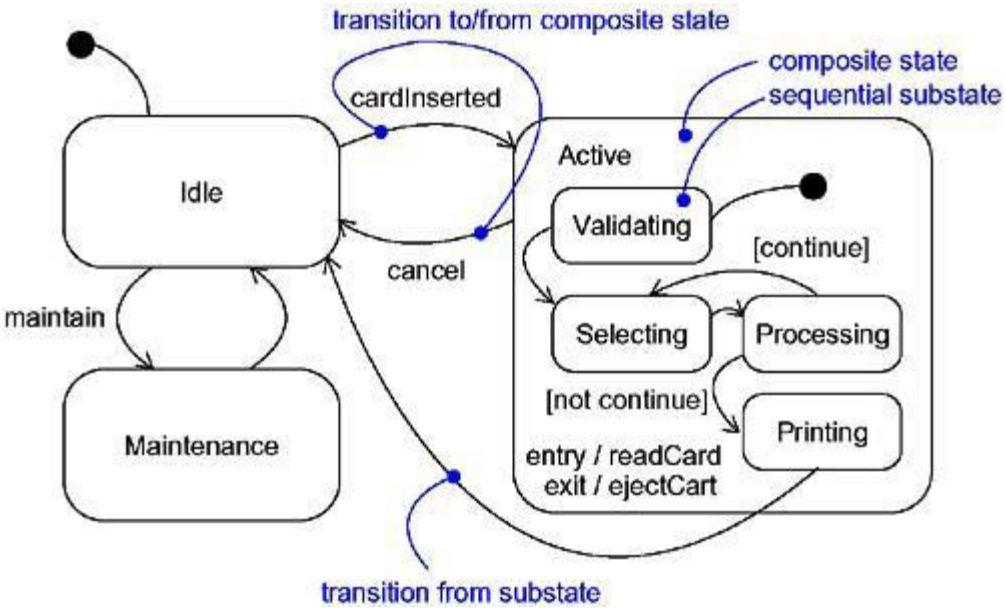


Figure 5: Sequential Substates

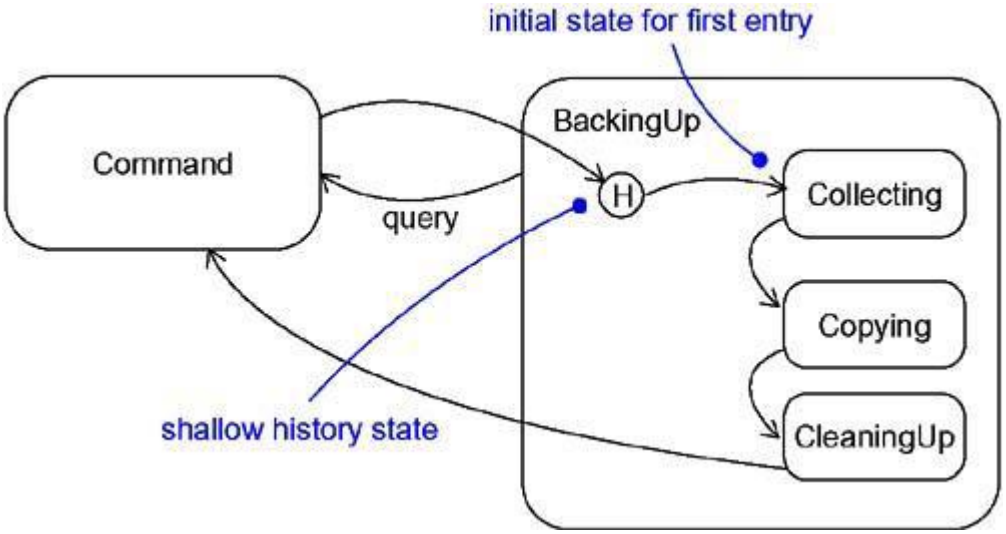


Figure 6: History State

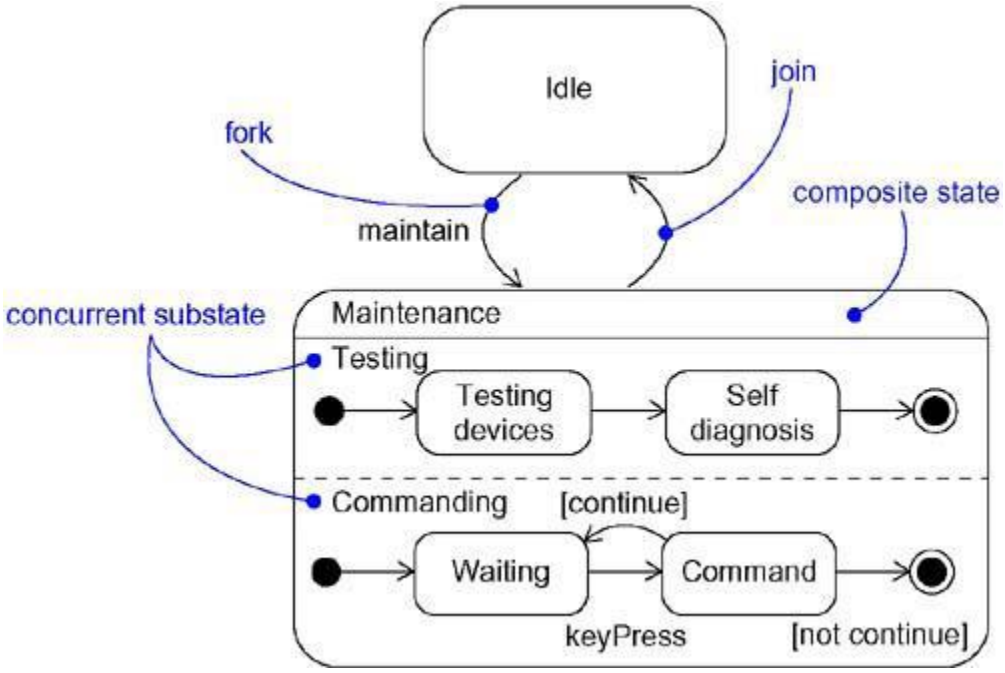


Figure 7: Concurrent Substates

Modeling the Lifetime of an Object

To model the lifetime of an object,

· Set the context for the state machine, whether it is a class, a use case, or the system as a whole.

If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.

If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.

- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- Expand these states as necessary by using substates.

- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.

· Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.

· After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure 8 shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house