LECTURE NOTES

ON

OBJECT ORIENTED ANALYSIS & DESIGN

2018 – 2019

III B. Tech I Semester (JNTUA-R15)

Mr. **D.Mukesh,** **Assistant Professor**

CHADALAWADA RAMANAMMA ENGINEERING COLLEGE

(AUTONOMOUS)

Chadalawada Nagar, Renigunta Road, Tirupati – 517 506

Department of Computer Science and Engineering

UNIT-1

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.

Systems: Systems are constructed by interconnecting components (Boundaries, Environments, Characters, Emergent Properties), which may well be systems in their own right. The larger the number of these components and relationships between them, higher will be the complexity of the overall system.

Complexity: Complexity depends on the number of the components embedded in them as well as the relationships and the interactions between these components which carry;

- Impossible for humans to comprehend fully
- Difficult to document and test
- Potentially inconsistent or incomplete
- Subject to change
- No fundamental laws to explain phenomena and approaches.

❖ The structure of Complex Systems

The structure of personal computer, plants and animals, matter, social institutions are some examples of complex system.

The structure of a Personal Computer: A personal computer is a device of moderate complexity. Major elements are CPU, monitor, keyboard and some secondary storage devices. CPU encompasses primary memory, an ALU, and a bus to which peripheral devices are attached. An ALU may be divided into registers which are constructed from NAND gates, inverters and so on. All are the hierarchical nature of a complex system.

The structure of Plants: Plants are complex multicellular organism which are composed of cells which is turn encompasses elements such as chloroplasts, nucleus, and so on. 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 water and minerals provided by stems to produce food through photosynthesis.

All parts at the same level of abstraction interact in well-defined ways. for example , at the highest level of abstraction, roots are responsible for observing water and minerals from the soil. Roots interact with stems, which transport these raw materials up to leaves. The leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

The structure of Animals: Animals exhibit a multicultural hierarchical structure in which collection of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system) and so on.

The structure of Matter: Nuclear physicists are concerned with a structural hierarchy of matter. Atoms are made up of electrons, protons and neutrons. Elements and elementary particles but protons, neutrons and other particles are formed from more basic components called quarks, which eventually formed from pro-quarks.

The structure of Social institutions: In social institutions, group of people join together to accomplish tasks that cannot be done by made of divisions which in turn contain branches which in turn encompass local offices and so on.

❖ The Inherent Complexity of Software

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements. They are

1. The complexity of the problem domain.
2. The difficultly of managing the developmental process.

3. The flexibility possible through software .
4. The problems of characterizing the behavior of discrete systems.

1. The complexity of the problem domain

The first reason has to do with the relationship between the application domains for which software systems are being constructed and the people who develop them. Often, although software developers have the knowledge and skills required to develop software they usually lack detailed knowledge of the application domain of such systems. This affects their ability to understand and express accurately the requirements for the system to be built which come from the particular domain. Note, that these requirements are usually themselves subject to change. They evolve during the construction of the system as well as after its delivery and thereby they impose a need for a continuous evolution of the system. Complexity is often increased as a result of trying to preserve the investments made in legacy applications. In such cases, the components which address new requirements have to be integrated with existing legacy applications. This results into interoperability problems caused by the heterogeneity of the different components which introduce new complexities.

2. The Difficulty of Managing the Development Process

The second reason is the complexity of the software development process. Complex software intensive systems cannot be developed by single individuals. They require teams of developers. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other. This complexity often gets even more difficult to handle if the teams do not work in one location but are geographically dispersed. In such situations, the management of these processes becomes an important subtask on its own and they need to be kept as simple as possible. None person can understand the system whose size is measured in hundreds of thousands, or even millions of lines of code. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules. The amount of work demands that we use a team of developers and there are always significant challenges associated with team development more developers means more complex communication and hence more difficult coordination.

3. The flexibility possible through software

Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.

The third reason is the danger of flexibility. Flexibility leads to an attitude where developers develop system components themselves rather than purchasing them from somewhere else. Unlike other industrial sectors, the production depth of the software industry is very large. The construction or automobile industries largely rely on highly specialized suppliers providing parts. The developers in these industries just produce the design, the part specifications and assemble the parts delivered.

The software development is different: most of the software companies develop every single component from scratch. Flexibility also triggers more demanding requirements which make products even more complicated.

Software offers the ultimate flexibility. It is highly unusual for a construction firm to build an on site steel mill to forge (create with hammer) custom girders (beams) for a new building. Construction industry has standards for quality of raw materials, few such standards exist in the software industry.

4. The problem of characterizing the behavior of discrete systems

The final reason for complexity according to Booch is related to the difficulty in describing the behavior of software systems. Humans are capable of describing the static structure and properties of complex systems if they are properly decomposed, but have problems in describing their behavior. This is because to describe behavior, it is not sufficient to list the properties of the system. It is also necessary to describe the sequence of the values that these properties take over time.

❖ The five Attributes of a complex system:

There are five attributes common to all complex systems. They are as follows:

1. **Hierarchic Structure**

   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.

   The fact that many complex systems have a nearly decomposable, and hierarchic structure is a major facilitating factor enabling us to understand, describe, and even "see" such systems their parts.

2. **Relative Primitives**

   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 class structure and the object structure are not completely independent each object in object structure represents a specific instance of some class.

3. **Separation of Concerns**

   Hierarchic systems decomposable because they can be divided into identifiable parts; he calls them nearly decomposable because their parts are not completly independent. This leads to another attribute common to all complex systems:

   Intra-component linkages are generally stronger than inter-component linkages. This fact has the involving the high frequency dynamics of the components-involving the internal structure of the components – from the low frequency dynamic involving interaction among components.

4. **Common Patterns**

   Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements. In other words, complex systems have common patterns. These patterns may involve the reuse of small components such as the cells found in both plants or animals, or of larger structures, such as vascular systems, also found in both plants and animals.

5. **Stable intermediate Forms**

   A complex system that works is invariably bound to have evolved from a simple system that worked …….. A complex system designed from scratch never works and can't be patched up to make it work. You have to start over, beginning with a working simple system.

❖ Organized and Disorganized Complexity

One mechanism to simplify concerns in order to make them more manageable is to identify and understand abstractions common to similar objects or activities. We can use a car as an example (which are considerable complex systems). Understanding common abstractions in this particular example would, for instance, involve the insight that clutch, accelerator and brakes facilitate the use of a wide range of devices, namely transport vehicles depending on transmission of power from engine to wheels.

Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other. In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, and electrical). In object-orientation, the class structure and the object structure relationship is the simplest form of related hierarchy. It forms a canonical representation for object oriented analysis.

The canonical form of a complex system

The discovery of common abstractions and mechanisms greatly facilitates are standing of complex system. For example, if a pilot already knows how to fly a given aircraft, it is easier to know how to fly a similar one. May different hierarchies are present within the complex system. For example an aircraft may be studied by decomposing it into its propulsion system. Flight control system and so on the decomposition represent a structural or "part of" hierarchy. The complex system also includes an "Is A" hierarchy. These hierodules for class structure and object structure combining the concept of the class and object structure together with the five attributes of complex system, we find that virtually all complex

system take on the same (canonical) form as shown in figure. There are two orthogonal hierarchies of system, its class structure and the object structure.

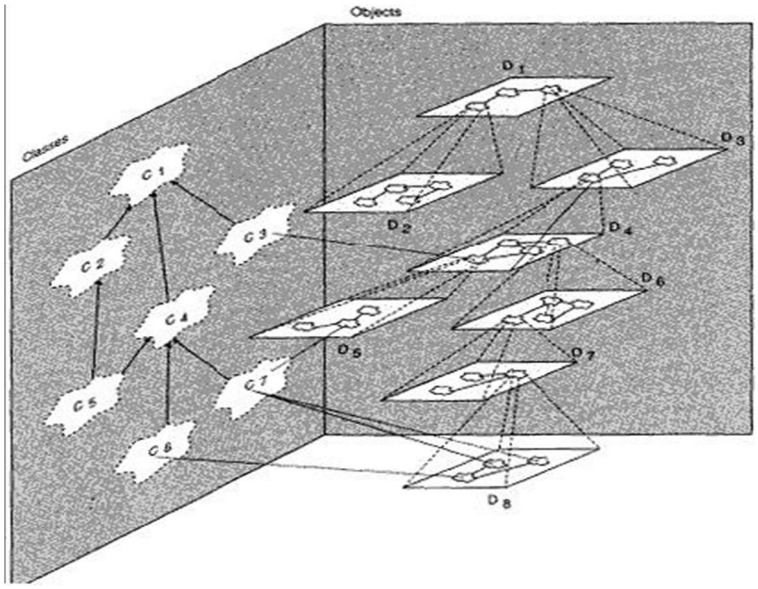


Figure : Canonical form of a complex system

The figure represents the relationship between two different hierarchies: a hierarchy of objects and a hierarchy of classes. The class structure defines the 'is-a' hierarchy, identifying the commonalities between different classes at different levels of abstractions. Hence class C4 is also a class C1 and therefore has every single property that C1 has. C4, however, may have more specific properties that C1 does not have; hence the distinction between C1 and C4. The object structure defines the 'part- of' representation. This identifies the composition of an object from component objects, like a car is composed from wheels, a steering wheel, a chassis and an engine. The two hierarchies are not entirely orthogonal as objects are instances of certain classes. The relationship between these two hierarchies is shown by identifying the instance-of relationship as well. The objects in component D8 are instances of C6 and C7 As suggested by the diagram, there are many more objects then there are classes. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

The Limitations of the human capacity for dealing with complexity:

Object model is the organized complexity of software. As we begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their interactions. This is an example of disorganized complexity. In complex system, we find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their intricate. This is an example in an air traffic control system, we must deal with states of different aircraft at once, and involving such it is absolutely impossible for a single person to keep track of all these details at once.

❖ Bringing Order to chaos

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

1. The Role of Abstraction: Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object. For example, when studying about how photosynthesis works in a plant, we can focus upon the chemical reactions in certain cells in a leaf and ignore all other parts such as roots and stems. Objects are abstractions of entities in the real world.

In general abstraction assists people's understanding by grouping, generalizing and chunking information. Object- orientation attempts to deploy abstraction. The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an 'is-a' relationship defines the inheritance between these classes.

2. The role of Hierarchy: Identifying the hierarchies within a complex software system makes understanding of the system very simple. The object structure is important because it illustrates how different objects collaborate with one another through pattern of interaction (called mechanisms). By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells, we come to explicitly distinguish the common and distinct properties of different objects, which helps to master their inherent complexity.

Different hierarchies support the recognition of higher and lower orders. A class high in the 'is-a' hierarchy is a rather abstract concept and a class that is a leaf represents a fairly concrete concept. The 'is-a' hierarchy also identifies concepts, such as attributes or operations, that are common to a number of classes and instances thereof. Similarly, an object that is up in the part-of hierarchy represents a rather coarse-grained and complex objects, assembled from a number of objects, while objects that are leafs are rather fine grained. But note that there are many other forms of patterns which are nonhierarchical: interactions, 'relationships'.

3. The role of Decomposition: Decomposition is important techniques for copying with complexity based on the idea of divide and conquer. In dividing a problem into a sub problem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to sub problems that are small enough so that they can be conquered. After all the sub problems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem. The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

Algorithmic (Process Oriented) Decomposition: In Algorithmic decomposition, each module in the system denotes a major step in some overall process.

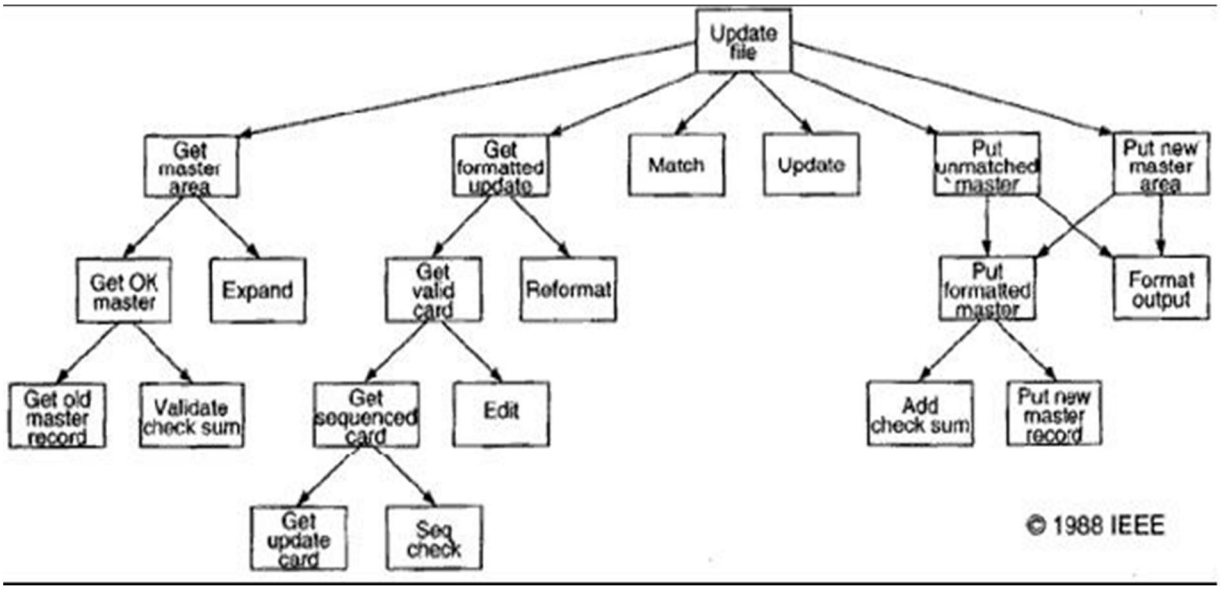


Figure : Algorithmic decomposition

Object oriented decomposition: Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem as shown in figure. We know the world as a set of autonomous agents that collaborate to perform some higher level behavior. 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. In this manner, each object in our solution embodies its own unique behavior .Each hierarchy in layered with the more abstract classes and objects built upon more primitive ones especially among the parts of the object structure, object in the real world. Here decomposition is based on objects and not algorithms.

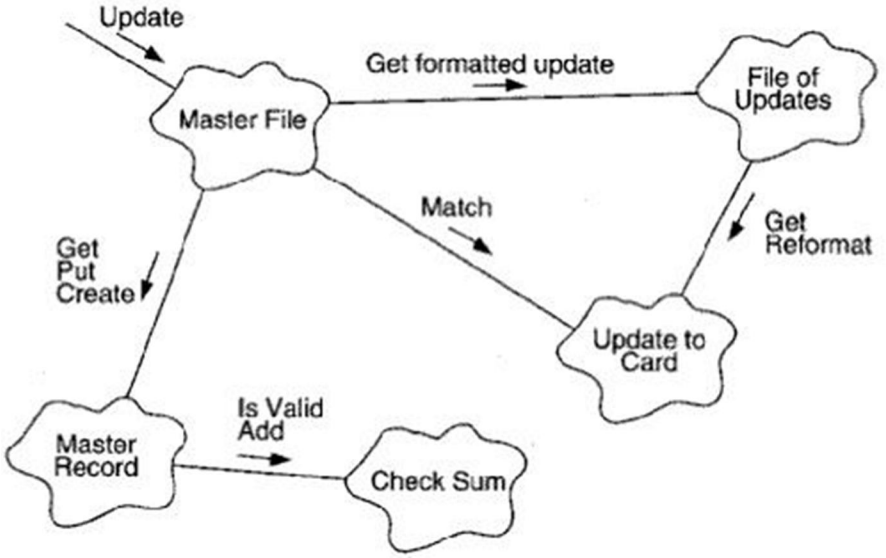


Figure 1.3: Object Oriented decomposition

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. We must start decomposing a system either by algorithms or by objects then use the resulting structure as the framework for expressing the other perspective generally object oriented view is applied because this approach is better at helping us organize the inherent complexity of software systems. object oriented algorithm has a number of advantages over algorithmic decomposition. Object oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression and are also more resident to change and thus better able to involve over time and it also reduces risks of building complex software systems. Object oriented decomposition also directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

Process-oriented decompositions divide a complex process, function or task into simpler sub processes until they are simple enough to be dealt with. The solutions of these sub functions then need to be executed in certain sequential or parallel orders in order to obtain a solution to the complex process. Object-oriented decomposition aims at identifying individual autonomous

objects that encapsulate both a state and a certain behavior. Then communication among these objects leads to the desired solutions.

Although both solutions help dealing with complexity we have reasons to believe that an object-oriented decomposition is favorable because, the object-oriented approach provides for a semantically richer framework that leads to decompositions that are more closely related to entities from the real world. Moreover, the identification of abstractions supports (more abstract) solutions to be reused and the object-oriented approach supports the evolution of systems better as those concepts that are more

likely to change can be hidden within the objects.

❖ On Designing Complex Systems

Engineering as a Science and an Art: Every engineering discipline involves elements of both science and art. The programming challenge is a large scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attribute of the competent engineer. The role of the engineer as artist is particularly challenging when the task is to design an entirely new system.

The meaning of Design: In every engineering discipline, design encompasses the discipline approach we use to invent a solution for some problem, thus providing a path from requirements to implementation. The purpose of design is to construct a system that.

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

According to Stroustrup, the purpose of design is to create a clean and relatively simple internal structure, sometimes also called as architecture. A design is the end product of the design process.

The Importance of Model Building: The buildings of models have 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. Models give us the opportunity to fail under controlled conditions. We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

The Elements of Software design Methods: Design of complex software system involves an incremental and iterative process. Each method includes the following:

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

The models of Object Oriented Development: 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. These models also over the spectrum of the important design decisions that we must consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well formed complex systems.

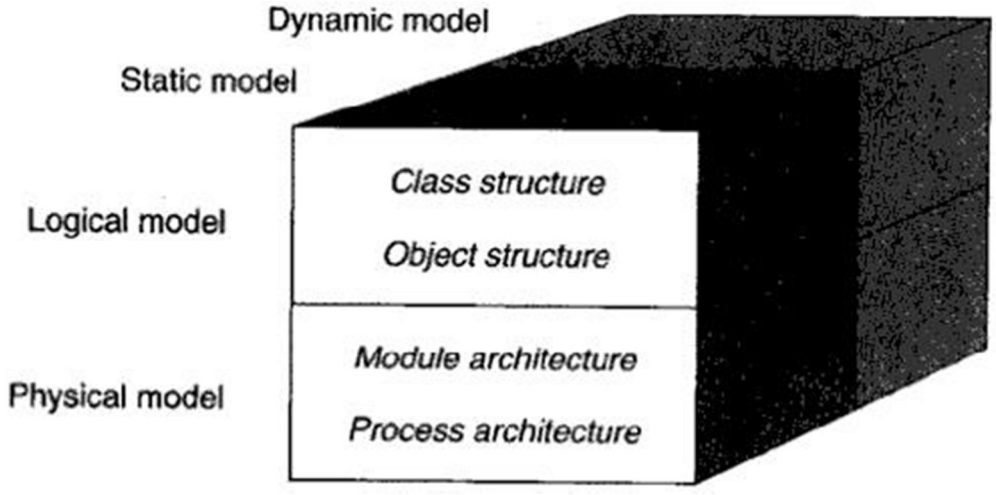


Figure :Models of object oriented development

Booch presents a model of object-oriented development that identifies several relevant perspectives. The classes and objects that form the system are identified in a logical model. For this logical model, again two different perspectives have to be considered. A static perspective identifies the structure of classes and objects, their properties and the relationships classes and objects participate in. A dynamic model identifies the dynamic behavior of classes and objects, the different valid states they can be in and the transitions between these states.

Besides the logical model, also a physical model needs to be identified. This is usually done later in the system's lifecycle. The module architecture identifies how classes are kept in separately compliable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those. Again for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication. Object-orientation has not, however, emerged fully formed. In fact it has developed over a long period, and continues to change.

The Object Model

The elements of the object oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

❖ The Evolution of the object Model

The generation of programming languages

Wegner has classified some of more popular programming languages in generations according to the language features.

1. First generation languages (1954 – 1958)
   - Used for specific & engineering application.
   - Generally consists of mathematical expressions.
   - For example: FORTRAN I, ALGOL 58, Flowmatic, IPLV etc.
2. Second generation languages (1959 – 1961)
   - Emphasized on algorithmic abstraction.
   - FORTRAN II - having features of subroutines, separate compilation
   - ALGOL 60 - having features of block structure, data type
   - COBOL - having features of data, descriptions, file handing
   - LISP - List processing, pointers, garbage collection
3. Third generation languages (1962 – 1970)
   - Supports data abstraction.
   - PL/1 – FORTRAN + ALGOL + COBOL
   - ALGOL 68 – Rigorous successor to ALGOL 60
   - Pascal – Simple successor to ALGOL 60
   - Simula - Classes, data abstraction
4. The generation gap (1970 – 1980)
   - C – Efficient, small executables
   - FORTRAN 77 – ANSI standardization
5. Object Oriented Boom (1980 – 1990)
   - Smalltalk 80 – Pure object oriented language
   - C++ - Derived from C and Simula
   - Ada83 – Strong typing; heavy Pascal influence
   - Eiffel - Derived from Ada and Simula

6. Emergence of Frameworks (1990 – today)

   - Visual Basic – Eased development of the graphical user interface (GUI) for windows applications
   - Java – Successor to Oak; designed for portability
   - Python – Object oriented scripting language
   - J2EE – Java based framework for enterprise computing
   - .NET – Microsoft's object based framework
   - Visual C# - Java competitor for the Microsoft .NET framework
   - Visual Basic .NET – VB for Microsoft .NET framework

Topology of first and early second generation programming languages

- Topology means basic physical building blocks of the language & how those parts can be connected.
- Arrows indicate dependency of subprograms on various data.
- Error in one part of program effect across the rest of system.

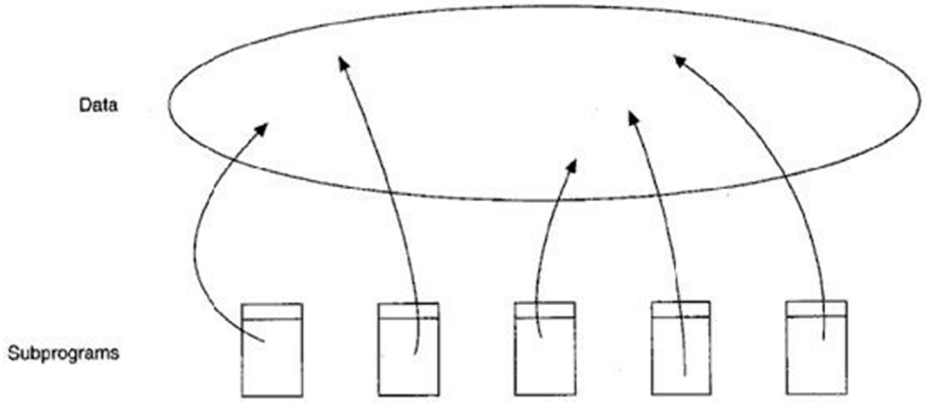


Fig : The Topology of First- and Early Second-Generation Programming Languages

Topology of late second and early third generation programming languages

Software abstraction becomes procedural abstraction; subprograms as an obstruction mechanism and three important consequences:

- Languages invented that supported parameter passing mechanism
- Foundations of structured programming were laid.
- Structured design method emerged using subprograms as basic physical blocks.

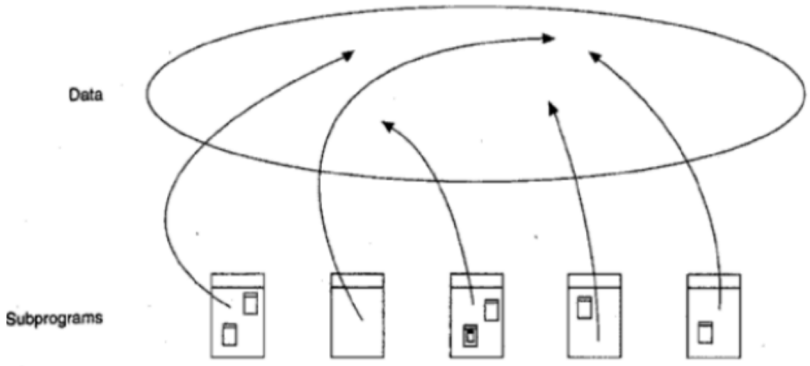


Fig : The Topology of Late Second- and Early Third-Generation Programming Languages

The topology of late third generation programming languages

- Larger project means larger team, so need to develop different parts of same program independently, i.e. complied module.
- Support modular structure.

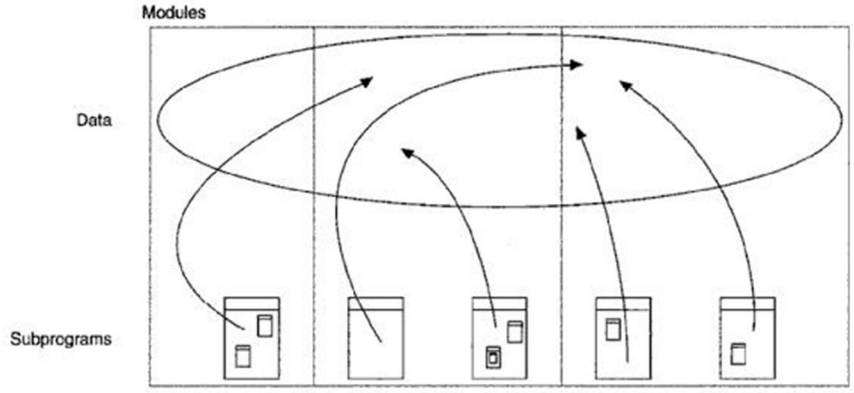


Fig : The Topology of Late Third-Generation Programming Languages

Topology of object and object oriented programming language

Two methods for complexity of problems

(i) Data driven design method emerged for data abstraction.

(ii) Theories regarding the concept of a type appeared

- Many languages such as Smalltalk, C++, Ada, Java were developed.
- Physical building block in these languages is module which represents logical collection of classes and objects instead of subprograms.
- Suppose procedures and functions are verbs and pieces of data are nouns, then
- Procedure oriented program is organized around verbs and object oriented program is organized around nouns.
- Data and operations are united in such a way that the fundamental logical building blocks of our systems are no longer algorithms, but are classes and objects.
- In large application system, classes, objects and modules essential yet insufficient means of abstraction.

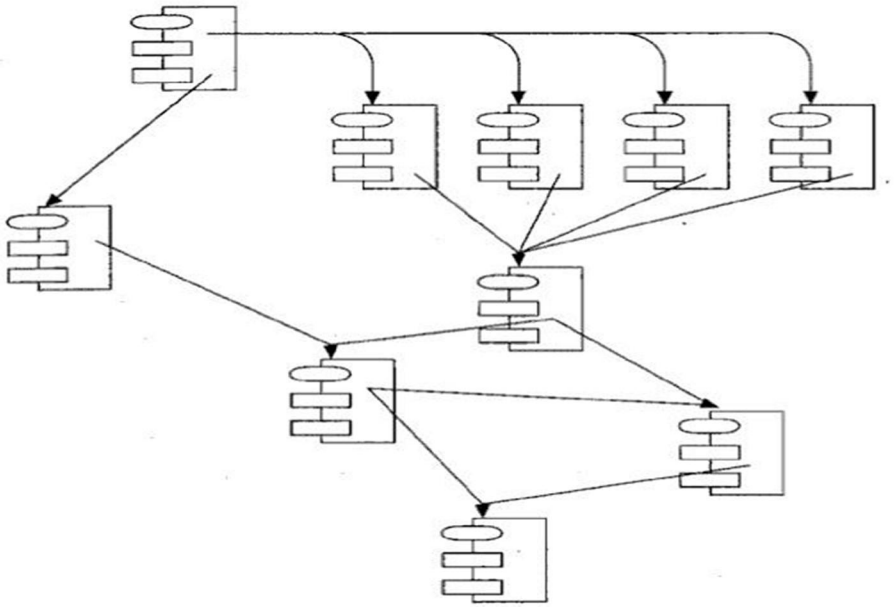


Fig : The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

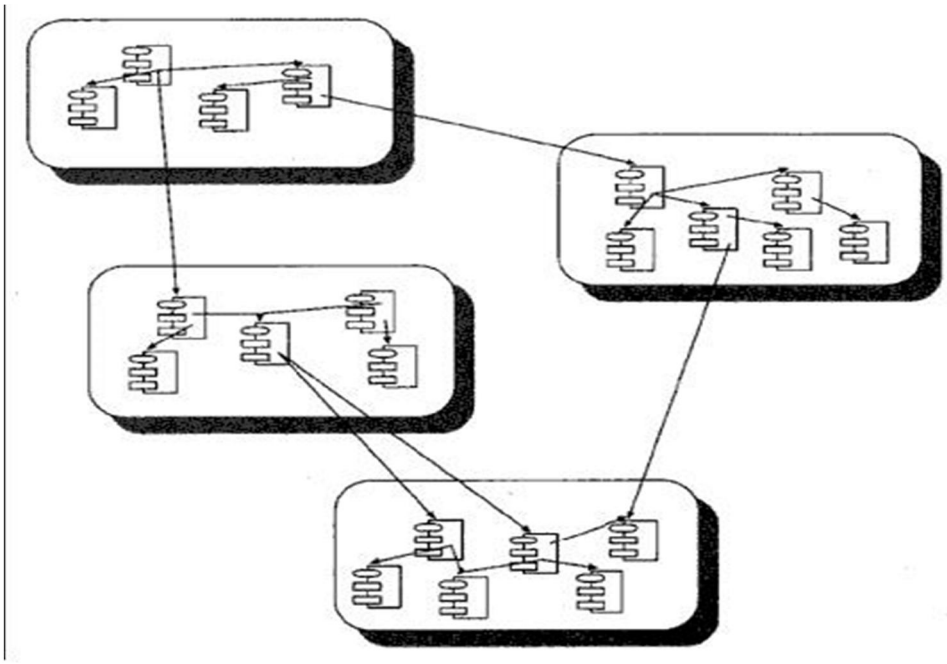


Fig : The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages.

❖ Foundations of the object model

In structured design method, build complex system using algorithm as their fundamental building block. An object oriented programming language, class and object as basic building block.
Following events have contributed to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hardware support for operating systems concepts
- Advances in programming languages, as 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

OOA (Object Oriented analysis)

During software requirement phase, requirement analysis and object analysis, it is a method of analysis that examines requirements from the perspective of classes and objects as related to problem domain. Object oriented analysis emphasizes the building of real-world model using the object oriented view of the world.

OOD (Object oriented design)

During user requirement phase, OOD involves understanding of the application domain and build an object model. Identify objects; it is methods of design showing process of object oriented decomposition. 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.

OOP (Object oriented programming)

During system implementation phase, t is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united in inheritance relationships. Object

oriented programming satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have associated type (class).
- Classes may inherit attributes from supertype to subtype.

❖ Elements of Object Model

Each requires a different mindset, a different way of thinking about the problem. Object model is the conceptual frame work for all things of object oriented.

There are four **major elements** of object model. They are:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

1. Abstraction

Abstraction is defined as a simplified description or specification of a system that emphasizes some of the system 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, not so significant, immaterial.

An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries on the perspective of the viewer. An abstraction focuses on the outside view of an object, Abstraction focuses up on the essential characteristics of some object, relative to the perspective of the viewer. From the most to the least useful, these kinds of abstraction include following.

- **Entity abstraction**: An object that represents a useful model of a problem domain or solution domain entity.
- **Action abstraction**: An object that provides a generalized set of operations all of which program the same kind of function.
- **Virtual machine abstractions**: An object that groups together operations that are used by some superior level of control, or operations that all use some junior set of operations.
- **Coincidental abstraction**: An object that packages a set of operations that have no relation to each other.

Abstraction: Temperature Sensor
temperature location

2. Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

3. Modularity

. Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- modules can be compiled separately. modules in C++ are nothing more than separately compiled files, generally called header files.
- Interface of module are files with .h extensions & implementations are placed in files with .c or .cpp suffix.
- Modules are units in pascal and package body specification in ada.
- modules Serve as physical containers in which classes and objects are declared like gates in IC of computer.
- Group logically related classes and objects in the same module.
- E.g. consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate their activities.

Example of modularity

Let's look at modularity in the Hydroponics Gardening System. Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Since one of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans (e.g., FruitGrowingPlan, GrainGrowingPlan). The implementations of these GrowingPlan classes would appear in the implementation of this module. We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

4. Hierarchy

Hierarchy is a ranking or ordering of abstractions Encapsulation hides company inside new of abstraction and modularity logically related abstraction & thus a set of abstractions form hierarchy. Hierarchies in complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Examples of Hierarchy: Single Inheritance

Inheritance defines a relationship among classes. Where one classes shares structure or behaviors defined in one (single in heritance) or more class (multiple inheritance) & thus represents a hierarchy of abstractions in which a subclass inherits from one or more super classes. Consider the different kinds of growing plans we might use in the Hydroponics Gardening System.

FruitGrowingPlan —— Is a ——▶ Growing plan

In this case, FruitGrowingPlan is more specialized, and GrowingPlan is more general. The same could be said for GrainGrowingPlan or VegetableGrowingPlan, that is, GrainGrowingPlan "is a" kind of GrowingPlan, and VegetableGrowingPlan "is a" kind of GrowingPlan. Here, GrowingPlan is the more general superclass, and the others are specialized subclasses.

Examples of Hierarchy: Multiple Inheritance

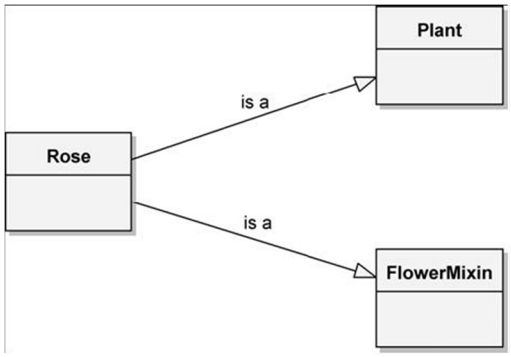


Figure 2.9: The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)

Hierarchy: Aggregation

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggregation relationships. For example, consider the abstraction of a garden. We can contend that a garden consists of a collection of plants together with a growing plan. In other words, plants are "part of" the garden, and the growing plan is "part of" the garden. This "part of" relationship is known as aggregation.

There are three **minor elements** which are useful but not essential part of object model. Minor elements of object model are:

1. Typing
2. Concurrency
3. Persistence

1. Typing

A type is a precise characterization of structural or behavioral properties which a collection of entities share. Type and class are used interchangeably class implements a type. Typing is the enforcement of the class of an object. Such that object of different types may not be interchanged. Typing implements abstractions to enforce design decisions. E.g. multiplying temp by a unit of force does not make serve but multiplying mass by force does. So this is strong typing. Example of strong and weak typing: In strong type, type conformance is strictly enforced. Operations can not be called upon an object unless the exact signature of that operation is defined in the object's class or super classes.

Benefits of Strongly typed languages:

- Without type checking, program can crash at run time
- Type declaration help to document program
- Most compilers can generate more efficient object code if types are declared.

Examples of Typing: Static and Dynamic Typing

Static typing (static binding/early binding) refers to the time when names are bound to types i.e. types of all variables are fixed at the time of compilation. Dynamic binding (late binding) means that types of all variables and expressions are not known until run time. Dynamic building (object pascal, C++) small talk (untyped).

2. Concurrency

OO-programming focuses upon data abstraction, encapsulation and inheritance concurrency focuses upon process abstraction and synchronization. Each object may represent a separate thread of actual (a process abstraction). Such objects are called active. In a system based on an object oriented design, we can conceptualize the word as consisting of a set of cooperative objects, some of which are active (serve as centers of independent activity) . Thus concurrency is the property that distinguishes an active object from one that is not active. For example: If two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of object being acted upon is not computed when both active objects try to update their state simultaneously. In the preserve of concurrency, it is not enough simply to define the methods are preserved in the presence of multiple thread of control.

Examples of Concurrency

Let's consider a sensor named ActiveTemperatureSensor, whose behavior requires periodically sensing the current temperature and then notifying the client whenever the temperature changes a certain number of degrees from a given setpoint. We do not explain how the class implements this behavior. That fact is a secret of the implementation, but it is clear that some form of concurrency is required.

There are three approaches to concurrency in object oriented design

- Concurrency is an intrinsic feature of languages. Concurrency is termed as task in ada, and class process in small talk. class process is used as super classes of all active objects. we may create an active object that runs some process concurrently with all other active objects.
- We may use a class library that implements some form of light weight process AT & T task library for C++ provides the classes' sched, Timer, Task and others. Concurrency appears

through the use of these standard classes.

- Use of interrupts to give the illusion of concurrency use of hardware timer in active Temperature sensor periodically interrupts the application during which time all the sensor read the current temperature, then invoke their callback functions as necessary.

3. Persistence

Persistence is the property of an object through which its existence transcends time and or space i.e. objects continues to exist after its creator ceases to exist and/or the object's location moves from the address space in which it was created. An object in software takes up some amount of space and exists for a particular amount of time. Object persistence encompasses the followings.

- Transient results in expression evaluation
- Local variables in procedure activations
- Global variables where exists is different from their scope
- Data that exists between executions of a program
- Data that exists between various versions of the program
- Data that outlines the Program.

❖ Applying the Object Model

Benefits of the Object Model:

Object model introduces several new elements which are advantageous over traditional method of structured programming. The significant benefits are:

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilent to change.
- Object model appears to the working of human cognition, many people who have no idea how a computer works find the idea of object oriented systems quite natural.

Application of Object Model

OOA & Design may be in only method which can be employed to attack the complexity inherent in large systems. Some of the applications of the object model are as follows:

- Air traffic control
- Animation
- Business or insurance software
- Business Data Processing
- CAD
- Databases
- Expert Systems
- Office Automation
- Robotics
- Telecommunication
- Telemetry System etc.

Two-marks Questions

1. Define Software complexity.
2. List out five attributes of complex system
3. list elements of object model.
4. Distinguish between object-oriented and object-based languages.

Previous Questions

1. Explain structure of complex systems
2. State and explain five attributes of a complex system.
3. Illustrate the design of complex system.
4. Discuss the evolution of object model with an example.
5. a. Explain the benefits of object model.
   b. Explain the application of an object model.

UNIT II
Classes and Objects

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

❖ The nature of an object

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 the object plus the current (usually dynamic) values of each of these properties.

Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when someone puts money in a slot and pushes a button to make a selection, a drink emerges from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing because the user has violated the basic assumptions of their operation. Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says, "Correct change only," and puts in extra money. Most machines are user-hostile; they will happily swallow the excess money.

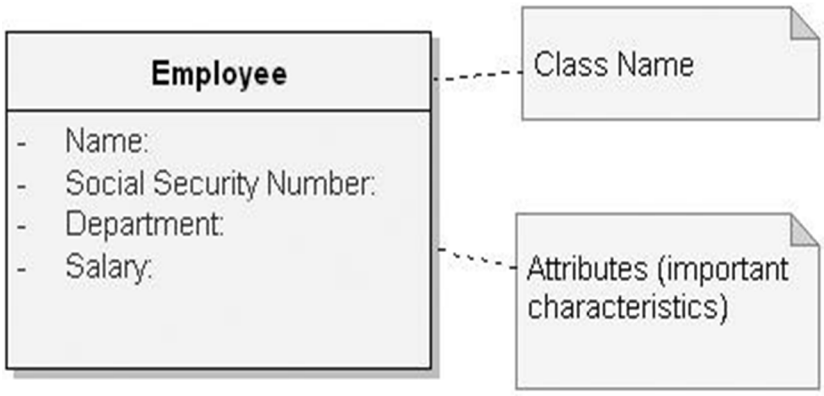


Figure :Employee Class with Attributes

Figure : Employee Objects Tom and Kaitlyn

Behavior

Behavior is how an object acts and reacts, in terms of its state changeable state of object affect its behavior. In vending machine, if we don't deposit change sufficient for our selection, then the machine will probably do nothing. So behavior of an object is a function of its state as well as the operation performed upon it. The state of an object represents the cumulative results of its behavior.

An operation is some action that one object performs on another in order to elicit a reaction.

For example, a client might invoke the operations append and pop to grow and shrink a queue object, respectively. A client might also invoke the operation length, which returns a value denoting the size of the queue object but does not alter the state of the queue itself.

Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well.

Operations

An operation denotes a service that a class offers to its clients. A client performs 5 kinds of operations upon an object.

In practice, we have found that a client typically performs five kinds of operations on an object

The three most common kinds of operations are the following:

- **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. In queue example operations, clear, append, pop, remove) are modifies, const functions (length, is empty, front location) are selectors.

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.

Roles and Responsibilities

A role is a mask that an object wears and so defines a contract between an abstraction and its clients. 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".

The state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction's responsibilities.

Examples:

1. A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money. However, to a taxing authority, the account may play the role of an entity whose dividends must be reported on annually.
2. To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, the same share denotes a legal instrument to which are attached certain rights.
3. In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

Identity

Identity is that property of an object which distinguishes it from all others.

Example:

Consider a class that denotes a display item. A display item is a common abstraction in all GUI-centric systems: It represents the base class of all objects that have a visual representation on some window and so captures the structure and behavior common to all such objects. Clients expect to be able to draw, select, and move display items, as well as query their selection state and location. Each display item has a

location designated by the coordinates x and y.

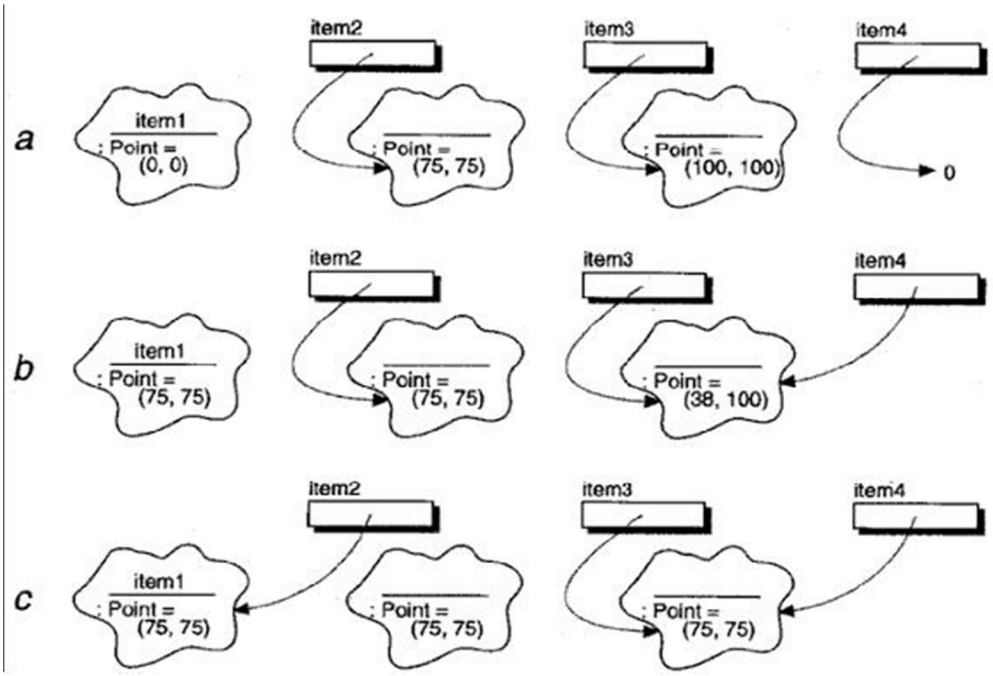


Figure : Object Identity

First declaration creates four names and 3 distinct objects in 4 diff location. Item 1 is the name of a distinct display item object and other 3 names denote a pointer to a display item objects. Item 4 is no such objects, we properly say that item 2 points to a distinct display item object, whose name we may properly refer to indirectly as * item2. The unique identity (but not necessarily the name) of each object in preserved over the lifetime of the object, even when its state is changed. Copying, Assignment, and Equality Structural sharing takes place when the identity of an object is aliased to a second name.

❖ Relationships among Objects

Objects contribute to the behavior of a system by collaborating with one another.
E.g. object structure of an airplane.

The relationship between any two objects encompasses the assumptions that each makes about the other including what operations can be performed. There are Two kinds of objects relationships are links and aggregation.

1.Links

- A link denotes the specific association through which one object (the client) applies the services of another object (the supplier) or through which are object may navigate to another.
- A line between two object icons represents the existence of pass along this path.
- Messages are shown as directed lines representing the direction of message passing between two objects is typically unidirectional, may be bidirectional data flow in either direction across a link.

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

- **Controller:** This object can operate on other objects but is not operated on by other objects. In some contexts, the terms active object and controller are interchangeable.
- **Server:** This object doesn't operate on other objects; it is only operated on by other objects.
- **Proxy:** This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.

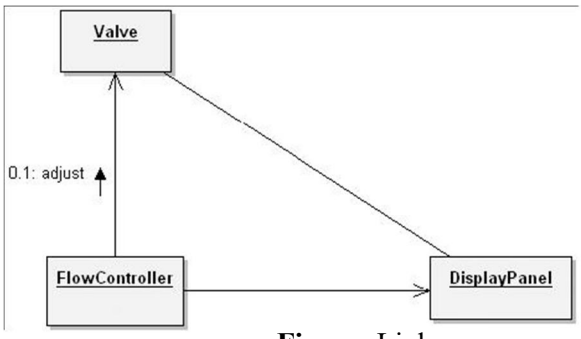


Figure : Links

In the above figure, FlowController acts as a controller object, DisplayPanel acts as a server object, and Valve acts as a proxy.

Visibility

Consider two objects, A and B, with a link between the two. In order for A to send a message to object B, B must be visible to A. Four ways of visibility

- 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 locally declared object in some operation of the client.

Synchronization

Wherever one object passes a message to another across a link, the two objects are said to be synchronized. Active objects embody their own thread of control, so we expect their semantics to be guaranteed in the presence of other active objects. When one active object has a link to a passive one, we must choose one of three approaches to synchronization.

1. **Sequential:** The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
2. **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.
3. **Concurrent:** The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

2.Aggregation

.

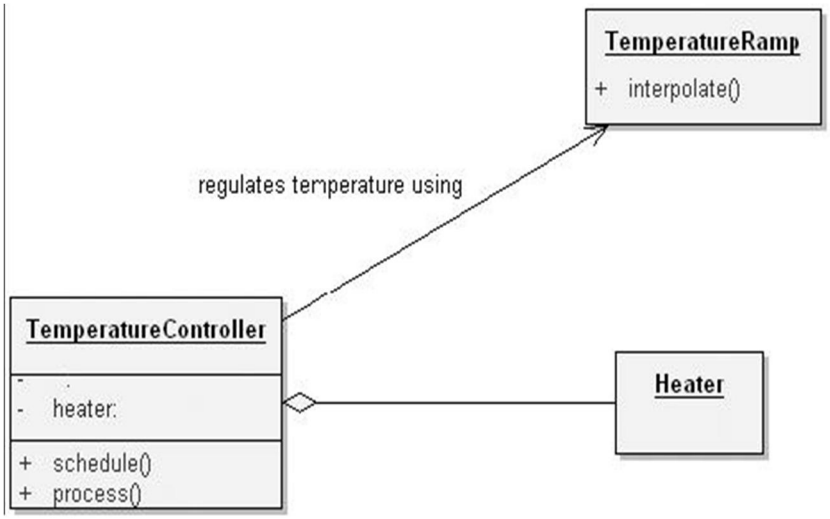


Figure : Aggregation

Whereas 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. Aggregation is a specialized kind of association. Aggregation may or may not denote physical containment. E.g. airplane is composed of wings, landing gear, and so on. This is a case of physical containment. The relationship between a shareholder and her shares is an aggregation relationship that doesn't require physical containment.

❖ The Nature of the class

A class is a set of objects that share a common structure, common behavior and common 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 secrets of its behavior. The interface primarily consists of the declarations of all the operators applicable to instance of this class, but it may also include the declaration of other classes, constants variables and exceptions as needed to complete the abstraction. The implementation of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class consists of the class. Interface of the class is divided into following four parts.

- **Public**: a declaration that is accessible to all clients
- **Protected**: a declaration that is accessible only to the class itself and its subclasses
- **Private:** a declaration that is accessible only to the class itself
- **Package:** a declaration that is accessible only by classes in the same package.

❖ Relationship among Classes

We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some kind of sharing. Second, a class relationship might indicate some kind of semantic connection.

There are four basic kinds of class relationships. They are Association, Aggregation, Inheritance (Generalization) and Dependency

1. **Association:**

   An association is a structural relationship that describes a set of links, a link being connection among objects. 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 an example for a vehicle, two of our key abstractions include the vehicle and wheels. As shown in Figure , we may show a simple association between these two classes: the class Wheel and the class Vehicle.

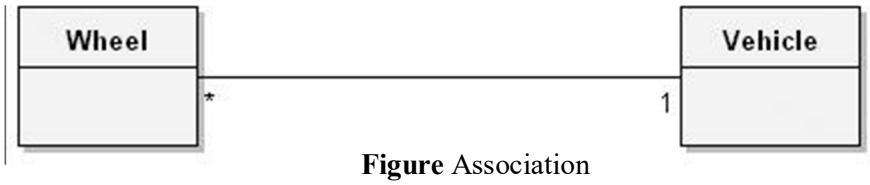


Figure Association

Multiplicity/Cardinality

This multiplicity denotes the cardinality of the association. There are three common kinds of multiplicity across an association:

1. One-to-one 2. One-to-many 3. Many-to-many

2. Aggregation

Aggregation is a special kind of association, representing a structural relation ship between a whole and its parts. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

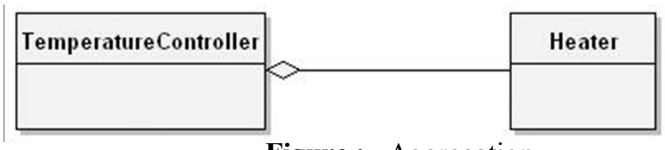


Figure : Aggregation

As shown in Figure, the class Temperature Controller denotes the whole, and the class Heater is one of its parts.

3. Inheritance

Inheritance, perhaps the most semantically interesting of the 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. Inheritance means that subclasses inherit the structure of their superclass.

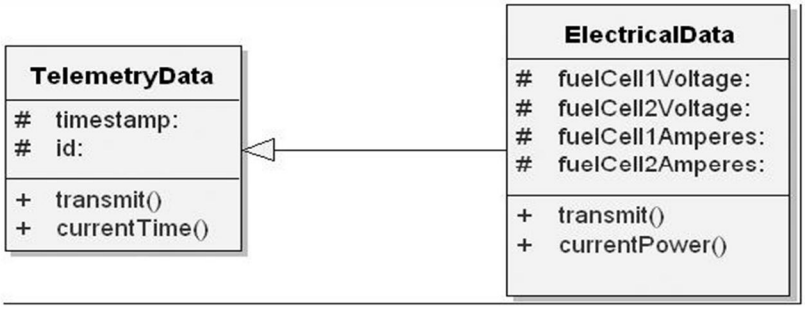


Figure : ElectricalData Inherits from the Superclass TelemetryData

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 Inheritence:

The derived class having only one base class is called single inheritance. It is a relationship among classes where in one class shares the structure and/or behavior defined one class.

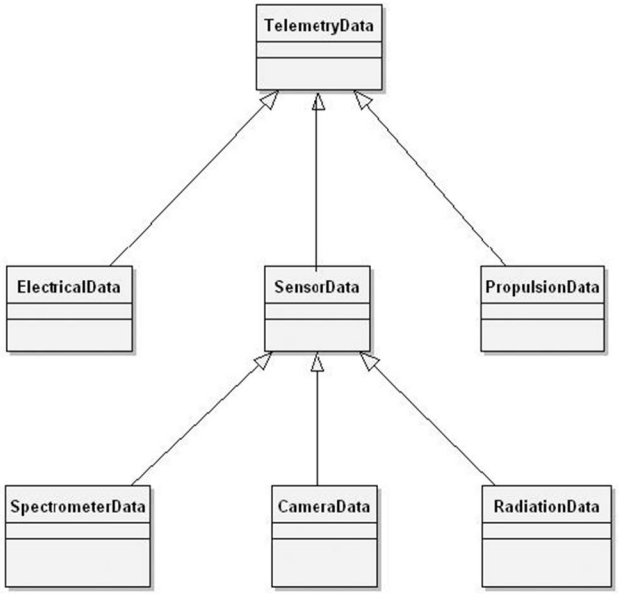
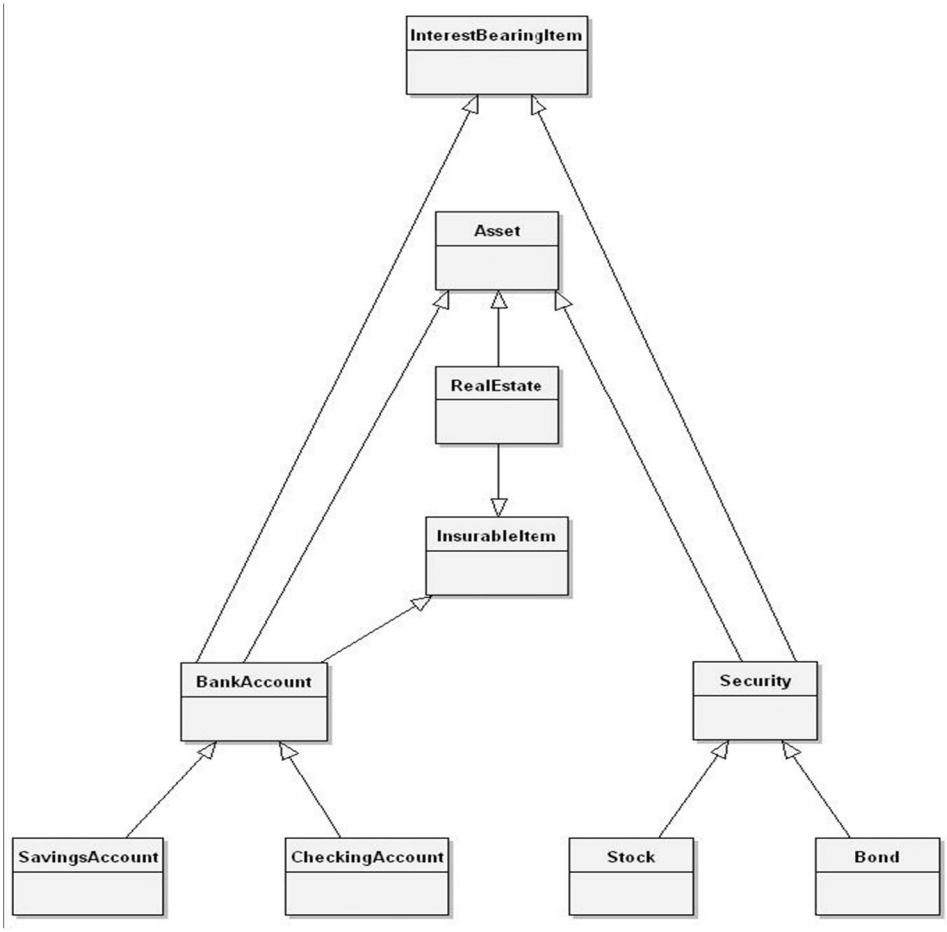


Figure illustrates the single inheritance relationships deriving from the superclass TelemetryData. Each directed line denotes an "is a" relationship. For example, CameraData "is a" kind of SensorData, which in turn "is a" kind of TelemetryData.

Multiple Inheritence:

The derived class having more than one base class is known as multiple inheritance. It is a relationship among classes where in one class shares the structure and/or behavior defined more classes.



Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance.Above Figure 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.

Polymorphism

Polymorphism is an ability to take more than one form. It 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.

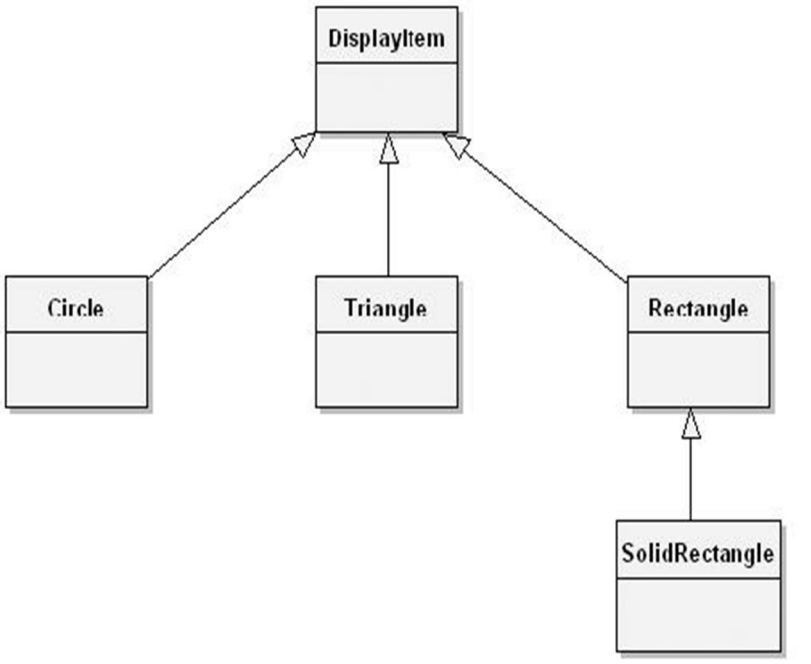


Figure : Polymorphisms

Consider the class hierarchy in Figure, 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 theCenter (denoting the coordinates for the center of the displayed item), along with the following operations:

- draw: Draw the item.
- move: Move the item.
- location: Return the location of the item.

The operation location is common to all subclasses and therefore need not be redefined, but we expect the operations draw and move to be redefined since only the subclasses know how to draw and move themselves.

❖ The interplay of classes and relationships:

- Classes and objects are separate yet intimately related concepts.
- Specifically, every object is the instance of some class, and every class has zero or more instances.
- For all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program.
- Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed.
- Objects are typically created and destroyed at a furious rate during the lifetime of an application

Relationship between classes and objects:

For example, consider the classes and objects in the implementation of an air traffic control system. Some of the more important abstractions include planes, flight plans, runways and air spaces. These classes and objects are relatively static. Conversely, the instances of these classes are dynamic. At a fairly slow rate, new run ways are built and old ones are deactivated.

The 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 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

Collectively, we call such classes and objects the *key abstractions* of the problem, and We call these cooperative structures the *mechanisms* of the implementation.

Part-II(Classification)

❖ Importance of proper classification

Classification is the means whereby we order knowledge. There is no any golden path to classification. Classification and object oriented development: The identification of classes and objects is the hardest part of object oriented analysis and design, identification involves both discovery and invention. Discovery helps to recognize the key abstractions and mechanisms that form the vocabulary of our problem domain. Through invention, we desire generalized abstractions as well as new mechanisms that specify how objects collaborate discovery and inventions are both problems of classifications.

Intelligent classification is actually a part of all good science class of should be meaningful is relevant to every aspect of object oriented design classify helps us to identify generalization, specialization, and aggregation hierarchies among classes classify also guides us making decisions about modularizations.

The difficulty of classification

Examples of classify: Consider start and end of knee in leg in recognizing human speech, how do we know that certain sounds connect to form a word, and aren't instead a part of any surrounding words?

In word processing system, is character class or words are class? Intelligent classification is difficult e.g. problems in classify of biology and chemistry until 18^{th} century, organisms were arranged from the most simple to the most complex, human at top of the list. In mid 1970, organisms were classified according to genus and species. After a century later, Darwin's theory came which was depended upon an intelligent classification of species. Category in biological
taxonomy is the kingdom, increased in order from phylum, subphylum class, order, family, genus and finally species. Recently classify has been approached by grouping organisms that share a common generic heritage i.e. classify by DNA. DNA in useful in distinguishing organisms that are structurally similar but genetically very different classify depends on what you want classification to do. In ancient times, all substances were through to be sure ambulation of earth, fire, air and water. In mid 1960s – elements were primitive abstractive of chemistry in 1869 periodic law came.

The incremental and iterative nature of classification

Intelligent classification is intellectually hard work, and that it best comes about through an incremental and iterative process. Such processes are used in the development of software technologies such as GUI, database standards and programming languages. The useful solutions are understood more systematically and they are codified and analyzed. 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 in the class structure early in a design and then revise this structure over time. Only at later in the design, once clients have been built that use this structure, we can meaningfully evaluate the quality of our classification. On the basis of this experience, we may decide to create new subclasses from existing once (derivation). We may split a large class into several smaller ones (factorization) or create one large class by uniting smaller ones (composition). Classify is hard

because there is no such as a perfect classification (classify are better than others) and intelligent classify requires a tremendous amount of creative insight.

❖ Identifying classes and objects

Classical and modern approaches: There are three general approaches to classifications.

- Classical categorization
- Conceptual clustering
- Prototypal theory

1. Classical categorizations

All the entities that have a given property or collection of properties in common forms a category. Such properties are necessary and sufficient to define the category. i.e. married people constitute a category i.e. either married or not. The values of this property are sufficient to decide to which group a particular person belongs to the category of tall/short people, where we can agree to some absolute criteria. This classification came from Plato and then from Aristotle's classification of plants and animals.

This approach of classification is also reflected in modern theories of child development. Around the age of one, child typically develops the concept of object permanence, shortly thereafter, the child acquires skill in classifying these objects, first using basic category such as dogs, cats and toys.

2. Conceptual clustering

It is a more modern variation of the classical approach and largely derives from attempts to explain how knowledge is represented in this approach, classes are generated by first formulating conceptual description of these classes and then classifying the entities according to the descriptions. e.g. we may state a concept such as "a love song". This is a concept more than a property, for the "love songness" of any song is not something that may be measured empirically. However, if we decide that a certain song is more of a love song than not, we place it in this category. thus this classify represents more of a probabilistic clustering of objects and objects may belong to one or more groups, in varying degree of fitness conceptual clustering makes absolute judgments of classify by focusing upon the best fit.

3. Prototype theory

It is more recent approach of classify where a class of objects is represented by a prototypical object, an object is considered to be a member of this class if and only if it resembles this prototype in significant ways. e.g. category like games, not in classical since no single common properties shared by all games, e.g. classifying 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 system.

Object oriented Analysis

The boundaries between analysis and design are furry, although the focus of each is quite district. An analysis, we seek to model the world by discovering. The classes and objects that form the vocabulary of the problem domain and in design, we invent the abstractions and mechanisms that provide the behavior that this model requires following are some approaches for analysis that are relevant to object oriented system.

Classical approaches

It is one of approaches for analysis which derive primarily from the principles of classical categorization. e.g. Shlaer and Mellor suggest that classes and objects may come from the following sources:

- Tangible things, cars, pressure sensors

- Roles – Mother, teacher, politician
- Events – landing, interrupt
- Interactions – meeting

From the perspective of database modeling, ross offers the following list:

(i) People – human who carry out some function
(ii) Places – Areas set for people or thing
(iii) Things – Physical objects (tangible)
(iv) Organizations – organized collection of people resources
(v) Concepts – ideas
(vi) Events – things that happen

Coad and Yourdon suggest another set of sources of potential objects.

(i) Structure
(ii) Dences
(iii) Events remembered (historical)
(iv) Roles played (of users)
(v) Locations (office, sites)
(vi) Organizational units (groups)

Behavior Analysis

Dynamic behavior also be one of the primary source of analysis of classes and objects things can are grouped that have common responsibilities and form hierarchies of classes (including superclasses and subclasses). System behaviors of system are observed. These behaviors are assigned to parts of system and tried to understand who initiates and who participates in these behaviors. A function point is defined as one and user business functions and represents some kind of output, inquiry, input file or interface.

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, compliers, missile systems etc. Domain analysis defined as an attempt to identify the objects, operations and, relationships that are important to particular domain.

More and Bailin suggest the following steps in domain analysis.

i) Construct a strawman generic model of the domain by consulting with domain expert.
ii) Examine existing system within the domain and represent this understanding in a common format.
iii) Identify similarities and differences between the system by consulting with domain expert.
iv) 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

Earlier approaches require experience on part of the analyst such a process is neither deterministic nor predictably successful. Use case analysis can be coupled with all three of these approaches to derive the process of analysis in a meaningful way. Use case is defined as a particular form pattern or exemplar some transaction or sequence of interrelated events. Use case analysis is applied as early as requirements analysis, at which time end users, other domain experts and the development team enumerate the scenarios that are fundamental to system's operation. These scenarios collectively describe the system functions of the application analysis then proceeds by a study of each scenario. As the team walks through each scenario, they must identify the objects that participate in the scenario, responsibilities

of each object and how those objects collaborate with other objects in terms of the operations each invokes upon the other.

CRC cards

CRC are a useful development tool that facilitates brainstorming and enhances communication among developers. It is 3 x 5 index card (class/Responsibilities/collaborators i.e. CRC) upon which the analyst writes in pencil with the name of class (at the top of 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 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. Nouns represent candidate objects and the verbs represent candidate operations upon them. it is simple and forces the developer to work in the vocabulary of the problem space.

Structured Analysis

Same as English description as an alternative to the system, many CASE tools assists in modeling of the system. In this approach, we start with an essential model of the system, as described by data flow diagrams and other products of structured analysis. From this model we may proceed to identify the meaningful classes and objects in our problem domain in 3 ways.

- Analyzing the context diagrams, with list of input/output data elements; think about what they tell you or what they describe e.g. these make up list of candidate objects.
- Analyzing data flow domains, candidate objects may be derived from external entities, data stores, control stores, control transformation, candidate classes derive from data flows and candidate flows.
- By abstraction analysis: In structured analysis, input and output data are examined and followed inwards until they reach the highest level of abstraction.

❖ Key abstractions and mechanisms

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 problems. They highlight the things that are in the system and therefore relevant to our design and suppress the things that are outside of system.

Identification of Abstractions:

identification of key abstraction involves two processes. Discovery and invention through discovery we come to recognize the abstraction used by domain experts. If through inventions, we create new classes and objects that are not 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. These key abstractions are artifacts of the particular design, not of the problem domain.

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 thought 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. Sometimes we may find a general subclass and so may choose to move it up in the class structure, thus increasing the degree of sharing. This is called class promotion. Similarly, we may find a class to be too general, thus making inheritance by a

subclass difficult because of the large semantic gap. This is called a grain size conflict.
Naming conventions are as follows:

- Objects should be named with proper noun phrases such as the sensor or simply shapes.
- Classes should be named with common noun phrases, such as sensor or shapes.
- Modifier operations should be named with active verb phrases such as draw, moveleft.
- Selector operations should imply a query or be named with verbs of the form "to be" e.g. is open, extent of.

Identifying Mechanisms

A mechanism is a design decision about how collection of objects cooperates. Mechanisms represent patterns of behavior e.g. 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. Any mechanism may be employed as long as it delivers the required behavior and thus which mechanism is selected is largely a matter of design choice. Any of the following design might be considered.

- A mechanical linkage from the acceleration to the (the most common mechanism)
- An electronic linkage from a preserve 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. Its rate of flow is regulated by a clip around the fuel the pushing on the accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low cost mechanism)

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 new, the model being viewed and some client that knows when to display this model. The client first tells the window to draw itself. Since it may encompass several subviews, the window next tells each if its subviews to draw them. Each subview in turn tells the model to draw itself ultimately resulting in an image shown to the user.

PREVIOUS QUESTIONS

1. Discuss about possible relationships formed among the objects with example.
2. Describe how to identify classes and objects in detail.
3. Explain interplay of classes and objects.
4. State importance of proper classification using an example.
5. Describe Key abstraction.

Two Marks Questions

1. What are the different class associations?
2. State about relationships among objects.
3. What are the advantages of classification?
4. Explain CRC card. Write uses of crc cards.
5. Write about association.
6. What is the main impact of object oriented approach?
7. Enumerate three most common operations on object.
8. Examine the nature of class.

UNIT-III

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

Why we model

The importance of Modeling:

A model is a simplification of reality. A model provides the blueprints of a system. Every system may be described from different aspects using different models, and each model is therefore a semantically closed abstraction of the system.

A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system. 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.

Modeling is not for big systems. Even the software equivalent of a dog house can benefit from some modeling. For example if you want to build a dog house ,you can start with a pile of lumber, some nails, and a basic tools, such as a hammer,saw,and a tape measure. In few hours, with little prior planning, you will likely
end up a with a dog house that's reasonably functional. finally your will happy and get a less demanding dog.

if you want to build a house for your family , you can start with a pile of lumber, some nails, and a basic tools. But it's going to take you a lot longer, and your family will certainly be more demanding than the dog. If you want to build a quality house that meets the needs of your family and you will need to draw some blue prints.

Principles of Modeling:

UML is basically a modeling language; hence its principles will also be related to modeling concepts. Here are few basic principal of UML.

First: "The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped"

In other words ,choose your models well. The right models will brilliantly illuminate the most wicked development problems. The wrong models will mislead you, causing you to focus on irrelevant issues.

Second: " Every model may be expressed at different levels of precision ".

Best approach to a given problem results in a best model. If the problem is complex mechanized level of approach & if the problem is simple decent approach is followed.

Third: "The best models are connected to reality."

The model built should have strong resemblance with the system.

Fourth: " No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models."

If you constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you will need floor plans,elevations,electical plans, heating plans, and plumbing plans.

Object-Oriented Modeling:

In software , there are several ways to approaches a model. The two most common ways are

1. Algorithmic perspective
2. Object-Oriented perspective

1. **Algorithmic perspective:**

In this approach, the main building blocks of all software is the procedure or functon.This view leads developers to focus on issues of control and decomposition of larger algorithms into smaller ones.

2. Object-Oriented perspective:

In this approach, the main building blocks of all software is the object or class. Simply put, an object is a thing. A class is a description of a set of common objects. Every object has identity, state and behavior.

For example, consider a simple a three-tier -architecture for a billing system, involving a user interface ,middleware, and a data base. In the user interface, you will find concrete objects, such as buttons, menus, and dialog boxes. In the database, you will find concrete objects ,such as tables. In the middle layer ,you will find objects such as transitions and business rules.

→A Conceptual Model of the UML

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

* Basic Building blocks of the UML
* Rules of the UML
* Common mechanisms in the UML.

**Basic Building Blocks of the UML:

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things in the UML
2. Relationships in the UML
3. Diagrams in the UML

1.Things in the UML: -

Things are the abstractions that are first-class citizens in a model. There are four kinds of things in the UML:

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

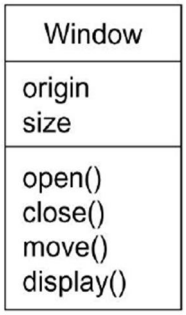
1. Structural things:

Structural things are the nouns of UML models. These are the mostly static parts of a model. There are seven kinds of structural things. They are a. Class

b. Interface
c. Collaboration
d. Use case
e. Activity Class
f. Component
g. Node

a) C**lass:**

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



b) Interface:

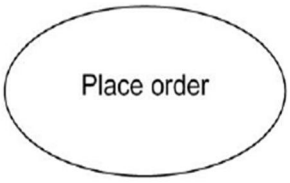
An **i**nterface is a collection of operations that specify a service of a class or component. Graphically, an interface is rendered as a circle together with its name.

C) Collaboration:

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

d) Use case:

A 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. A 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.

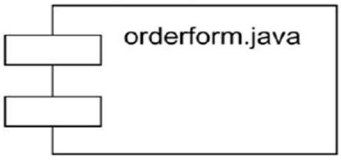


e)Active class

An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations.

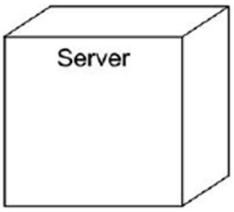
f) Component:

A 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, usually including only its name.



g) N**ode:**

A **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:

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. There are two primary kinds of behavioral things. They are

a) Interaction
b)State machine

a) Interaction

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

b)State machine

A State machine a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition).Graphically, a state is rendered as a **rounded rectangle**, usually including its name and its sub states.

3. Grouping things:

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

package:

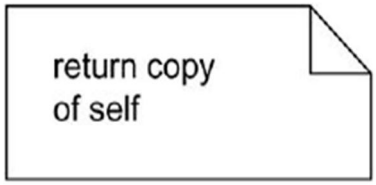
A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package.Graphically, a package is rendered as a **tabbed folder**, usually including only its name and, sometimes, its contents.

4.Annotational things:

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note.

Note:

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.



1.Relationships in the UML:

Relationships are used to connect things. There are four kinds of relationships in the UML:

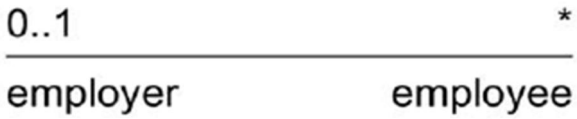
1. Dependency
2. Association
3. Generalization
4. Realization

1. **Dependency:**

A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.

2. **Association:**

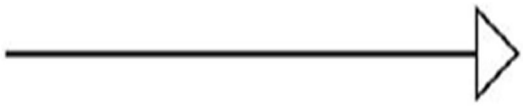
An **association** is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.



3. G**eneralization:**

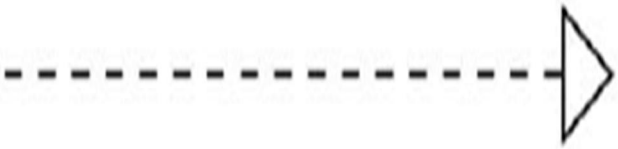
A Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically,

a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



4.R**ealization:**

A R**ealization** 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.



3. Diagrams in the UML:-

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). The UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. State chart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

1.Class diagram:

A class diagram shows a set of classes, interfaces, and collaborations and their relationships.These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system.

2.Object diagram:

An object diagram shows a set of objects and their relationships. 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 .

3.use case diagram:

A use case diagram shows a set of use cases and actors (a special kind of class) 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.

4. **Sequence diagram:**

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; Interaction diagrams address the dynamic view of a system. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

5.collaboration diagram:

A **collaboration diagram** is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Interaction diagrams address the dynamic view of a system. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

6. **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.

7. **Activity diagram**

An activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system.

8. **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.

9. **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 architecture.

→ Rules of the UML:-

Like any language, the UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models. The UML has semantic rules for

- **Names** : What you can call things, relationships, and diagrams
- **Scope** : The context that gives specific meaning to a name
- **Visibility** : How those names can be seen and used by others
- **Integrity** : How things properly and consistently relate to one another
- **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

- Elided Certain elements are hidden to simplify the view
- Incomplete Certain elements may be missing
- 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. They are:

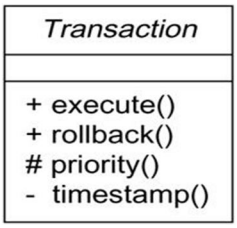
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

1.Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies;

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.



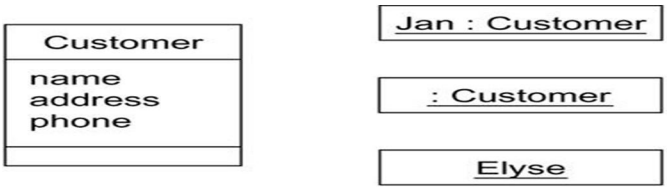
For example, Figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

3.Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

1. **class and object**:

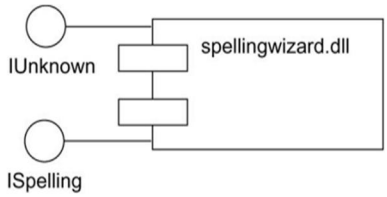
A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure



In this figure, there is one class, named Customer, together with three objects: Jan (which ismarked explicitly as being a Customer object), :Customer (an anonymous Customer object),and Elyse (which in its specification is marked as being a kind of Customer object, although it‘snot shown explicitly here).Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

2**. interface and implementation**.:

An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure



In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and Ispelling.

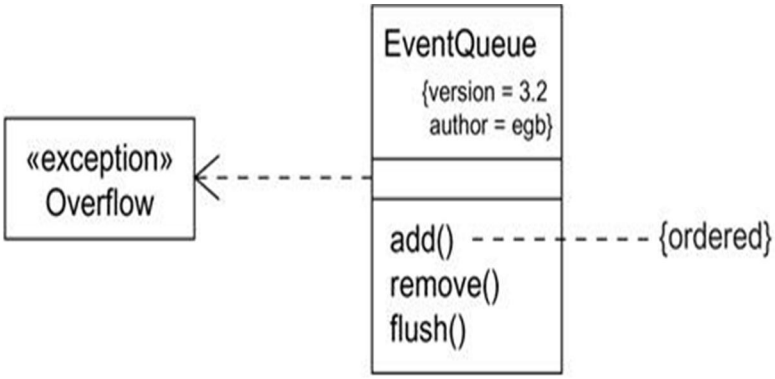
4.Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

1. Stereotypes

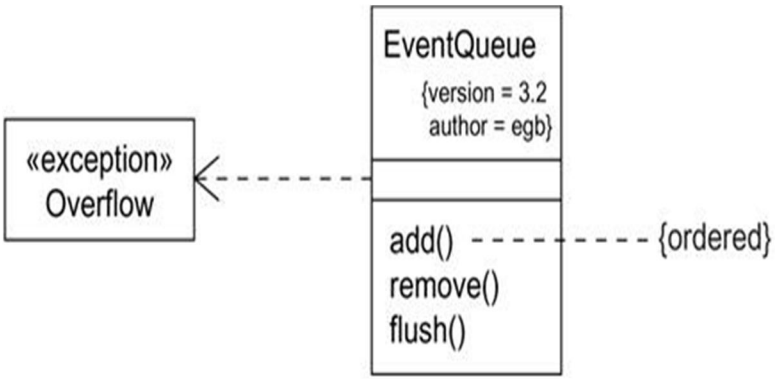
A *stereotype extends the vocabulary of the UML, allowing you to create new kinds of building* blocks that are derived from existing ones but that are specific to your problem.For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions.In these languages, exceptions are just classes, You can make exceptions first class citizens in your models, meaning that they are treated like basic building blocks, by marking them with an appropriate stereotype, as for the class Overflow in Figure.



2. Tagged values

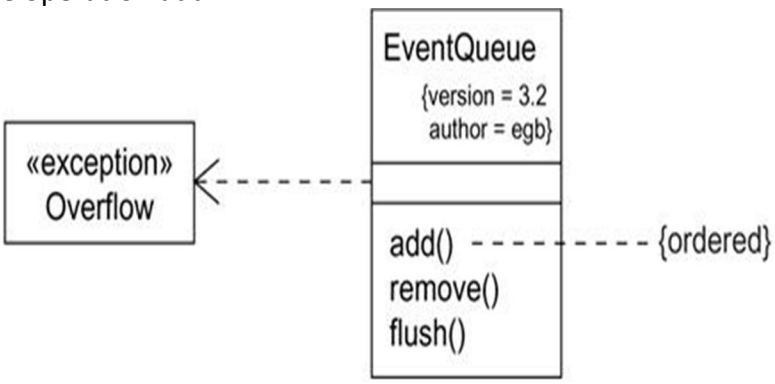
A *tagged value extends the properties of a UML building block, allowing you to create new* information in that element's specification. For example, if you want to specify the version and author of certain critical abstractions. Version and author are not primitive UML concepts.

- For example, the class EventQueue is extended by marking its version and author explicitly.



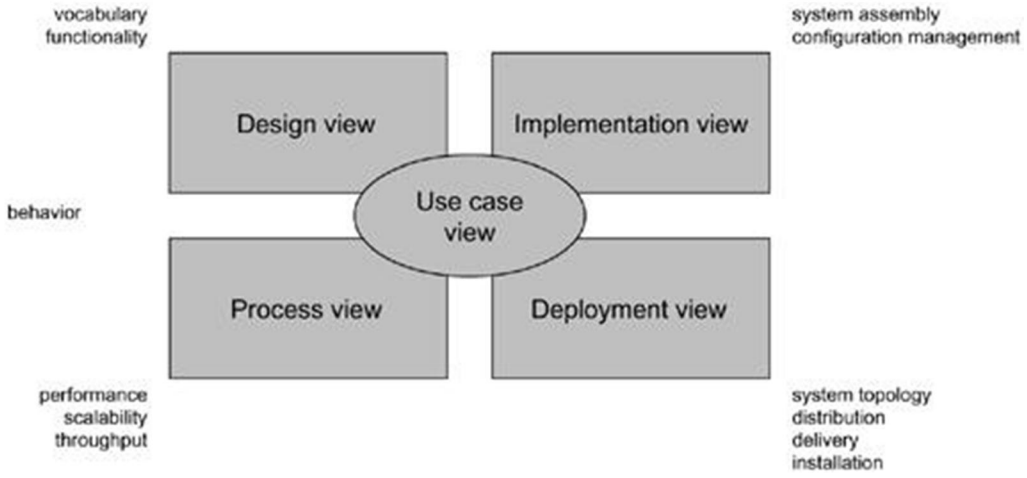
3. Constraints

A *constraint extends the semantics of a UML building block, allowing you to add new rules or* modify existing ones. For example, you mght want to constrain the EventQueue class so that all additions are done in order. As Figure shows, you can add a constraint that explicitly marks these for the operation add.



→ Architecture:-

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints. Below Figure shows, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.



1. *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. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagram.

2. 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. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagram.

3.U*se 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, statechart diagrams, and activity diagrams.

4. 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.With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

5. *Deployment view:*

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.With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

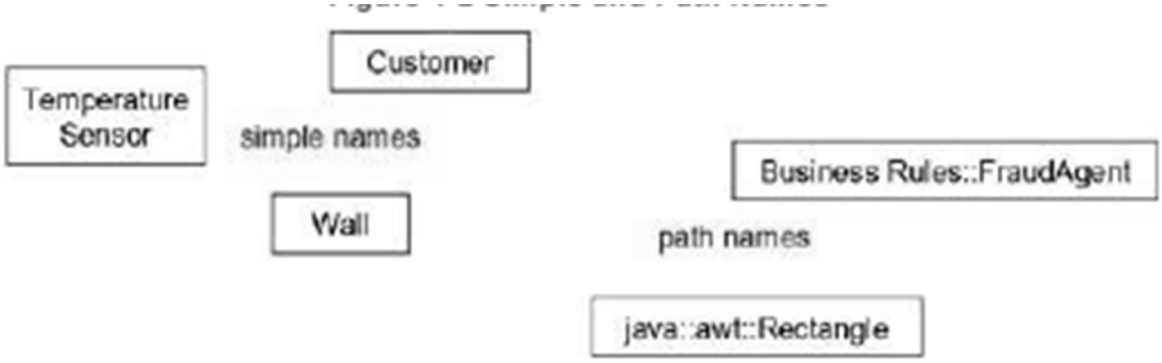
⟶ Classes:-

Terms and Concepts

A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle

Names:

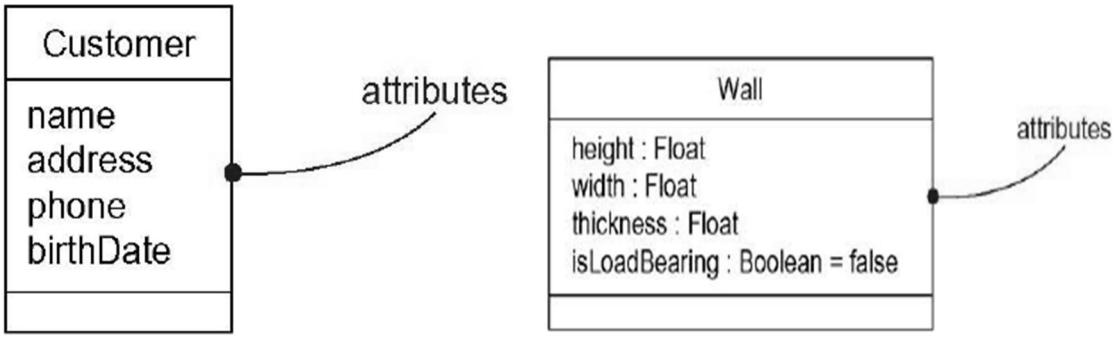
Every class must have a name that distinguishes it from other classes.. That name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as shown in Figure.



Class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines.

Attributes:

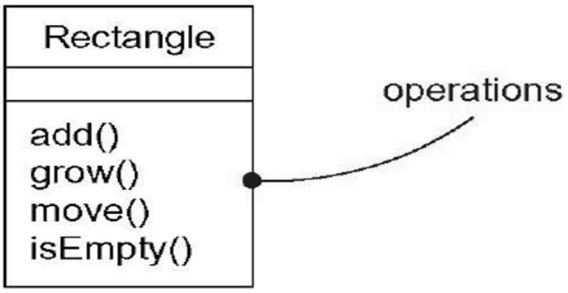
An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing we are modeling that is shared by all objects of that class.



- For example, every wall has a height, width, and thickness; we might model our customers in such a way that each has a name, address, phone number, and date of birth.
- Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names.

Operations

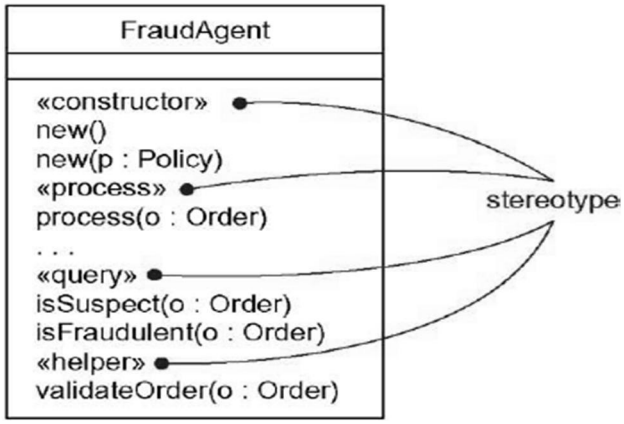
An operation is the implementation of a service that can be requested from any object of the class to affect behavior. A class may have any number of operations or no operations at all.



For example, all objects of the class Rectangle can be moved, resized, or queried for their properties. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure.

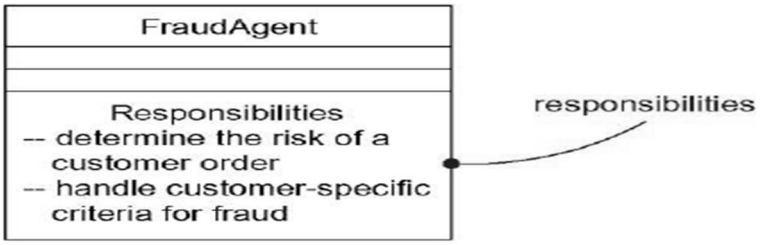
Organizing Attributes and Operations

To better organize long lists of attributes and operations, we can also prefix each group with a descriptive category by using stereotypes, as shown in Figure



Responsibilities

A responsibility is a contract or an obligation of a class. When we create a class, we are making a statement that all objects of that class have the same kind of state and the same kind of behavior. Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure.



Other Features

- Attributes, operations, and responsibilities are the most common features you'll need when we create abstractions. In fact, for most models we build, the basic form of these three features will be all we need to convey the most important semantics of your classes.
- Sometimes, we'll need to visualize or specify other features, such as the visibility of individual attributes and operations; language-specific features of an operation, such as whether it is polymorphic or constant; or even the exceptions that objects of the class might produce or handle.
- These and many other features can be expressed in the UML, but they are treated as advanced concepts.
- Finally, classes rarely stand alone. Rather, when we build models, we will typically focus on groups of classes that interact with one another.
- In the UML, these societies of classes form collaborations and are usually visualized in class diagrams.

Common Modeling Techniques of classes:

1.Modeling the Vocabulary of a System

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

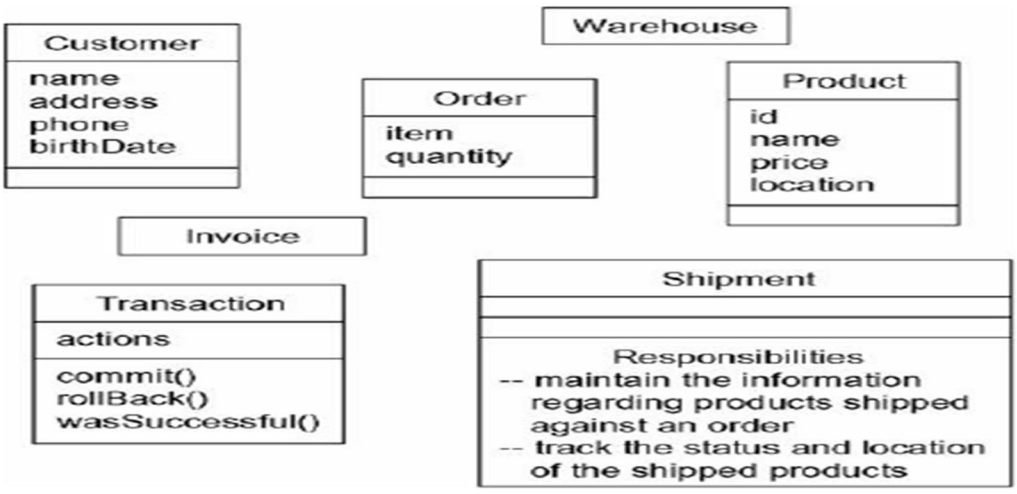
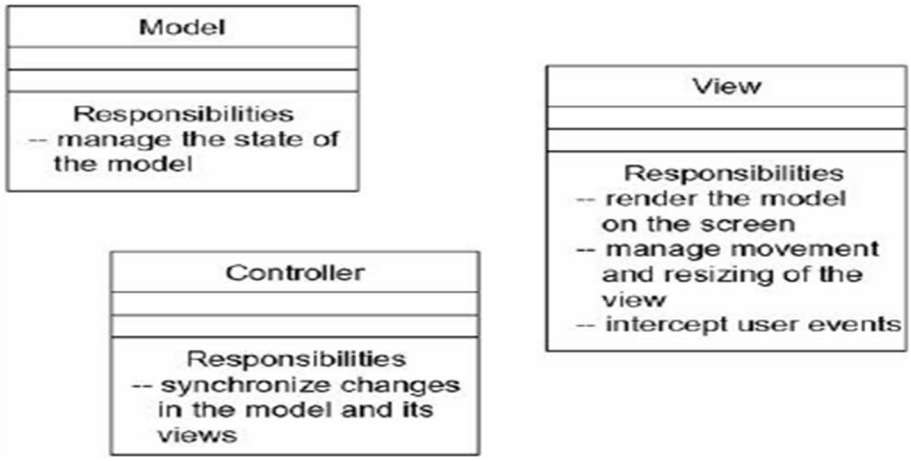


Figure shows a set of classes drawn from a retail system, including Customer, Order, and Product. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, Transaction, which applies to orders and shipments.

2. Modeling the Distribution of Responsibilities in a System

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.

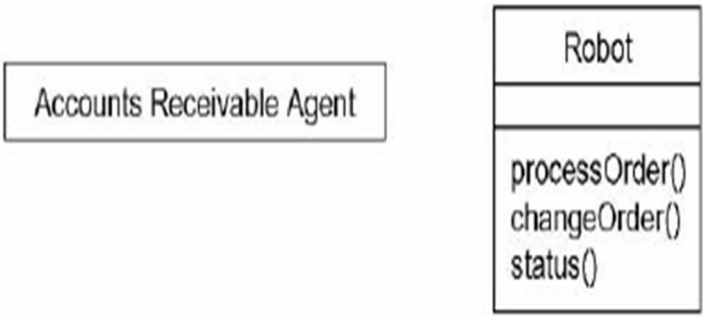


For example, Figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among Model, View, and Controller classes. Notice how all these classes work together such that no one class does too much or too little.

3. Modeling Nonsoftware Things

To model nonsoftware things,

- Model the thing we are abstracting as a class.
- If we 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 we are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that we can further expand on its structure.

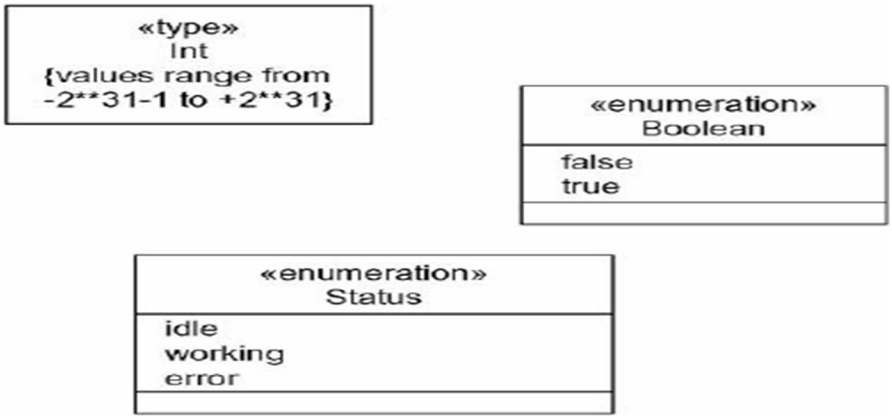


As Figure shows, it's perfectly normal to abstract humans (like AccountsReceivableAgent) and hardware (like Robot) as classes, because each represents a set of objects with a common structure and a common behavior.

4.Modeling Primitive Types

To model primitive types,

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



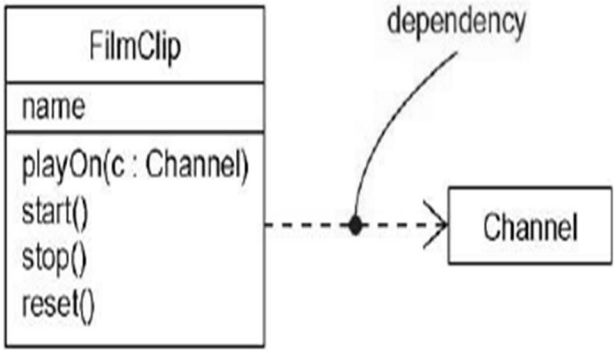
As Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class Int) are modeled as types, and we can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as Boolean and Status, can be modeled as enumerations, with their individual values provided as attributes.

⟶ Relationships:

A ***relationship*** is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

1.Dependency

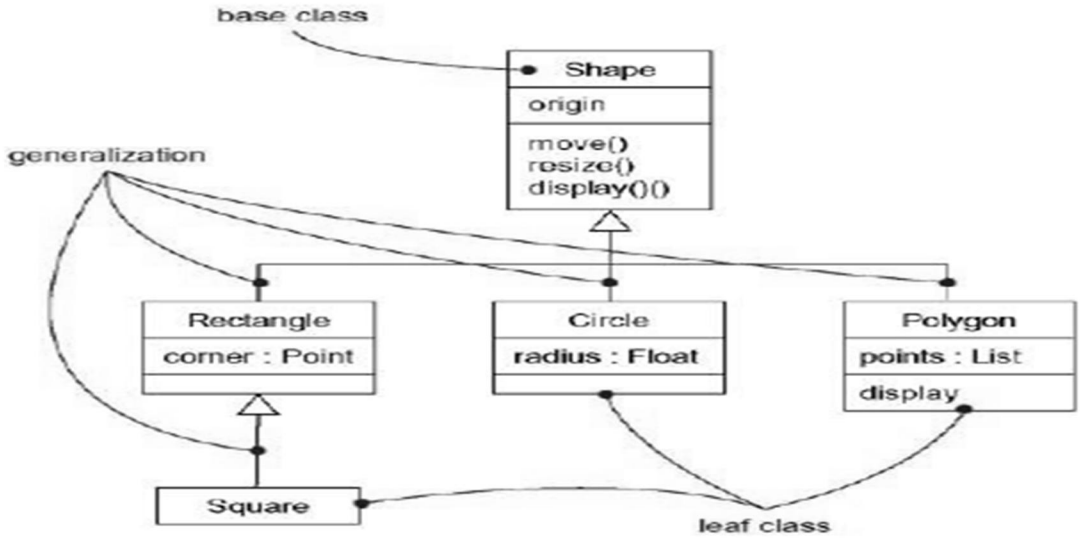
A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



2.Generalization

A Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

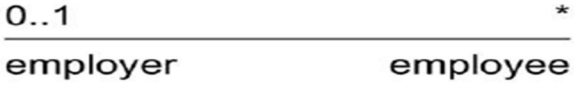
Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window).



- A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

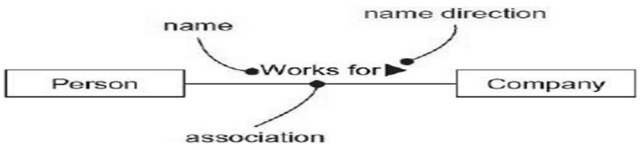
3.Association

An **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.



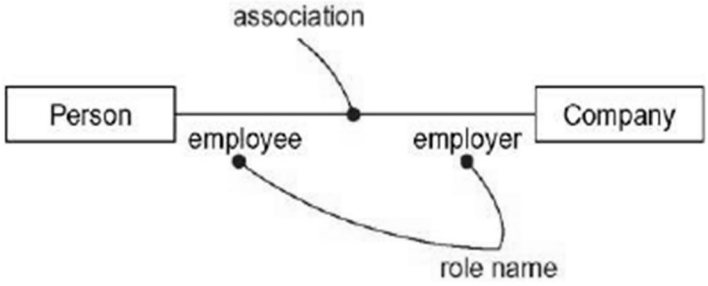
Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that point in the direction you intend to read the name, as shown in Figure.



Role

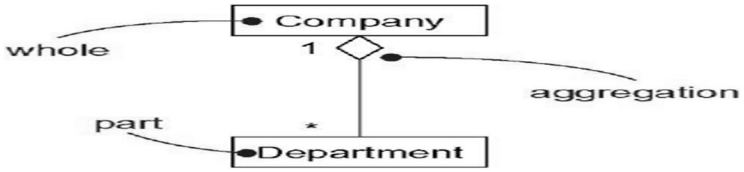
When a class participates in an association, it has a specific role that it plays in that relationship;
A role is just the face the class at the near end of the association presents to the class at the other end of the association.



In Figure, a Person playing the role of employee is associated with a Company playing the role of employer.An instance of an association is called a link.The same class can play the same or different roles in other associations

Aggregation

Aggregation is a special kind of association, representing a structural relationship between a whole and its parts.



Other Features

- Plain, unadorned dependencies, generalizations, and associations with names, multiplicities, and roles are the most common features you'll need when creating

abstractions. In fact, for most of the models you build, the basic form of these three relationships will be all you need to convey the most important semantics of your relationships.

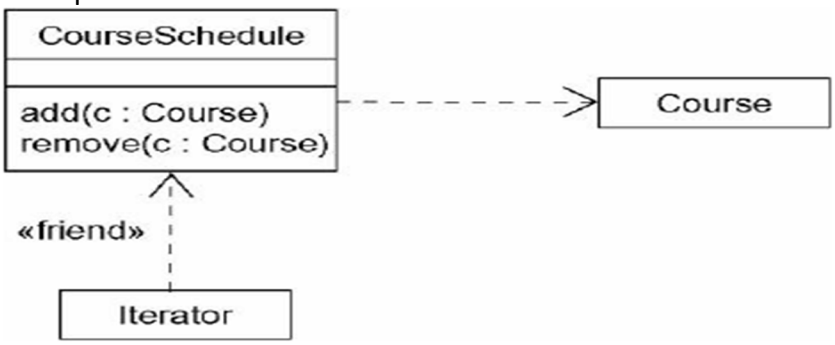
- Sometimes, you'll need to visualize or specify other features, such as composite aggregation, navigation, discriminates, association classes, and special kinds of dependencies and generalizations.
- These and many other features can be expressed in the UML, but they are treated as advanced concepts.
- Dependencies, generalization, and associations are all static things defined at the level of classes.
- In the UML, these relationships are usually visualized in class diagrams.
- When you start modeling at the object level, and especially when you start working with dynamic collaborations of these objects, you'll encounter two other kinds of relationships, links (which are instances of associations representing connections among objects across which messages may be sent) and transitions (which are connections among states in a state machine).

Common Modeling Techniques of classes:

Modeling Simple Dependencies

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.



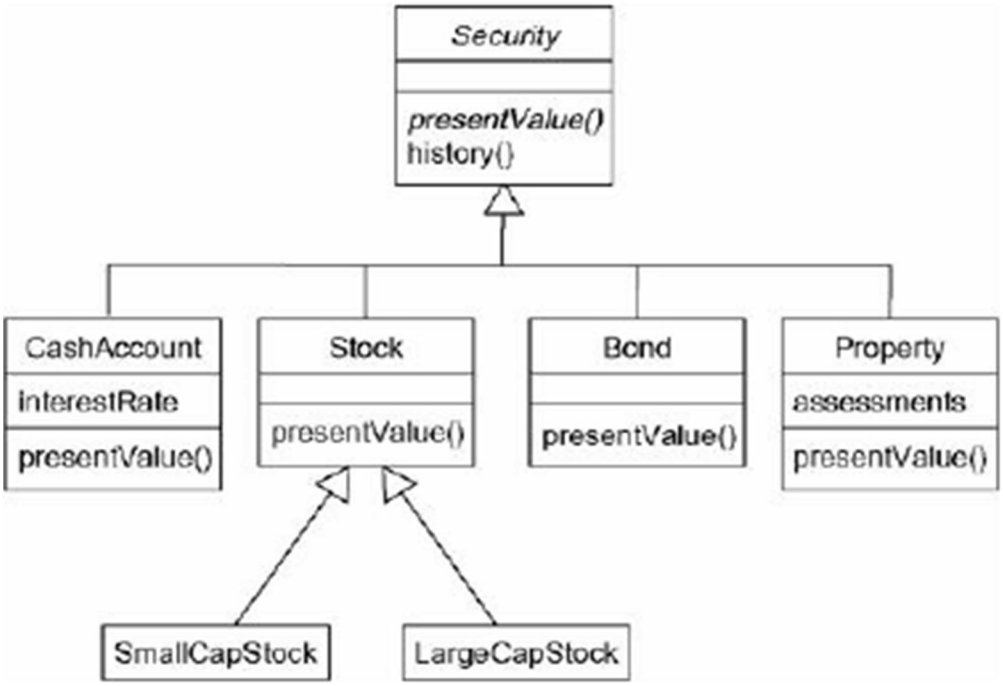
For example, Figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.

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++.

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 elements (but be careful about introducing too many levels).
- 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.



For example, Figure shows a set of classes drawn from a trading application. You will find a generalization relationship from four classes• CashAccount, Stock, Bond, and Property, to the more-general class named Security. Security is the parent, and CashAccount, Stock, Bond, and Property are all children. Each of these specialized children is a kind of Security. You'll notice that Security includes two operations: presentValue and history. Because Security is their parent, CashAccount, Stock, Bond, and Property all inherit these two operations, and for that matter, any other attributes and operations of Security that may be elided in this figure.

You can also create classes that have more than one parent. This is called multiple inheritance and means that the given class has all the attributes, operations, and associations of all its parents.

Modeling Structural Relationships:

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.

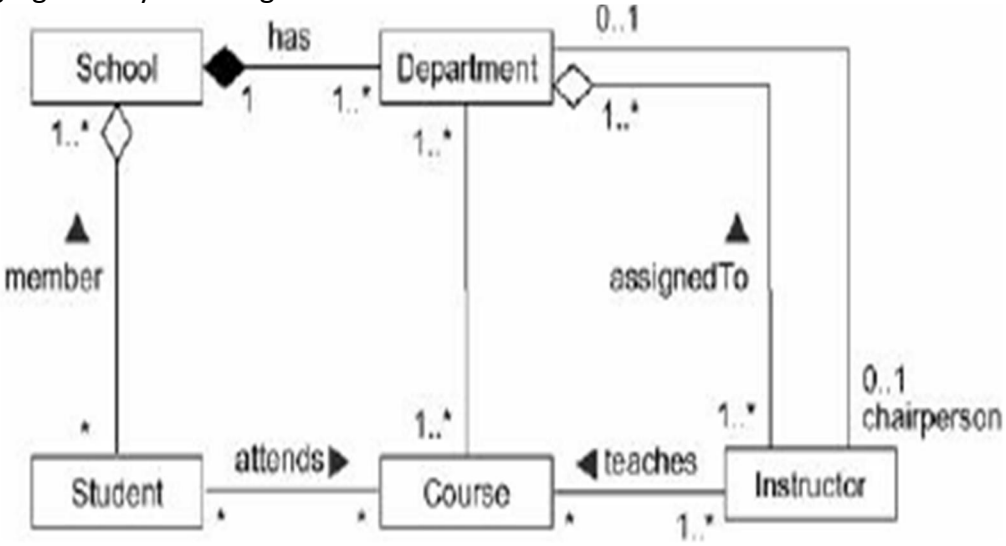


Figure shows a set of classes drawn from an information system for a school The aggregation relationship between School and Department is composite aggregation, The relationships between School and the classes Student and Department are a bit different. Here you'll see aggregation relationships. A school has zero or more students, each student may be a registered member of one or more schools, a school has one or more departments, each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that School is a whole and that Student and Department are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from their school.

You'll also see that there are two associations between Department and Instructor. One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally, departments are at a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

→ Common Mechanisms

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

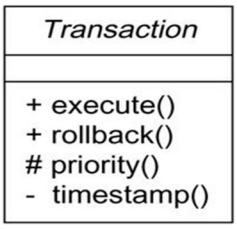
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

1.Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies;

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.



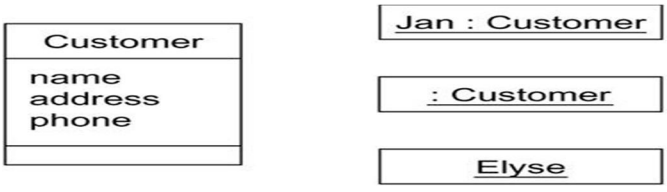
For example, Figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

3.Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

1. **class and object**:

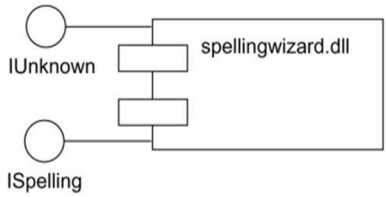
A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure



In this figure, there is one class, named Customer, together with three objects: Jan (which ismarked explicitly as being a Customer object), :Customer (an anonymous Customer object),and Elyse (which in its specification is marked as being a kind of Customer object, although it'snot shown explicitly here).Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

2**. interface and implementation**.:

An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure



In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and Ispelling.

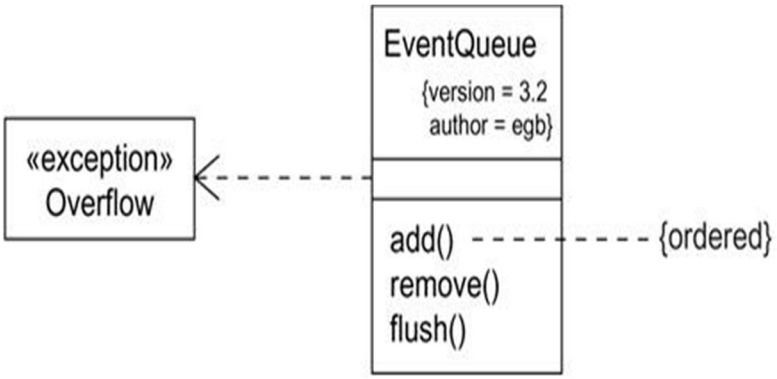
4.Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

1. Stereotypes

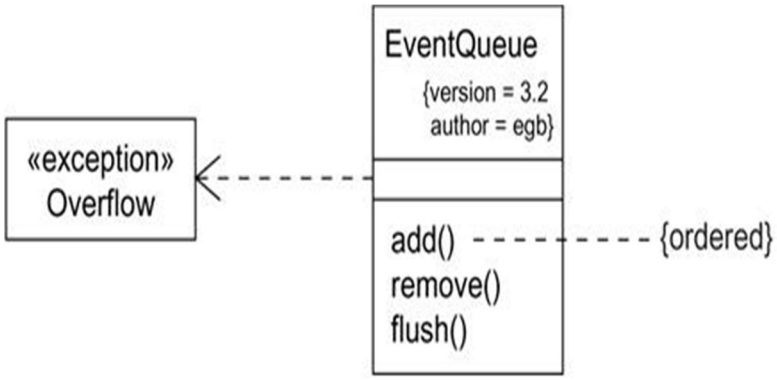
A *stereotype extends the vocabulary of the UML, allowing you to create new kinds of building* blocks that are derived from existing ones but that are specific to your problem.For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions.In these languages, exceptions are just classes, You can make exceptions first class citizens in your models, meaning that they are treated like basic building blocks, by marking them with an appropriate stereotype, as for the class Overflow in Figure.



2. Tagged values

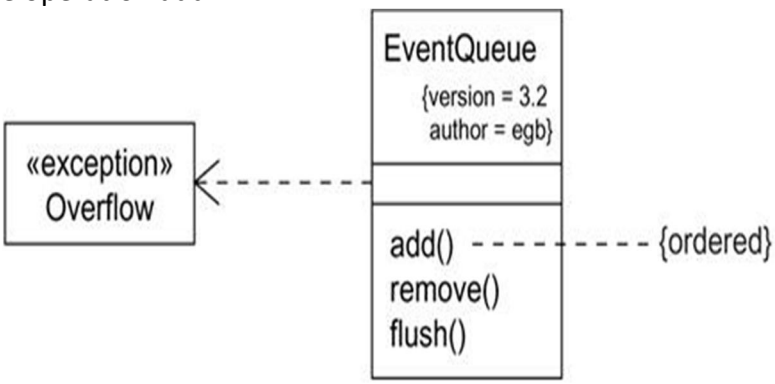
A *tagged value extends the properties of a UML building block, allowing you to create new* information in that element's specification. For example, if you want to specify the version and author of certain critical abstractions. Version and author are not primitive UML concepts.

- For example, the class EventQueue is extended by marking its version and author explicitly.



3. Constraints

A *constraint extends the semantics of a UML building block, allowing you to add new rules or* modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As Figure shows, you can add a constraint that explicitly marks these for the operation add.



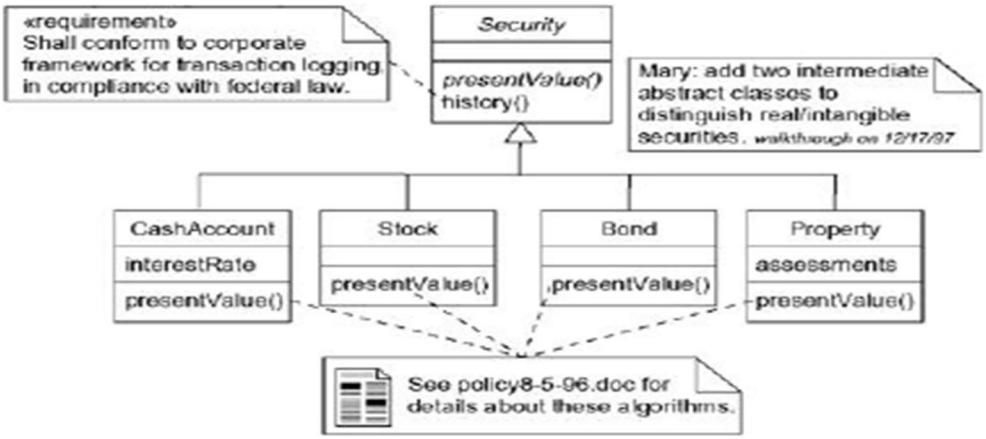
Common Modeling Techniques of common mechanisms

Modeling Comments

To model a comment,

- Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.

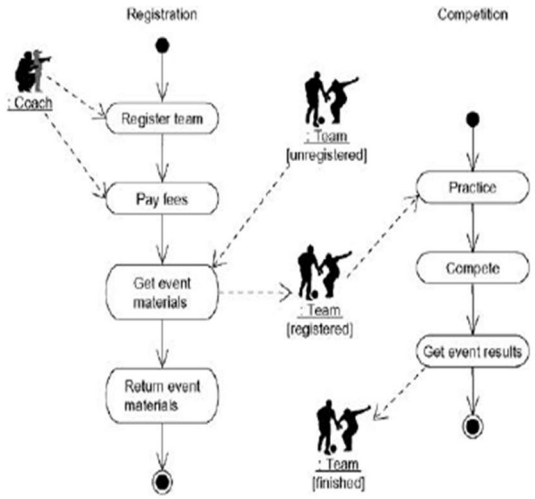
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and unless they are of historic interest discard the others.



Modeling New Building Blocks .

To model new building blocks,

- Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are there's already some standard stereotype that will do what you want.
- If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing. Remember that you can define hierarchies of stereotypes so that you can have general kinds of stereotypes along with their specializations (but as with any hierarchy, use this sparingly).
- Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.

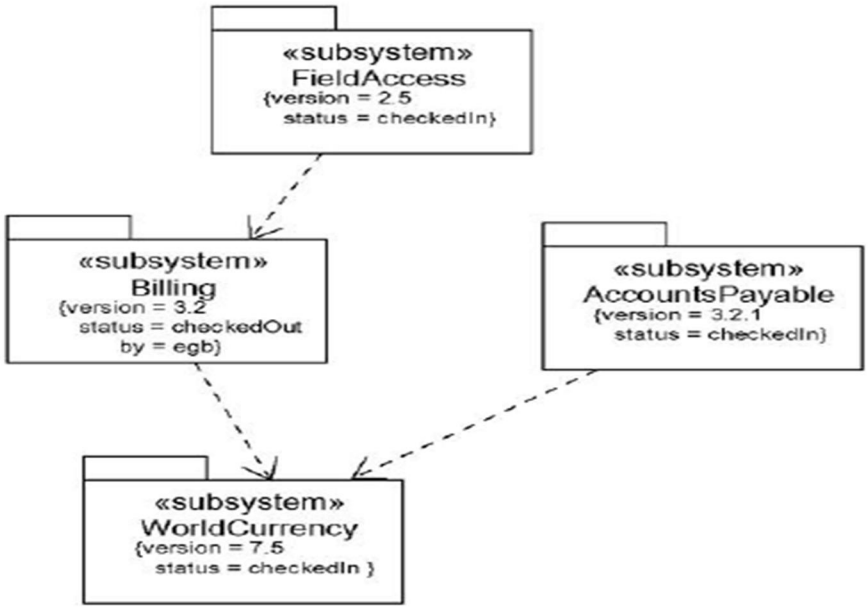


For example, suppose you are using activity diagrams to model a business process involving the flow of coaches and teams through a sporting event. In this context, it would make sense to visually distinguish coaches and teams from one another and from the other things in this domain, such as events and divisions.

Modeling New Properties

To model new properties,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
- If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply•tagged values defined for one kind of element apply to its children.



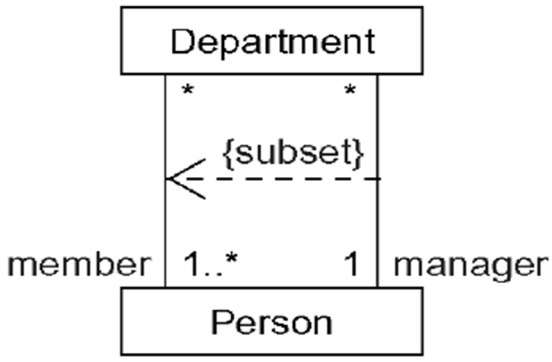
For example, suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.

Figure shows four subsystems, each of which has been extended to include its version number and status. In the case of the Billing subsystem, one other tagged value is shown the person who has currently checked out the subsystem.

Modeling New Semantics

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already somestandard constraint that will do what you want.
- If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



For example, Figure models a small part of a corporate human resources system This diagram shows that each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member. This diagram goes on to indicate that each Department must have exactly one Person as a manager and every Person may be the manager of zero or more Departments. All of these semantics can be expressed using simple UML. However, to assert that a manager must also be a member of the department is something that cuts across multiple associations and cannot be expressed using simple UML. To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the Department, connecting the two associations and the constraint by a dependency from the subset to the superset.

→ Class Diagrams :

Class diagrams are the most common diagram found in modeling object- oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. You use class diagrams to model the static design view of a system.

Terms and Concepts :

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

Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams, a name and graphical content that are a projection into a model. What distinguishes a class diagram from all other kinds of diagrams is its particular content.

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, respectively.

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• the services the system should provide to its end users. When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. **To model 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. You use class diagrams to specify these abstractions and their responsibilities.

2. **To model 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. For example, when you're modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. **To model a logical database schema**

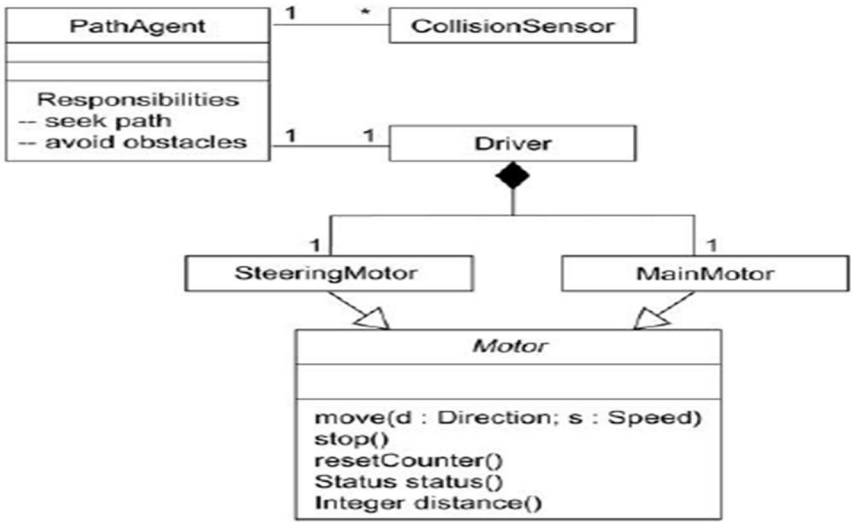
Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

Common Modeling Techniques of class diagrams:

1.Modeling Simple Collaborations

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.



For example, Figure shows a set of classes drawn from the implementation of an autonomous robot. The figure focuses on the classes involved in the mechanism for moving the robot along a path. You'll find one abstract class (Motor) with two concrete children, SteeringMotor and MainMotor. Both of these classes inherit the five operations of their parent, Motor. The two classes are, in turn, shown as parts of another class, Driver. The class PathAgent has a one-to-one association to Driver and a one-to-many association to CollisionSensor. No attributes or operations are shown for PathAgent, although its responsibilities are given.

Modeling a Logical Database Schema

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.

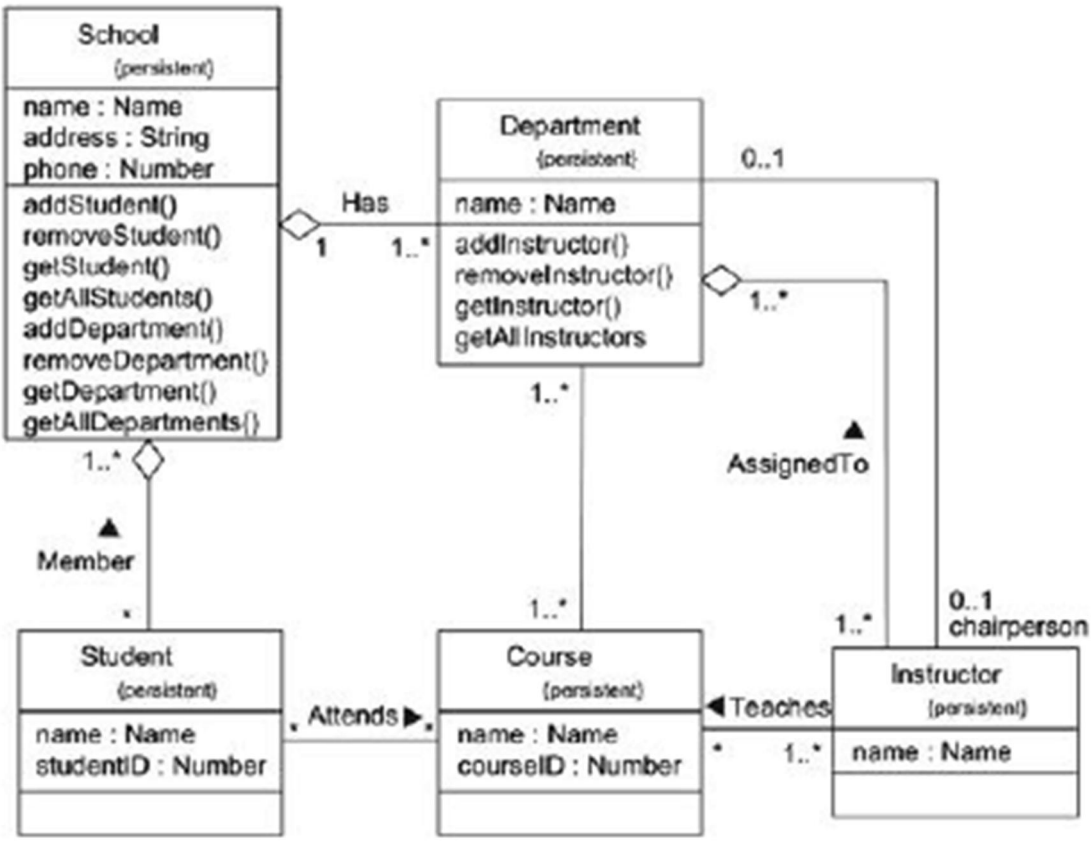


Figure shows a set of classes drawn from an information system for a school. This figure expands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database. Starting at the bottom-left of this diagram, you will find the classes named Student, Course, and Instructor. There's an association between

Student and Course, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students.

All of these classes are marked as persistent, indicating that their instances are intended to live in a database or some other form of persistent store. This diagram also exposes the attributes of all six of these classes. Notice that all the attributes are primitive types. When you are modeling a schema, you'll generally want to model the relationship to any nonprimitive types using an explicit aggregation rather than an attribute.

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.

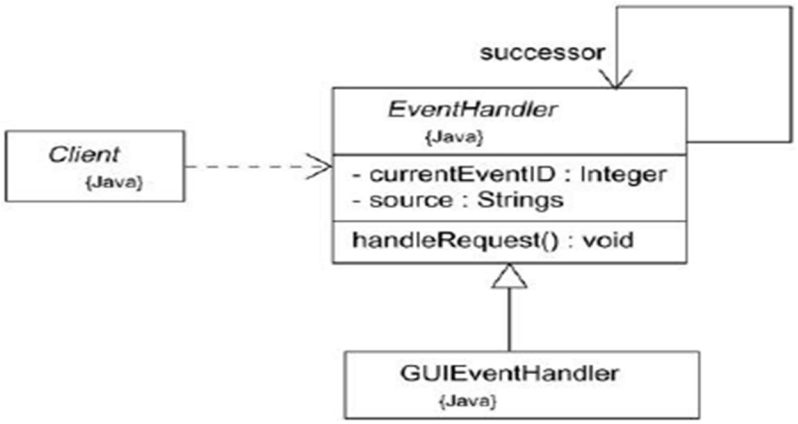


Figure illustrates a simple class diagram specifying an instantiation of the chain of responsibility pattern. This particular instantiation involves three classes: Client, EventHandler,

and GUIEventHandler. Client and EventHandler are shown as abstract classes, whereas GUIEventHandler is concrete. EventHandler has the usual operation expected of this pattern (handleRequest), although two private attributes have been added for this instantiation.

All of these classes specify a mapping to Java, as noted in their tagged value. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class EventHandler yields the following code.

```
public abstract class EventHandler {

  EventHandler successor;
  private Integer currentEventID;
  private String source;

  EventHandler() {}
  public void handleRequest() {}
 }
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific 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 Diagrams:

Terms and Concepts

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.

Common Properties

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. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

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, but from the perspective of real or prototypical instances. This view primarily supports the functional requirements of a system• that is, the services the system should provide to its end users. Object diagrams let you model static data structures.

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

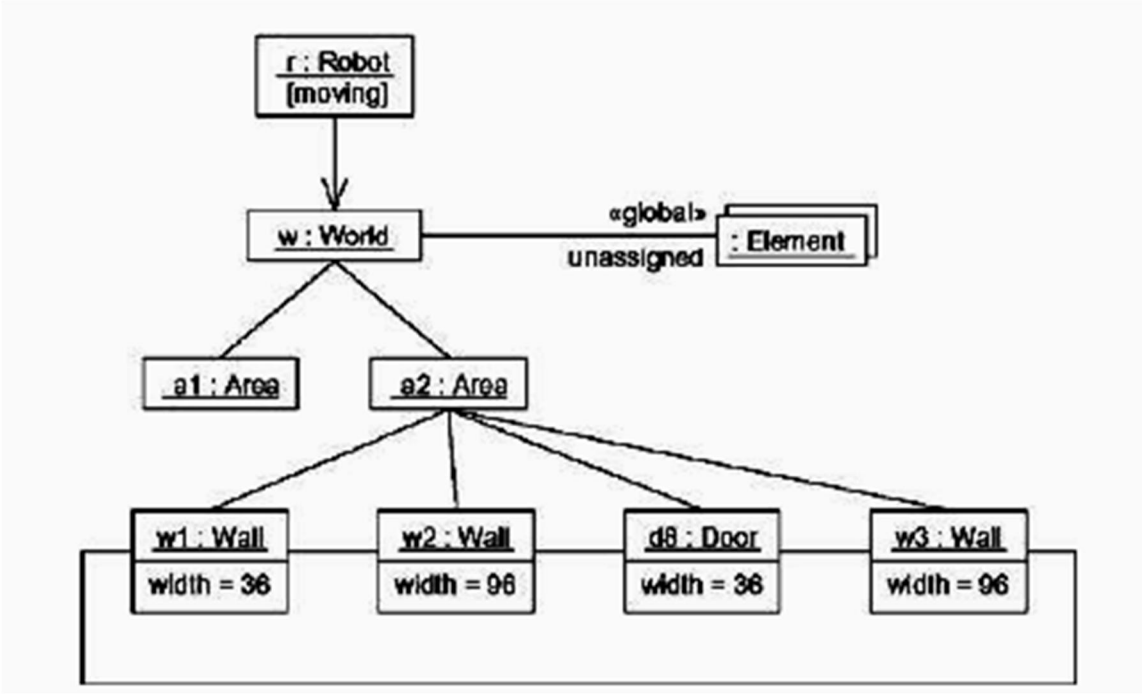
Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another.

Common Modeling Techniques of object diagrams:

Modeling Object Structures

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.



For example, Figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

As this figure indicates, one object represents the robot itself (r, an instance of Robot), and r is currently in the state marked moving. This object has a link to w, an instance of World, which represents an abstraction of the robot's world model. This object has a link to a multiobject that consists of instances of Element, which represent entities that the robot has identified but not yet assigned in its world view. These elements are marked as part of the robot's global state.

At this moment in time, w is linked to two instances of Area. One of them (a2) is shown with its own links to three Wall and one Door object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) 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.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

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.

Short Questions

1. What does Unified mean?
2. List the scope of UML.
3. List the types of relationships.
4. Give graphical representation diagram for dependencies, associations, generalizations and realizations.
5. Write about Association.
6. Differentiate static models and dynamic models.
7. Write about importance of Public, Private and Protected in class.
8. What is an interface?
9. What is the need for modeling?
10. Compare different class diagrams.

Essay Questions

1. Explain the modeling of system's architecture with diagram.

(or)

Demonstrate the working of UML architecture with neat diagram.

(or)

Describe about five interlocking views involved in the architecture of a software-intensive system modeling

2. Explain about common mechanisms in the UML.
3. Explain the Conceptual model of UML in detail.

4. What is the UML approach to software development life cycle? Briefly explain various building blocks of UML.

5.List out the common properties of object and class diagram and demonstrate the classes and objects in bank management system with neat sketch.

❖ Package Diagrams

While performing object-oriented analysis and design, we need to organize the artifacts of the development process to clearly present the analysis of the problem space and the associated design. The benefits of organizing the OOAD artifacts include the following

- Provides clarity and understanding in a complex systems development
- Supports concurrent model use by multiple users
- Supports version control
- Provides abstraction at multiple levels—from systems to classes in a component
- Provides encapsulation and containment; supports modularity

Essentials: The Package Notation

The notation for the package is a rectangle with a tab on the top left. UML 2.0 specifies that the name of the package is placed in the interior of the rectangle if the package contains no UML elements. If it does contain elements, the name should be placed within the tab.

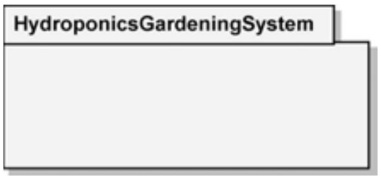
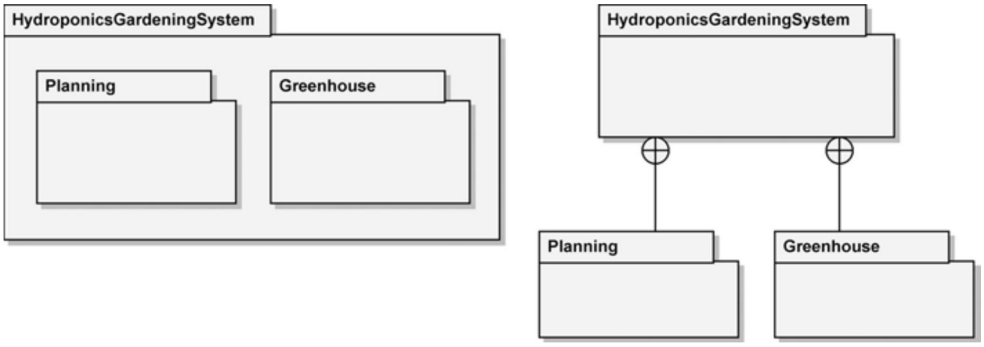


Figure again shows the HydroponicsGardeningSystem package, but with two of its contained elements represented as packages themselves. In the representation on the left, we show the Planning and Greenhouse packages as physically contained packages inside the HydroponicsGardening-System package. On the right appears an alternate notation for the containment relationship.

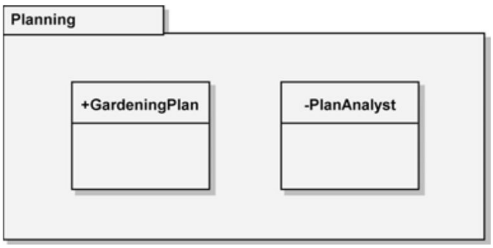


Essentials: Visibility of Elements

Access to the services provided by a group of collaborating classes within a package or more generically, to any elements within a package is determined by the visibility of the individual elements, including nested packages. The visibility of the elements, defined by the containing package to be either public or private.

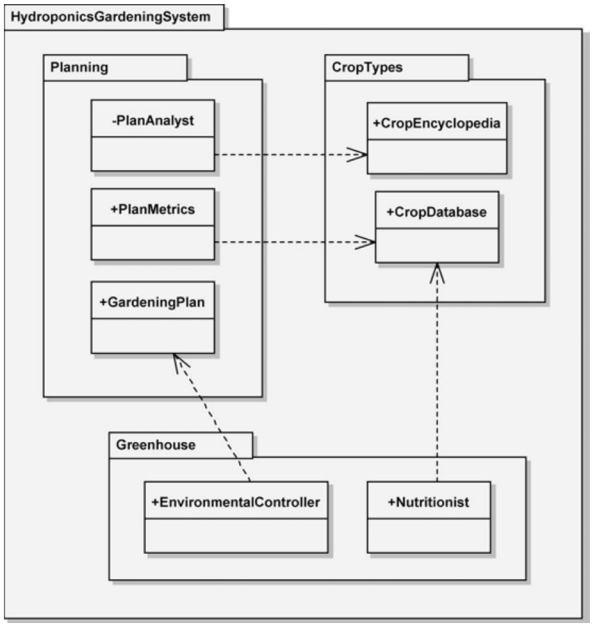
■ Public (+) Visible to elements within its containing package, including nested packages, and to external elements
■ Private (-) Visible only to elements within its containing package and to nested packages

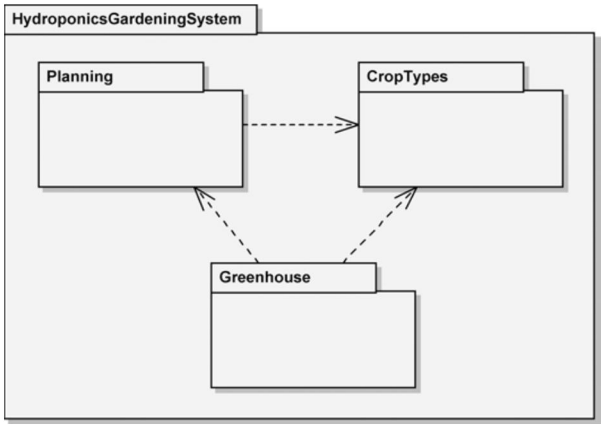
On a visual diagram, this visibility notation is placed in front of the element name, as shown in Figure. The GardeningPlan class has public visibility to permit other elements to access it, while the PlanAnalyst class has private visibility.



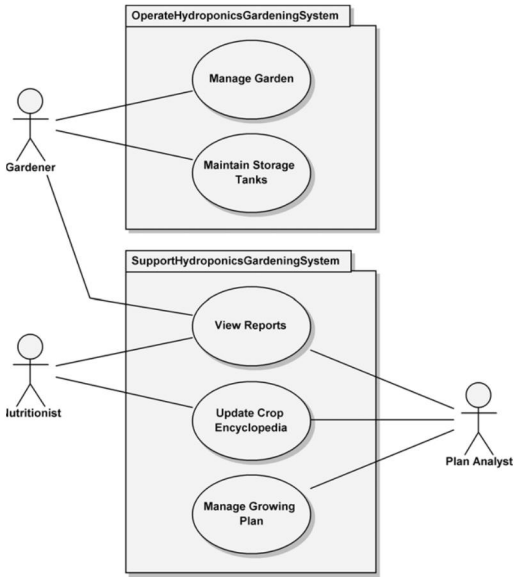
Essentials: The Dependency Relationship

A dependency shows that an element is dependent on another element as it fulfills its responsibilities within the system. Dependencies between UML elements (including packages), as shown in Figure, are represented as a dashed arrow with an open arrowhead. The tail of the arrow is located at the element having the dependency (*client*), and the arrowhead is located at the element that supports the dependency (*supplier*). If multiple contained element dependencies exist between packages, these dependencies are aggregated at the package level.





Essentials: Package Diagrams



The package diagram is the UML 2.0 structure diagram that contains packages as the primary represented UML element and shows dependencies between the packages.

However, the package notation can be used to show the structuring and containment of many different model elements, such as classes, as shown earlier in Figures. It can also be used on UML diagrams that are not structure
diagrams. We alluded to this earlier when we mentioned that a package can be used to organize use cases. An example appears in Figure, where packages are used to group use cases of the HydroponicsGardeningSystem to
facilitate their specification among two groups with different expertise operations and support.

Advanced Concepts: Import and Access

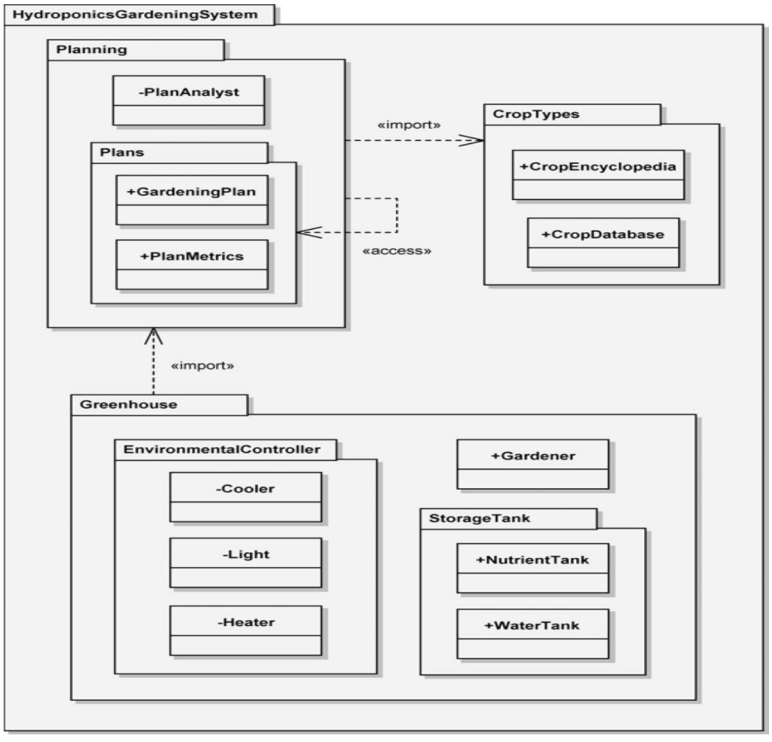
Import is a public package import, whereas access is a private package import. What this really means is that in an import, other elements that have visibility into the importing package can see the imported items. But, when a package performs an access, no other elements can see those elements that have been added to the importing package's namespace. These items are private; they are not visible outside the package that performed the access.

Doing so gives us the ability to refer to the public elements of another namespace by using unqualified names; the importing package adds the names of the imported elements to its namespace. However, if any of the imported elements are of the same type and have the same name as an owned element, they are not added to the importing namespace.

Figure 5–8 also shows the Planning package performing a private import of the Plans package, as illustrated by the dependency labeled with «access». This is necessary to allow the PlanAnalyst class to access the GardeningPlan and PlanMetrics classes with unqualified names. But, since an access dependency is private, the Greenhouse package's import of the Planning package doesn't provide the Greenhouse package elements, such as the Gardener
class, with the ability to reference GardeningPlan and PlanMetrics with unqualified names. In addition, the elements of the Greenhouse package can't even see the PlanAnalyst class because it has private visibility. Looking inside the Greenhouse package, the Gardener class must use the qualified names of the elements within the StorageTank package because its namespace does not import the package. For example, it must use the name StorageTank::WaterTank to reference the WaterTank class. Taking this one more step, we look at the elements within the EnvironmentalController package. They all have private visibility. This means they are not visible outside their namespace, that is, the EnvironmentalController package.

To summarize, an unqualified name (often called a *simple name*) is the name of the element without any path information telling us how to locate it within our model. This unqualified name can be used to access the following elements in a package:

- Owned elements
- Imported elements
- Elements within outer packages



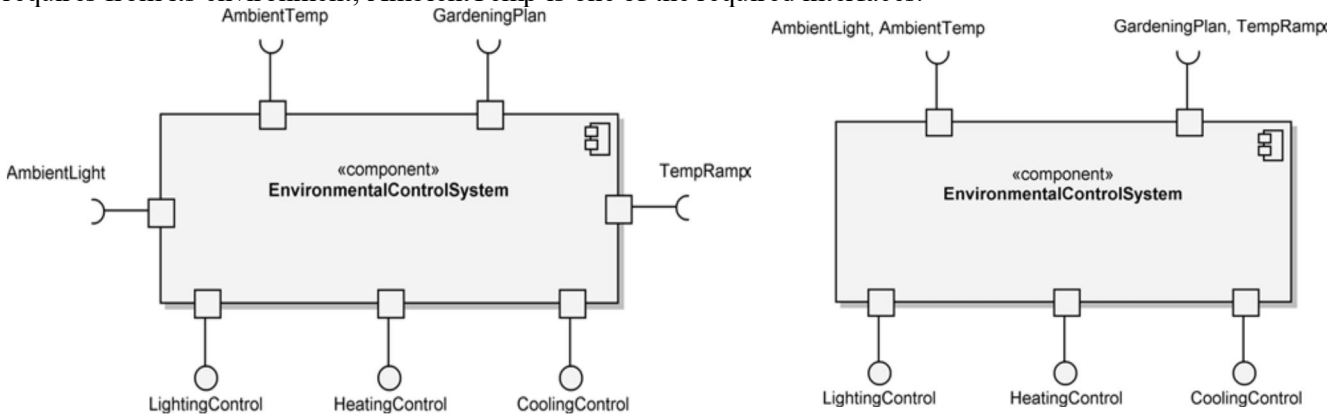
❖ Component Diagrams

A component represents a reusable piece of software that provides some meaningful aggregate of functionality. Components are a type of structured classifier whose collaborations and internal structure can be shown on a component diagram. A component, collaborating with other components through well-defined interfaces to provide a system's functionality, may itself be comprised of components that collaborate to provide its own functionality. Thus, components may be used to hierarchically decompose a system and represent its logical architecture. This logical perspective of a component is new with UML 2.0.

Essentials: The Component Notation

Figure shows the notation used to represent a component. Its name, EnvironmentalControlSystem in this case, is included within the classifier rectangle in bold lettering, using the specific naming convention defined by the development team. In addition, one or both of the component tags should be included: the keyword label «component» and the component icon shown in the upper right-hand corner of the classifier rectangle. On the boundary of the classifier rectangle, we have seven ports, which are denoted by small squares. Ports have public visibility unless otherwise noted.

Ports are used by the component for its interactions with its environment, and they provide encapsulation to the structured classifier. These seven ports are unnamed but should be named, in the format of *port name* : *Port Type*, when needed for clarity. The port type is optional when naming a port. To the ports shown in Figure, we have connected interfaces, which define the component's interaction details. The interfaces are shown in the ball-and-socket notation. Provided interfaces use the ball notation to specify the functionality that the component will provide to its environment; LightingControl is an example of a provided interface. Required interfaces use the socket notation to specify the services that the component requires from its environment; AmbientTemp is one of the required interfaces.

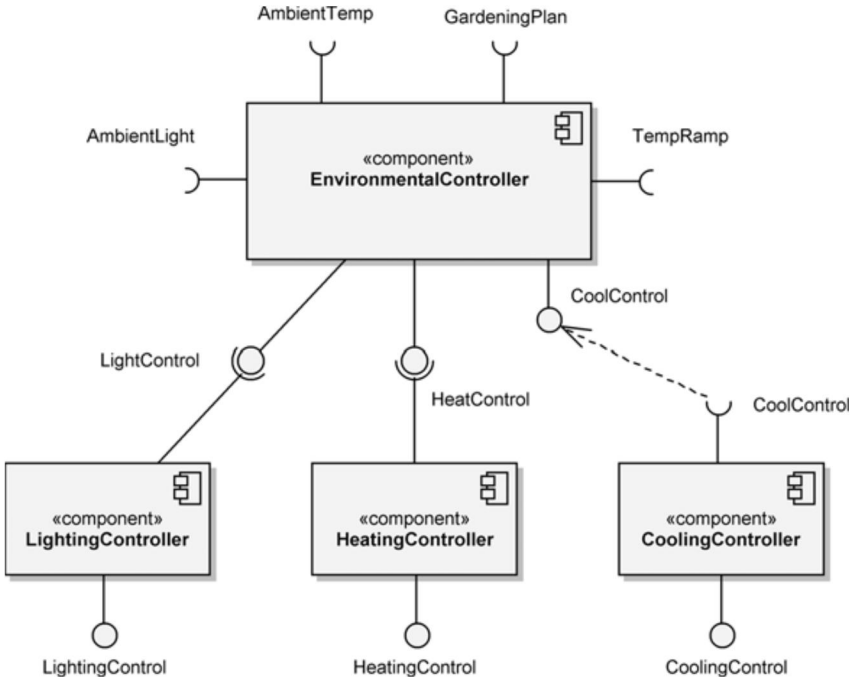


A one-to-one relationship between ports and interfaces is not required; ports can be used to group interfaces, as shown in Figure. This may be done, for example, to provide clarity in a very complex diagram or to represent the intention of having one port through which certain types of interactions will take place. In Figure, the ambient measurements of light and temperature are received at one port. Similarly, the gardening plan and temperature ramp information provided by the staff of the Hydroponics Gardening System are received at a single port.

Essentials: The Component Diagram

During development, we use component diagrams to indicate the logical layering and partitioning of our architecture. In them, we represent the interdependencies of components, that is, their collaborations through well-defined interfaces to provide a system's functionality. Figure shows the component diagram for EnvironmentalControlSystem. As in Figure, the ball-and-socket notation is used to specify the required and provided interfaces of each of the components. The interfaces between the components are

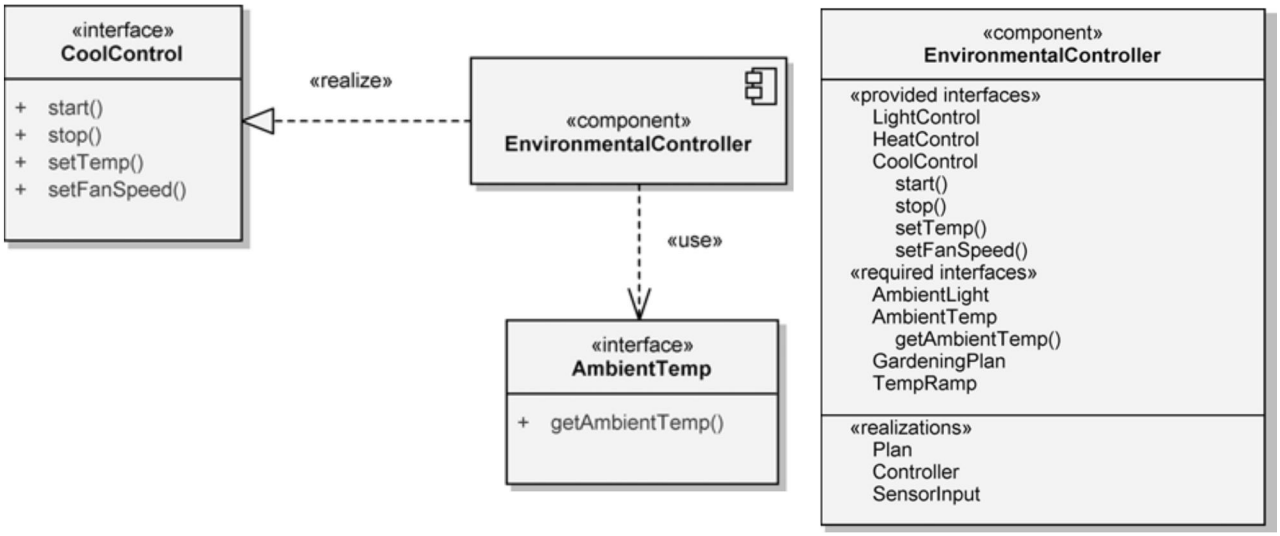
called *assembly connectors*; they are also known as *interface connectors*. The interface between Environmental Controller and CoolingController is shown with a dependency to illustrate another form of representation.



Essentials: Component Interfaces

In our case, the specification focuses on only two of the seven interfaces of EnvironmentalController: CoolControl and AmbientTemp. EnvironmentalController realizes the CoolControl interface; this means that it provides the functionality specified by the interface. This functionality is starting, stopping, setting the temperature, and setting the fan speed for any component using the interface, as shown by the contained operations. These operations
may be further detailed with parameters and return types, if needed. The CoolingController component (shown in Figure) requires the functionality of this interface.

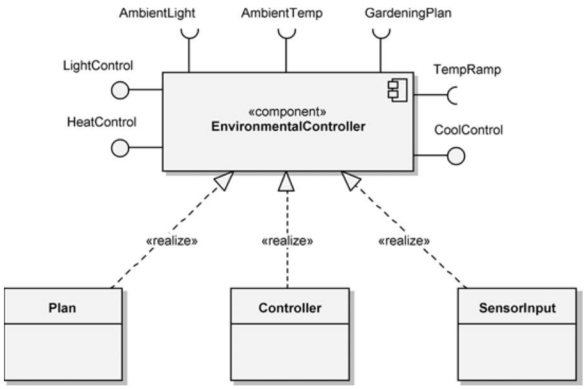
Figure also shows the dependency of the EnvironmentalController component on the AmbientTemp interface. Through this interface, EnvironmentalController acquires the ambient temperature that it requires to fulfill its responsibilities within the EnvironmentalControl System component.

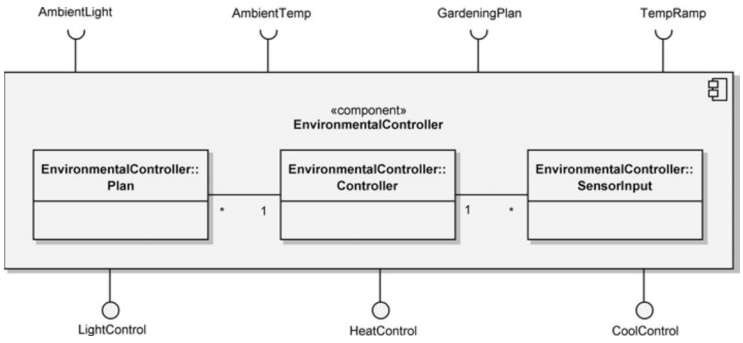


In the above Figure, we show an alternate notation for the interfaces of EnvironmentalController. Here we see the three provided interfaces listed under the heading «provided interfaces». For the CoolControl interface specified in Figure, we have provided the associated operations. Likewise, the required interfaces are shown under the heading «required interfaces», along with three classes listed under the «realizations» heading.

Essentials: Component Realizations

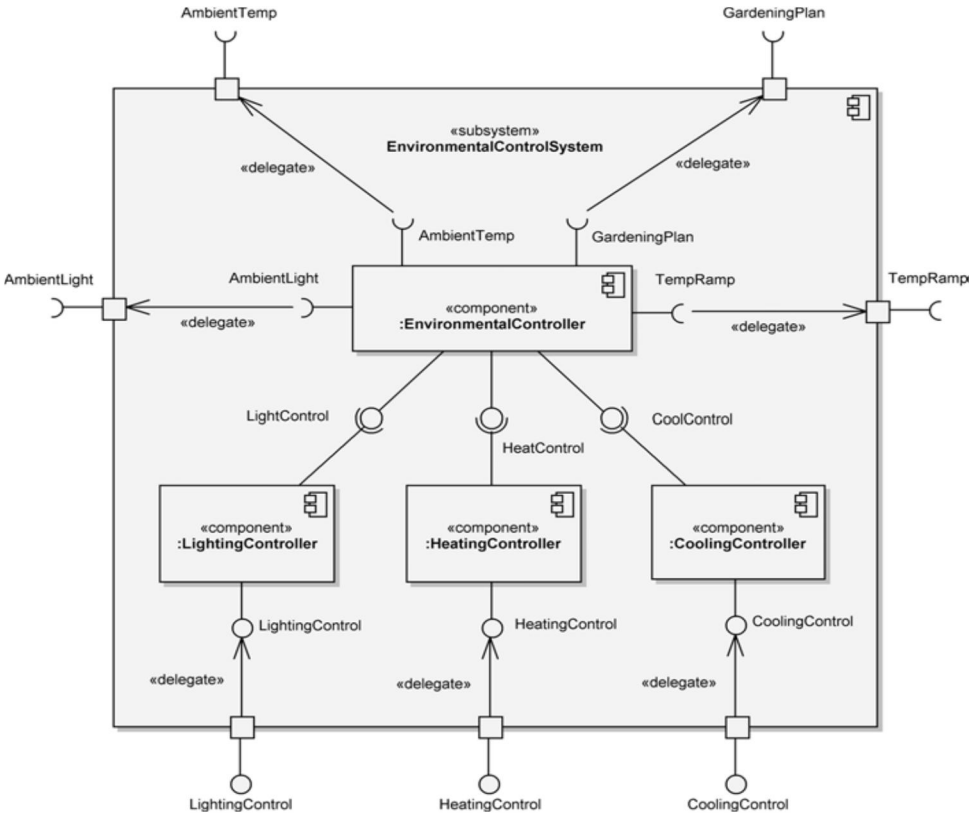
Figure specifies that the EnvironmentalController component is realized by the classes Plan, Controller, and SensorInput. These three classes provide all of the functionality advertised by its provided interfaces. But, in doing so, they require the functionality specified by its required interfaces. This realization relationship between the EnvironmentalController component and the Plan, Controller, and SensorInput classes is shown in Figure. Here, we see a realization dependency from each of the classes to EnvironmentalController. This same information may be represented with a containment relationship, as shown in Figure; each of the classes is physically contained by the EnvironmentalController component. The naming convention used for these internal classifiers is tool-specific. Also, note the associations between the classes and the specification of multiplicity.





Advanced Concepts: A Component's Internal Structure

The internal structure of a component may be shown by using an internal structure diagram; Figure shows just such a diagram for the Environmental ControlSystem subsystem. In addition, the «delegate» label on the lines between the interfaces of the internal components and the ports on the edge of the Environmental ControlSystem. These connections provide the means to show which internal component fulfills the responsibility of the provided interfaces and which internal component needs the services shown in the required interfaces. A subsystem is an aggregate containing other subsystems and other components.



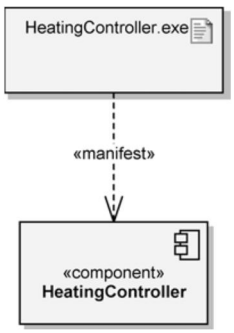
To be specific, :EnvironmentalController requires GardeningPlan, which specifies the environmental needs (lighting, heating, and cooling) of the Hydroponics Gardening System. The needs of this required interface are delegated to an unnamed port, to which is attached the GardeningPlan interface. In this manner, we know that we must provide the EnvironmentalControlSystem component with a gardening plan if we intend to use its services. We also recognize that we must provide it with AmbientLight, AmbientTemp, and TempRamp services. The connectors of EnvironmentalControlSystem provide its communication links to its environment, as well as the means for its parts to communicate internally.

❖ Deployment Diagrams

During development, we use deployment diagrams to indicate the physical collection of nodes that serve as the platform for execution of our system.

Essentials: The Artifact Notation

An artifact is a physical item that implements a portion of the software design. It is typically software code (executable) but could also be a source file, a document, or another item related to the software code. Artifacts may have relationships with other artifacts, such as a dependency or a composition.

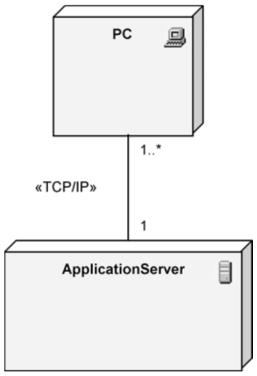


The notation for an artifact consists of a class rectangle containing the name of the artifact, the keyword label «artifact», and an optional icon that looks like a sheet of paper with the top right-hand corner folded over. The name of this artifact includes the extension .exe, indicating that it is an executable (i.e.,

software code). The HeatingController.exe artifact has a dependency relationship to the HeatingController component, labeled with «manifest». This means that it physically implements the component, thereby connecting the implementation to the design. An artifact may manifest more than one component.

Essentials: The Node Notation

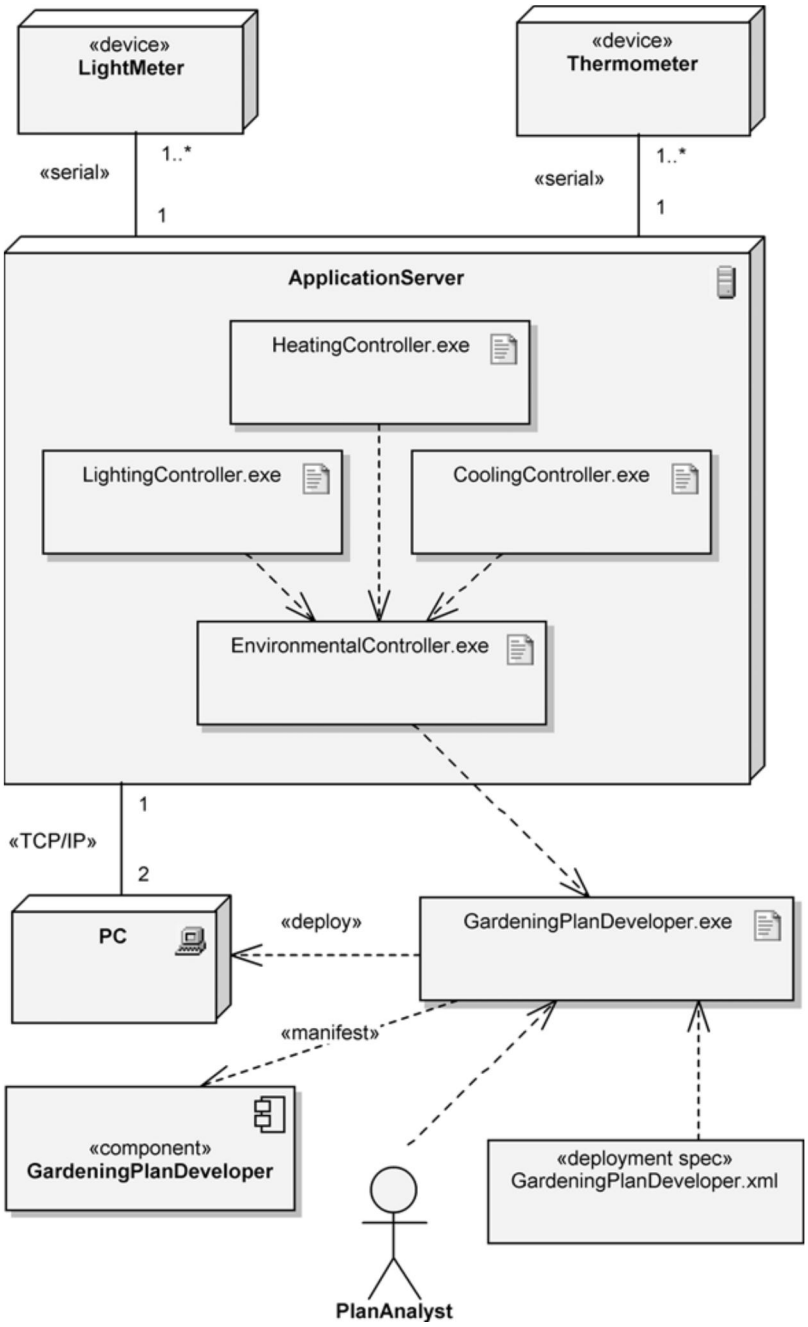
A node is a computational resource, typically containing memory and processing, on which artifacts are deployed for execution. Nodes may contain other nodes to represent complex execution capability; this is shown by nesting or using a composition relationship. There are two types of nodes: devices and execution environments. A device is a piece of hardware that provides computational capabilities, such as a computer, a modem, or a sensor. An execution environment is software that provides for the deployment of specific types of executing artifacts; examples include «database» and «J2EE server». Figure shows the three-dimensional cube icon that we use to represent a node, in this case, the PC and ApplicationServer nodes. The icon may be adorned with a symbol to provide additional visual specification of the node type.



Nodes communicate with one another, via messages and signals, through a communication path indicated by a solid line. Communication paths are usually considered to be bidirectional, although if a particular connection is unidirectional, an arrow may be added to show the direction. Each communication path may include an optional keyword label, such as «http» or «TCP/IP», that provides information about the connection. We may also specify multiplicity for each of the nodes connected via a communication path.

Essentials: The Deployment Diagram

In Figure, we provide an example of a deployment diagram drawn primarily from the physical architecture of the Environmental Control System within the Hydroponics Gardening System. Here we see that our system architects have decided to decompose this portion of our system into a network of two nodes (PC and ApplicationServer) and two devices (LightMeter and Thermometer).



The deployment of the EnvironmentalController.exe, LightingController.exe, HeatingController.exe, and CoolingController.exe artifacts on the ApplicationServer node is indicated by containment. Another way to denote deployment is shown by the dependency from the GardeningPlanDeveloper.exe artifact to the PC node labeled with «deploy». A third way to denote deployment is through textually listing the artifacts within the node icon; this is especially useful for larger or more complex deployment diagrams. We have three unnamed dependencies within Figure between artifacts: from the LightingController.exe, HeatingController.exe, and CoolingController.exe artifacts to the EnvironmentalController .exe artifact. These denote the dependencies between the components that they implement, rather than deployment onto a node. We also have another dependency, from the EnvironmentalController.exe artifact to the GardeningPlanDeveloper.exe artifact. This relates back to the interface on the EnvironmentalController component, which requires a gardening plan. Here we see that the gardening plan will be developed by PlanAnalyst using the GardeningPlanDeveloper.exe artifact, which manifests the GardeningPlanDeveloper component. The two devices, LightMeter and Thermometer, provide the ambient light and ambient temperature sensor readings required by the Environmental Controller.exe artifact in support of its provision of functionality to the system. GardeningPlanDeveloper.xml

deployment specification, which has a dependency relationship to the GardeningPlanDeveloper.exe artifact. This deployment specification describes deployment properties of the artifact, such as its execution and transaction specifics.

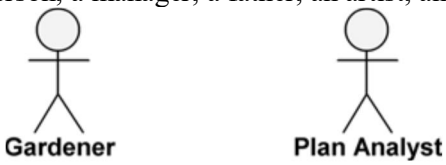
❖ Use Case Diagrams

Evidence over the years has shown that some of the most common reasons software projects fail center around poor or nonexistent communication between the key stakeholders. This is particularly critical when there is lack of alignment between the development organization and the line of business.

There is a strong need for an approach to system development that allows the development organization to understand what the business wants while not being cumbersome to the business staff (after all, their primary job is to run the daily operation of the business). Use case diagrams give us that capability. Use case diagrams are used to depict the context of the system to be built and the functionality provided by that system. They depict who (or what) interacts with the system. They show what the outside world wants the system to do.

Essentials: Actors

Actors are entities that interface with the system. They can be people or other systems. Actors, which are external to the system they are using, are depicted as stylized stick figures. Figure shows two actors for the Hydroponics Gardening System we discussed earlier. One way to think of actors is to consider the roles the actors play. In the real world, people (and systems) may serve in many different roles; for example, a person can be a salesperson, a manager, a father, an artist, and so forth.



Essentials: Use Cases

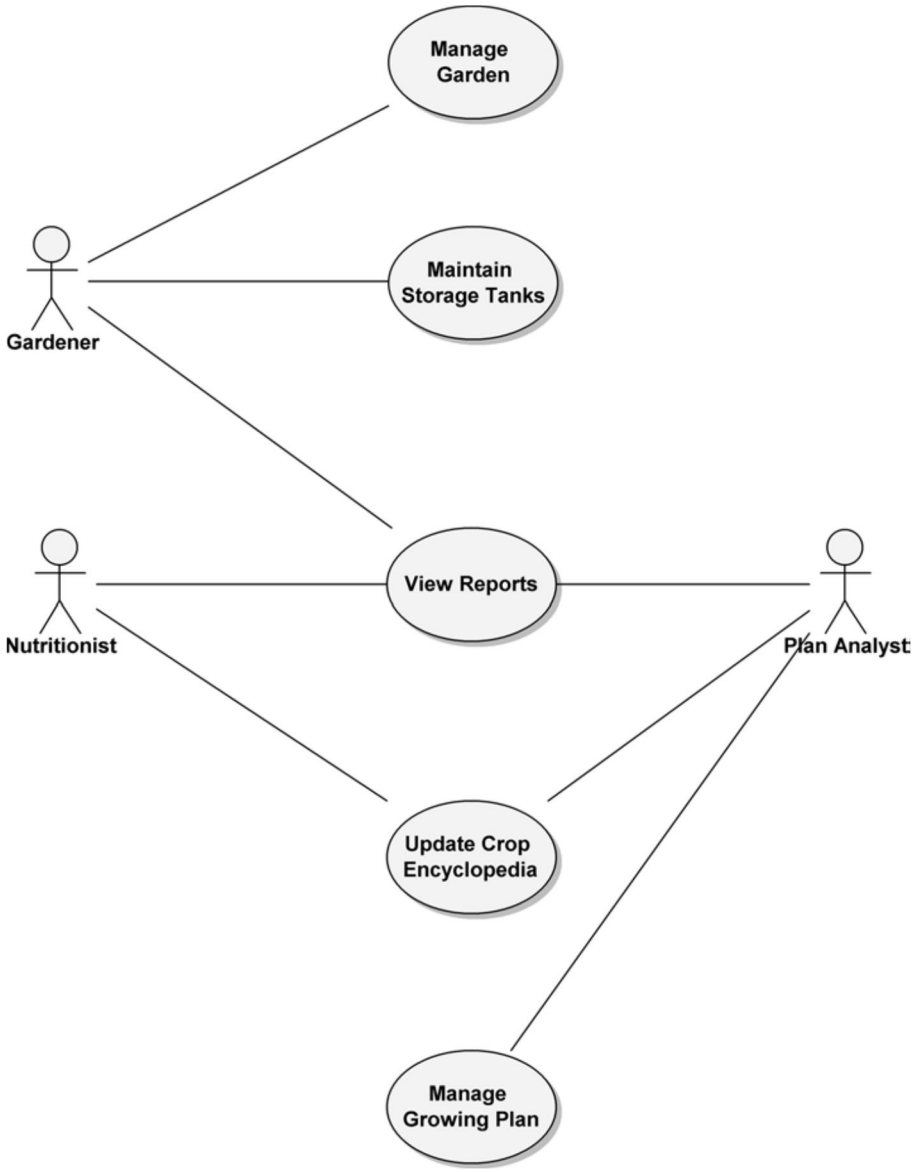
Use cases represent *what* the actors want your system to do for them. Figure depicts some use cases, shown as ovals, for the Hydroponics Gardening System. Use cases are not just any capability that your system may provide. A use case must be a *complete* flow of activity, from the *actor's point of view*, that provides *value* to the actor. A use case is a specific way of using the system by using some part of the functionality.

Essentials: The Use Case Diagram

To show which actors use which use cases, we can create a use case diagram by connecting them via basic associations, shown by lines, as in Figure. The associations in the use case diagram indicate which actors initiate which use cases. Here we see that only the Gardener actor can maintain the storage tanks, but all the actors may view reports.

Specifying Use Case Details

There are many different formats for use case specifications in the UML literature. Most include the following information at a minimum: the name of the use case; its purpose, in the form of a short description; the optimistic flow (i.e., the flow of events that will occur if everything goes right in the use case); and one or more pragmatic flows (i.e., those flows where things don't occur as you intended).



An Example Use Case Specification: Let us look at an example for the use case Maintain Storage Tanks.

Use case name: Maintain Storage Tanks

Use case purpose: This use case provides the ability to maintain the fill levels of the contents of the storage tanks. This use case allows the actor to maintain specific sets of water and nutrient tanks.

Optimistic flow:

A. Actor examines the levels of the storage tanks' contents.
B. Actor determines that tanks need to be refilled.
C. Normal hydroponics system operation of storage tanks is suspended by the actor.
D. Actor selects tanks and sets fill levels. For each selected tank, steps E through G are performed.
E. If tank is heated, the system disables heaters.
1. Heaters reach safe temperature.
F. The system fills tank.
G. When tank is filled, if tank is heated, the system enables heaters.
1. Tank contents reach operating temperature.
H. Actor resumes normal hydroponics system operation.

Pragmatic flows: **Conditions triggering alternate flow**

Condition 1: There is insufficient material to fill tanks to the levels specified by the actor.

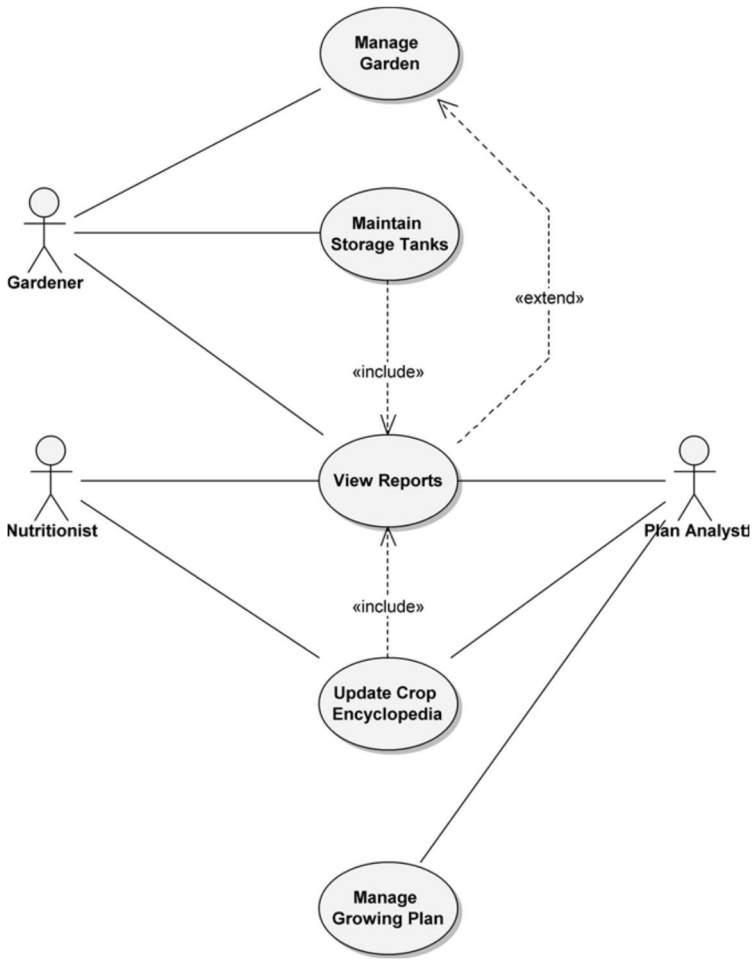
D1. Alert actor regarding insufficient material available to meet tank setting. Show amount of material
available.
D2. Prompt actor to choose to end maintenance or reset fill levels.
D3. If reset chosen, perform step D.
D4. If end maintenance chosen, perform step H.
D5. Else, perform step D2.

Condition 2: . . .

Other useful information may also be added to the specification, such as preconditions (what must be true prior to executing the use case), postconditions (what will be true after executing the use case), limitations, assumptions, and so forth.

Overall, a use case specification should not be very long it should be only a few pages. If your specifications are very long, you should reconsider whether your use case is doing too much. It may be that it is actually more than one use case. Also, for practical reasons, you cannot include all possible things that could trigger an alternate flow. Include the most important or critical alternates. Do not include every possible error condition, such as when the operator enters data in the wrong format (let the user interface handle that type of exception).

Advanced Concepts: «include» and «extend» Relationships



«include» Relationships:

The Nutritionist actor using the use case will have to see what is in the crop encyclopedia prior to updating it. This
is why the Nutritionist can invoke the View Reports use case. The same is true for the Gardener actor whenever invoking Maintain Storage Tanks. Neither actor should be executing the use cases blindly.

Therefore, the View Report use case is a common functionality that both other use cases need. This can be depicted on the use case model via an «include» relationship, as shown in Figure. This diagram states, for example, that the Update Crop Encyclopedia use case includes the View Reports use case. This means that View Reports must be executed when Update Crop Encyclopedia is executed. Update Crop Encyclopedia would not be considered complete without View Reports. Where an included use case is executed, it is indicated in the use case specification as an inclusion point. The inclusion point specifies where, in the flow of the including use case, the included use case is to be executed.

«extend» Relationships

While developing your use cases, you may find that certain activities might be performed as part of the use case but are not mandatory for that use case to run.

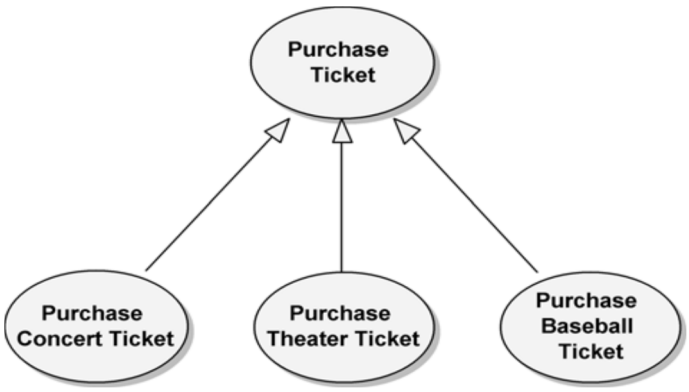
The Dangers of «include» and «extend» Relationships

These concepts are commonly misused during use case modeling. The primary cause is that some people are not clear about the semantic differences between «include» and «extend». Another common error we see with these relationships is violation of basic use case principles. Included and extending use cases are still use cases and must conform to the use case principles cited earlier; a use case represents a *complete* flow of activity of *what* the actor wants your system to do from the *actor's point of view* that provides *value* to the actor. This is the most prevalent

problem we see regarding use case models, in which people break down use cases into smaller and smaller pieces, using «include» or «extend» to tie them all together.

Advanced Concepts: Generalization

Generalization relationships, can also be used to relate use cases. As with classes, use cases can have common behaviors that other use cases (i.e., child use cases) can modify by adding steps or refining others. For example, Figure shows the use cases for purchasing various tickets. Purchase Ticket contains the basic steps necessary for purchasing any tickets, while the child use cases specialize Purchase Ticket for the specific kinds of tickets being purchased.



❖ Activity Diagrams

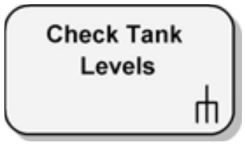
Activity diagrams provide visual depictions of the flow of activities, whether in a system, business, workflow, or other process. These diagrams focus on the activities that are performed and who (or what) is responsible for the performance of those activities. The elements of an activity diagram are action nodes, control nodes, and object

nodes. There are three types of control nodes: initial and final (final nodes have two varieties, activity final and flow final), decision and merge, and fork and join.

Essentials: Actions

Actions are the elemental unit of behavior in an activity diagram. Activities can contain many actions which are what activity diagrams depict. Figure shows an action that can be performed in our hydroponics example.

Note the rake symbol inside the action notation at its bottom right-hand corner. This denotes that this action is a callBehavior type action. So, as a practical matter, we may want to use the rake symbol only when we have actually defined that activity to be called.



Essentials: Starting and Stopping

Since an activity diagram shows a process flow, that flow must start and stop somewhere. The starting point (the initial node) for an activity flow is shown as a solid dot, and the stopping point (the activity final node) is shown as a bull's-eye.

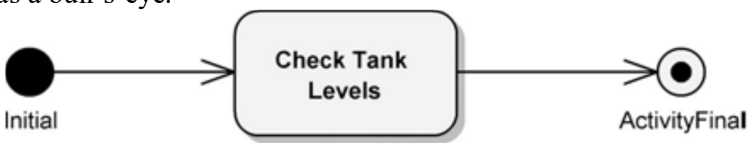
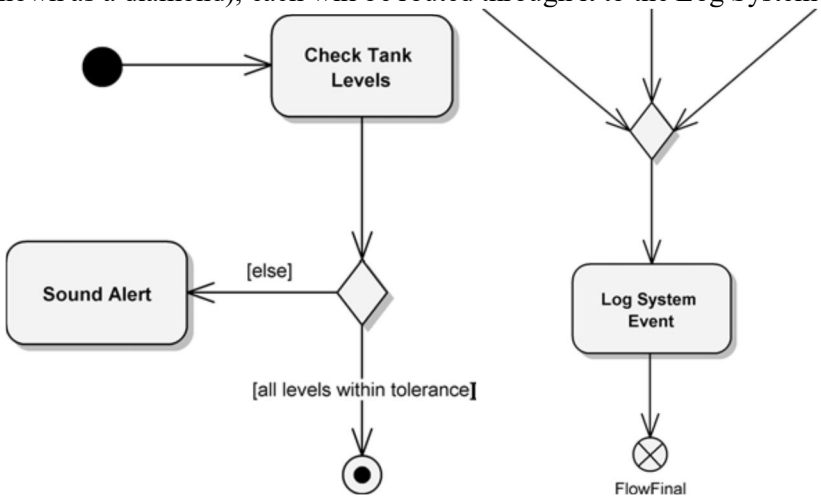


Figure depicts a simple activity diagram composed of one action, Check Tank Levels. Another type of final node is the flow final node, which is denoted by a circle with a nested "X" symbol.

Essentials: Decision and Merge Nodes

Decision and merge nodes control the flow in an activity diagram. Each node is represented by a diamond shape with incoming and outgoing arrows. A decision node has one incoming flow and multiple outgoing flows. Its purpose is to direct the one incoming flow into one (and only one) of the node's outgoing flows. The outgoing flows usually have guard conditions that determine which outgoing path is selected. Figure shows the guard condition [all levels within tolerance] and the alternative [else]. There is no waiting or synchronization
at a decision node.

Merge nodes take multiple input flows and direct any and all of them to one outgoing flow. There is no waiting or synchronization at a merge node. In Figure, whenever any of the three incoming flows reach the merge point (shown as a diamond), each will be routed through it to the Log System Event action.



Essentials: Partitions

The elements in an activity diagram can be grouped by using partitions. The purpose of a partition is to indicate where the responsibility lies for performing specific activities. For systems, the partitions may be other systems or subsystems. In application modeling, the partitions may be objects in the application. Figure shows how the various activities that comprise the Maintain Storage Tanks use case of our Hydroponics Gardening System are partitioned to the Gardener, WaterTank, and NutrientTank.

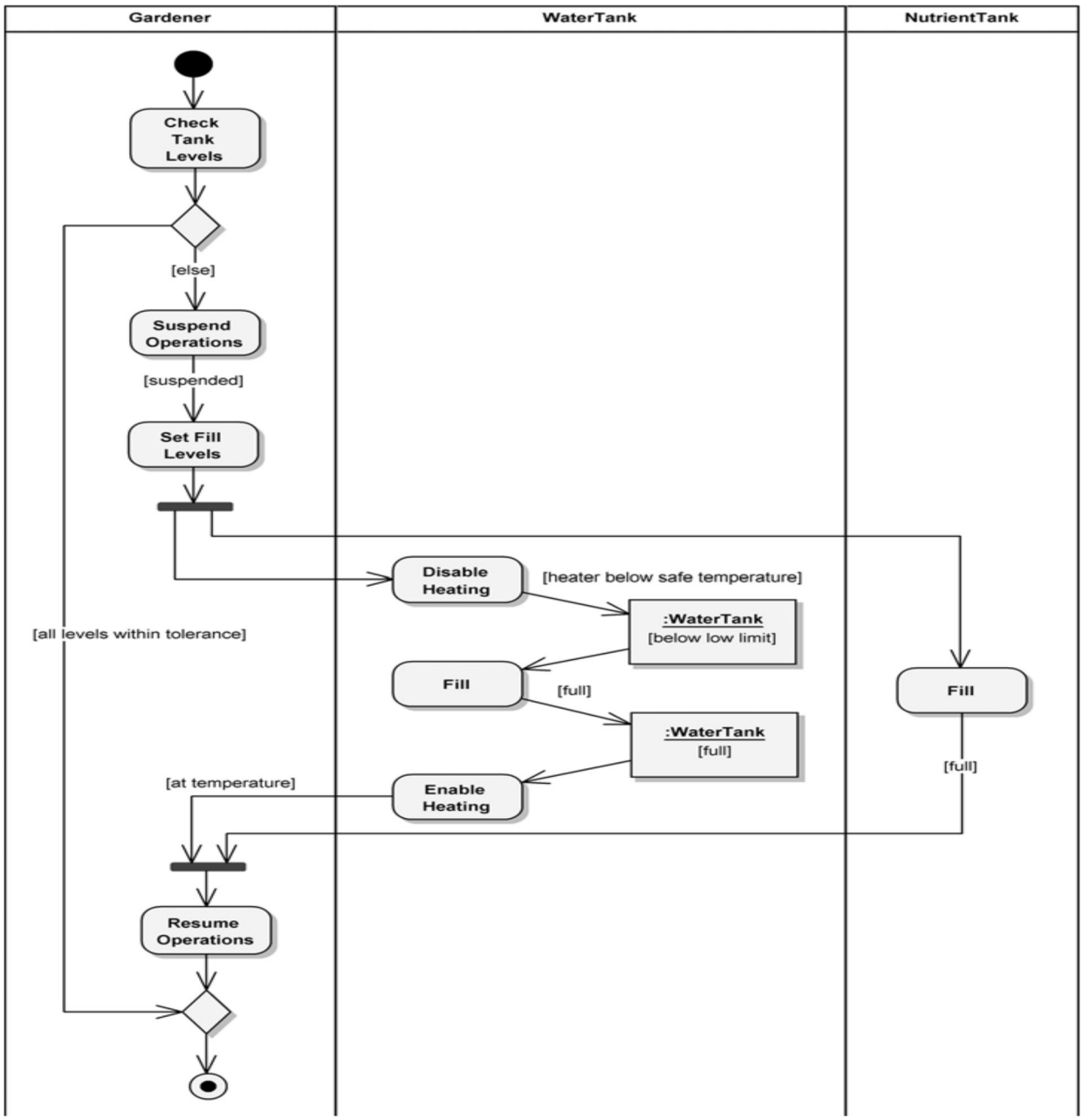
Advanced Concepts: Forks, Joins, and Concurrency

Fork and join nodes are analogous to decision and merge nodes, respectively. The critical difference is concurrency. Forks have one flow in and multiple flows out. All the outbound flows occur concurrently. In Figure, a single flow goes from the Set Fill Levels action into the fork, which is the first thick horizontal line. Thereafter, the NutrientTank flow (with the Fill action) and the WaterTank flow (with the Disable Heating, Fill, and Enable Heating actions) both occur in parallel.

A join has multiple incoming flows and a single outbound flow. In Figure, the second thick horizontal line is a join. Both of the incoming flows, NutrientTank and WaterTank, must be complete before the outbound flow continues to the Resume Operations action.

Advanced Concepts: Object Flows

Figure shows an object flow added to our previous activity diagram. In the WaterTank partition, two object nodes (rectangles labeled :WaterTank) have been added to the flow. This shows that, after the heating is disabled, the water tank is below its low operational limit and that, after the Fill action, the water tank is full.



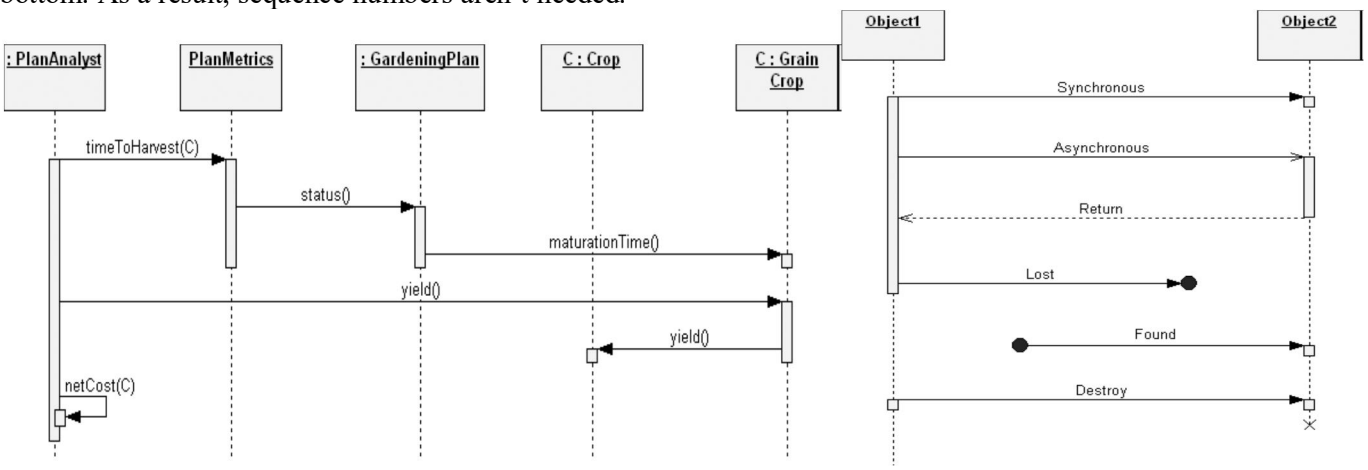
❖ **Sequence Diagrams**

A sequence diagram shows how objects operate with one another and in what order. It is a construct of a message sequence chart. A sequence diagram shows object interactions arranged in time sequence.

Essentials: Objects and Interactions

Essentials: Lifelines and Messages

A dashed vertical line, called the *lifeline*, is drawn below each object. These indicate the existence of the object. *Messages* (which may denote events or the invocation of operations) are shown horizontally. The endpoints of the message icons connect with the vertical lines that connect with the entities at the top of the diagram. Messages are drawn from the sender to the receiver. Ordering is indicated by vertical position, with the first message shown at the top of the diagram, and the last message shown at the bottom. As a result, sequence numbers aren't needed.



The notation used for messages (i.e., the line type and arrowhead type) indicates the type of message being used, as shown in Figure. A synchronous message (typically an operation call) is shown as a solid line with a filled arrowhead. An asynchronous message has a solid line with an open arrowhead. A return message uses a dashed line with an open arrowhead. A lost message (a message that does not reach its destination) appears as a synchronous message that terminates at an endpoint (a black dot). A found message (a message whose sender is not known) appears as a synchronous message that originates at an endpoint symbol.

Advanced Concepts: Destruction Events

A destruction event indicates when an object is destroyed. It is shown as an X at the end of a lifeline. See the Object2 lifeline in Figure for an example.

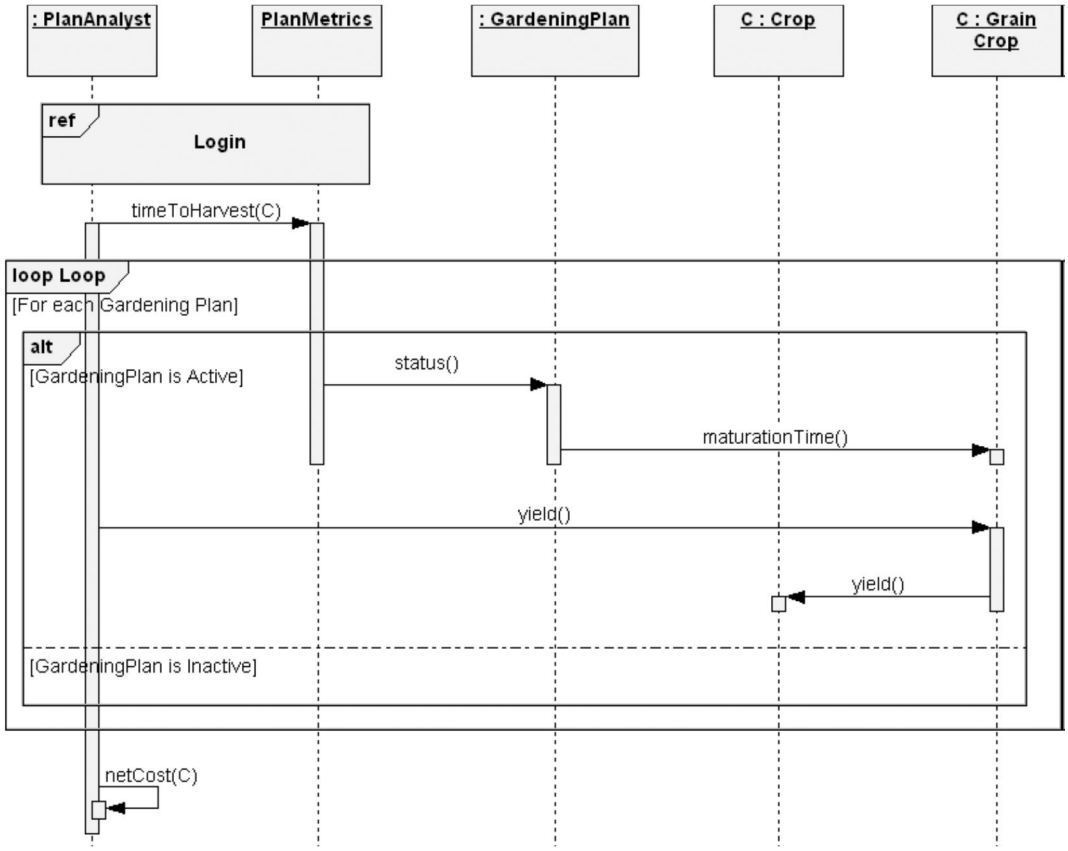
Advanced Concepts: Interaction Use

An interaction use is merely a way to indicate on a sequence diagram that we want to reuse an interaction that is defined elsewhere. This is shown, as in Figure, as a frame labeled ref. In this case, we have modified our earlier sequence diagram to introduce a login sequence, required before the PlanAnalyst uses the system. The frame, labeled ref, indicates that the Login sequence is inserted (i.e., copied) where this fragment is placed in this sequence. The actual login sequence would be defined on another sequence diagram.

Advanced Concepts: Control Constructs

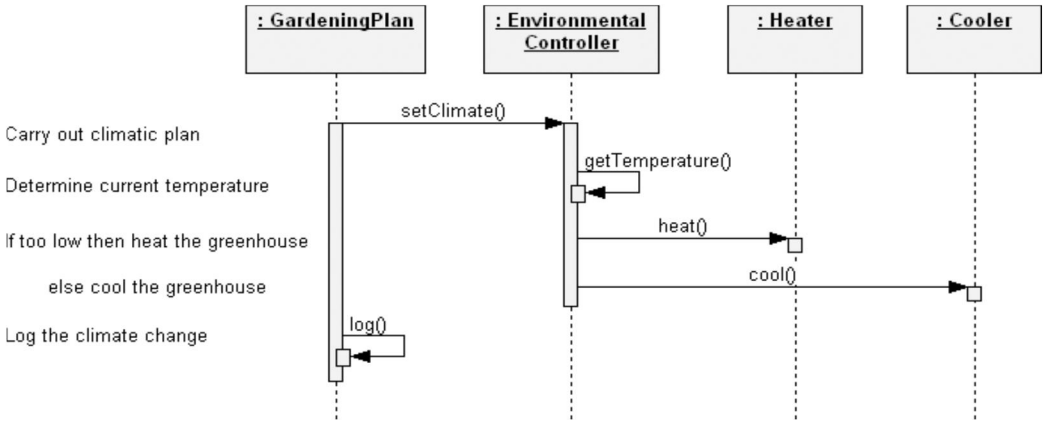
Just as we saw fragments being used to simplify sequence diagrams, they can similarly be used to indicate flow control constructs on sequence diagrams. For example, Figure shows the introduction of a loop in our sequence diagram. Now let us assume that we have many GardeningPlan objects, some active, some inactive (past plans that are now saved just for informational purposes). Here within the loop, an alternate choice is made, governed by the conditions [GardeningPlan is Active] and [GardeningPlan is Inactive]. These conditions select which part of the sequence is executed. The alt frame is divided into two regions,

each with its own condition. When a condition is true, the behavior in that region of the frame is performed.



Advanced Concepts: Execution Specification

For example, in Figure, we see that the anonymous instance of the GardeningPlan object is the ultimate focus of control, and its behavior of carrying out a climatic plan invokes other methods, which in turn call other methods that eventually return control back to the GardeningPlan object.



❖ Interaction Overview Diagrams

Interaction overview diagrams are a combination of activity diagrams and interaction diagrams that are intended to provide an overview of the flow of control between interaction diagram elements. Though any type of interaction diagram (sequence, communication, or timing) may be used, the sequence diagram will likely be the most prevalent.

Essentials: Frames

The interaction overview diagram is typically surrounded by a frame; however, the frame is optional when the context is clear. In Figure, we see the surrounding frame with the name sd MaintainTemperature lifelines :EnvironmentalController, :Heater, :Cooler in the compartment in the upper left-hand corner.

- sd: a tag that indicates this is an interaction diagram
- MaintainTemperature: a name describing the purpose of the diagram
- lifelines :EnvironmentalController, :Heater, :Cooler: an optional list of contained lifelines

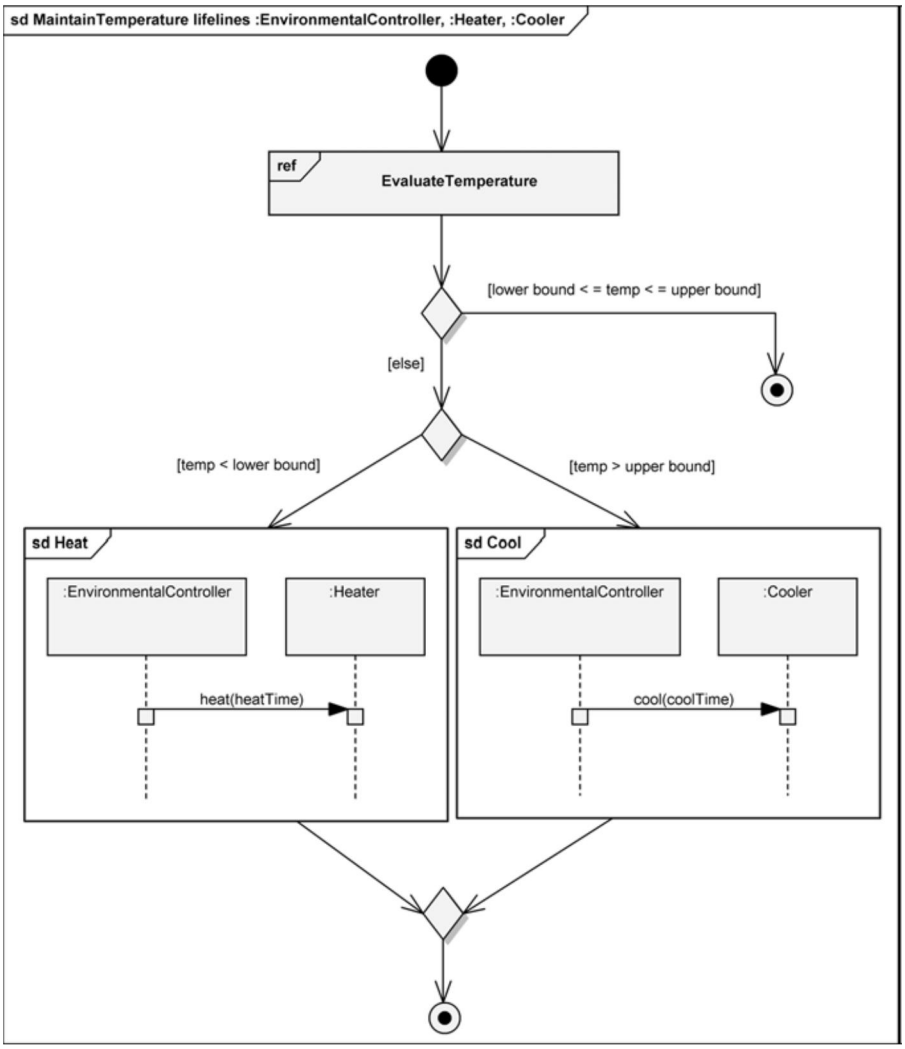
This interaction overview diagram contains flow of control elements and three frames, EvaluateTemperature, Heat, and Cool.

Essentials: Flow of Control Elements

The flow of control within an interaction overview diagram is provided by a combination of activity diagram elements to provide for both alternate and parallel paths.

The alternate path control is provided by combinations of a decision node, where the appropriate path is chosen, and a corresponding merge node (as appropriate) to bring the alternate paths together. This combination appears twice in Figure. First, a decision node is used to choose a path based on whether the temperature of the Hydroponics Gardening System is within bounds (therefore, requiring no action) or out of bounds, which requires either heating or cooling. The interaction constraint [lower bound < =temp < = upper bound] is used to choose the appropriate path. The second combination of a decision node and a merge node controls whether heating or cooling is applied by using the two interaction constraints, [temp < lowerbound] and [temp > upper bound].

Flow of control within parallel paths is provided by combinations of a fork node, to split into parallel paths, and a corresponding join node to bring the parallel paths together. One important concern with parallel paths is that tokens from all paths must arrive at the join node before the flow is allowed to continue. This requires us to ensure that, wherever an interaction constraint may block flow along a path, there is an alternate path for the token to proceed. .



Essentials: Interaction Diagram Elements

The interaction overview diagram contains two types of elements to provide the interaction diagram information, either an interaction or an interaction use. The interaction is any type of interaction diagram that provides the nested details of the interaction; these are typically sequence diagrams. They can be anonymous or named, as in Figure, which shows the Heat and Cool interactions.

The interaction use references an interaction diagram rather than providing its details. Figure contains an example, the EvaluateTemperature interaction use. The details of EvaluateTemperature would show how concerns, such as the following, would be managed:

- Periodicity of temperature readings
- Protection of compressors by not restarting the :Cooler sooner than five minutes since shutdown
- Temperature adjustments based on time of day
- Temperature ranges for different crops

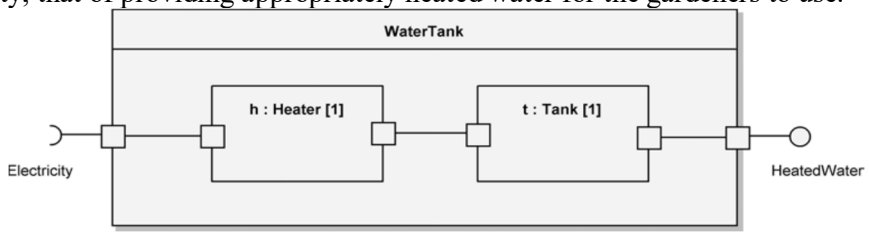
❖ Composite Structure Diagrams

Composite structure diagrams provide a way to depict a structured classifier with the definition of its internal structure. This internal structure is comprised of parts and their interconnections, all within the namespace of the composite structure.

Essentials: Composite Structure Parts

The composite structure diagram for the Hydroponics Gardening System's WaterTank is shown in Figure. Its name is placed in the top compartment; the specific naming convention should be defined by the

development team. WaterTank contains the Heater and Tank parts, which collaborate to provide its functionality, that of providing appropriately heated water for the gardeners to use.



The name of a composite structure part uses the format of *role name* :*Class Name* [*multiplicity*], where the role name defines the role played by a part within the composite structure. Though showing the multiplicity is optional, we include it in Figure to make clear that WaterTank consists of one Heater and one Tank.

Essentials: Composite Structure Ports and Interfaces

The composite structure and its parts interface with their external environment through ports, denoted by a small square on the boundary of the part or composite structure. In Figure, we see that Heater and Tank both have ports through which they interact with each other to provide the functionality of WaterTank. In addition, WaterTank has a port through which it receives electricity for the Heater and a port through which it provides the heated water from the Tank to its environment. Port names are in the format *port name* : *Port Type* [*multiplicity*]. The port type is optional when naming a port. These interfaces are commonly shown in the ball-andsocket notation. A required interface uses the socket notation to denote the services expected from its external environment, whereas the ball notation denotes the services it offers through its provided interfaces. As part of WaterTank, Heater receives electricity from the Hydroponics Gardening System, and Tank provides heated water to the gardeners.

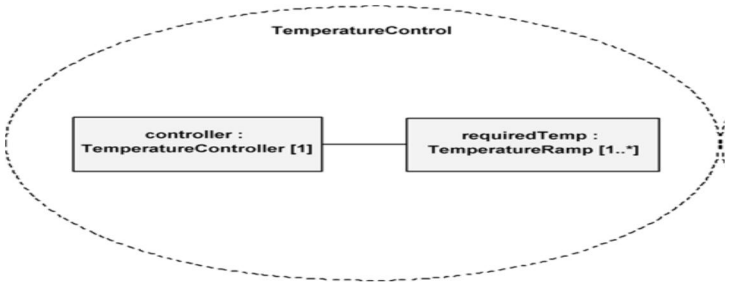
Essentials: Composite Structure Connectors

Connectors within composite structure diagrams provide the communication links between the composite and its environment, as well as the means for its parts to communicate internally. In Figure we have three connectors between its ports; the two that connect to the boundary of the composite are called *delegation connectors*, and the one between Heater and Tank is called an *assembly connector* (also known as an *interface connector*). Here we have Heater providing heat to Tank to fulfill its service need.

Advanced Concepts: Collaborations

A collaboration is a type of structured classifier that specifies the runtime interactions of classifier instances. It differs from the composite structure in that it is not instantiated and does not actually own these instances but defines the roles that classifier instances must assume and the connectors over which they collaborate to provide the functionality of interest.

The collaboration's primary use is to define templates, that is, patterns of roles joined by connectors. At runtime, classifier instances will be bound to these roles so they may cooperate to provide the functionality defined by the collaboration. For example, Figure shows the TemperatureControl collaboration defining a pattern for controlling the temperature within the Hydroponics Gardening System. In this pattern, TemperatureController uses TemperatureRamp, which defines the precise temperature variations required over time to support the needs of a crop. The name of a collaboration, shown inside the dashed oval that encapsulates the collaboration, may be partitioned from the role definitions by a horizontal dashed line (not shown here). In this collaboration, we have two defined roles, TemperatureController and TemperatureRamp, joined by a connector.



Since the connector is unnamed, it will be realized by a temporary runtime means such as a property or an argument. If it were named, it would be realized by an instance of an actual association, that is, a link. Roles are labeled with a name and type in the format of *role name* : *RoleType* [*multiplicity*]. The role name describes a particular classifier instance that may fulfill this role, and the role type is a constraint on this classifier instance. We have shown the role names, role types, and multiplicity for the TemperatureController and TemperatureRamp roles. A role defines the properties that a structured classifier must have to participate in the collaboration.

❖ State Machine Diagrams

State machines are also known as behavioral state machines. State machine diagrams are typically used to describe the behavior of individual objects. State machine diagrams focus on the states and transitions between those states versus the flow of activities.

Essentials: Initial, Final, and Simple States

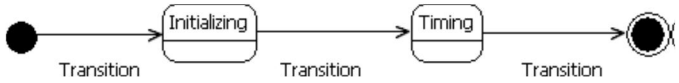
When an object is in a given state, it can do the following:

- Execute an activity
- Wait for an event
- Fulfill a condition
- Do some or all of the above

In every state machine diagram, there must be exactly one default initial state, which we designate by writing an unlabeled transition to the state from a special icon, shown as a filled circle. Less often, we need to designate a stop state. We designate a stop state by drawing an unlabeled state transition from the state to a special icon, shown as a filled circle inside a slightly larger unfilled circle. Initial and final states are technically called *pseudostates*. Figure depicts the elements for a duration timer in our hydroponics system. For simple states, the state name is shown in the rounded rectangle depicting the state. Here we have two simple states for our timer—Initializing and Timing.

Essentials: Transitions and Events

The movements between states are called transitions. On a state machine diagram, transitions are shown by directed arrows between states. Each state transition connects two states. Figure shows a transition from the initial state to the state named Initializing, from Initializing to Timing, and from Timing to the final state. Moving between states is referred to as *firing the transition*. A state may have a state transition to itself, and it is common to have many different state transitions from the same state, although each such transition must be unique, meaning that there will never be any circumstances that would trigger more than one state transition from the same state.



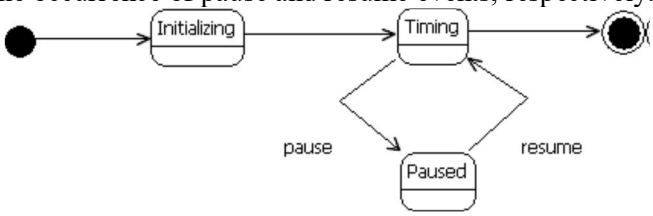
There are various ways to control the firing of a transition. A transition that has no annotation is referred to as a *completion transition*. This simply means that when the source state completes, the transition

automatically fires, and the target state is entered. You can see this in Figure between the Initializing state and the Timing state. When the duration timer is initialized, it immediately begins timing.

In other cases, certain events have to occur for the transition to fire. Such events are annotated on the transition. An event is some occurrence that may cause the state of a system to change. For example, in the Hydroponics Gardening System, the following events play a role in the system's behavior.

- A new crop is planted.
- A crop becomes ready to harvest.
- The temperature in a greenhouse drops because of inclement weather.
- A cooler fails.
- Time passes.

Each of the first four events is likely to trigger some action, such as starting or stopping the execution of a specific gardening plan, turning on a heater, or sounding an alarm to the gardener. In Figure, we develop the duration timer state diagram. Here you see that the timer execution can be put into a Paused state and timing resumed through the occurrence of pause and resume events, respectively.

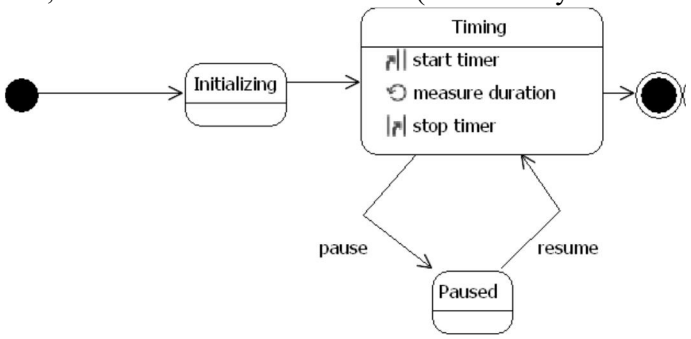


Advanced Concepts: State Activities—Entry, Do, and Exit Activities

In particular, we may specify some activity that is to be carried out at certain points in time with respect to a state.

- Perform an activity upon entry of the state.
- Do an activity while in the state.
- Perform an activity upon exit of the state.

Figure shows an example of this concept. Here we see that upon entering the Timing state, we start the timer (indicated by the icon of an arrow next to two parallel lines), and upon exiting this state (indicated by the icon of an arrow between two parallel lines), we stop the timer (note that these icons are tool specific). While in this state, we measure the time duration (indicated by the circular arrow).



Advanced Concepts: Controlling Transitions

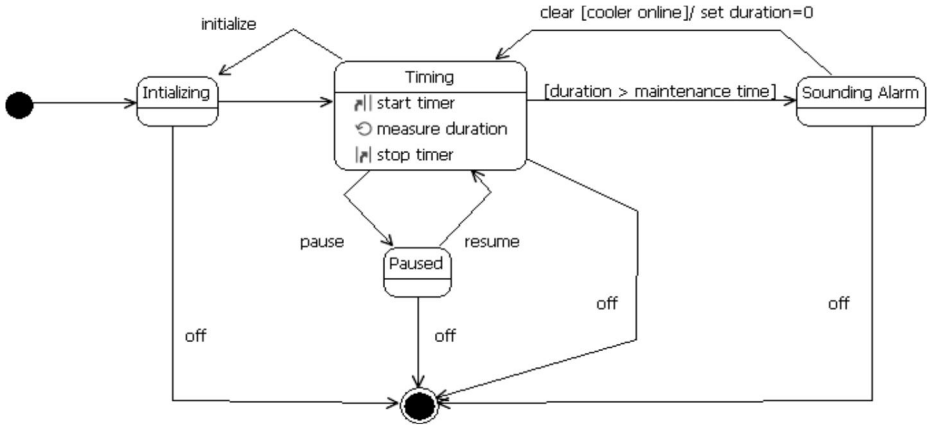
Conditions can be specified to control the transition. These conditions act as guards so that when an event occurs, the condition will either allow the transition (if the condition is true) or disallow the transition (if the condition is false). Another way to control transition behavior is to use effects. An effect is a behavior (e.g., an activity, an action) that occurs when a specified event occurs. Thus, when a transition event occurs, the transition fires and the effect also occurs.

Let us expand our duration timer example to show the use of effects. One important event that could happen in our hydroponics system is that a cooler could fail. We will change our basic timer to a duration timer used to measure the total operational time during which the cooler is running. The purpose is to alert us to perform maintenance on the cooler after it has been in operation for a certain period of time. We hope that regular maintenance will prevent a failure of the cooler. So we enhance our state machine

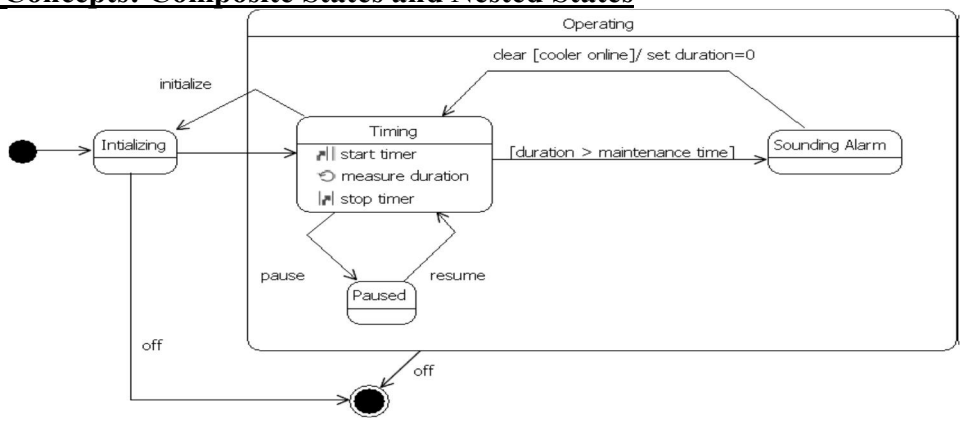
diagram as shown in Figure. In this diagram, you can see that the timeout transition has been replaced with a condition. This condition specifies that when the duration exceeds the maintenance time, the timer transitions into the Sounding Alarm state, alerting us to the need for maintenance. The use of an event, condition, and effect in combination is shown on the transition from Sounding Alarm to Timing. Here, once the maintenance is complete, we want to clear the alarm condition and resume timing. There is a clear event, but it is guarded by the condition that the cooler must be back online first (before timing is resumed), shown in brackets. If the cooler is not online, the transition does not fire. If it is, the effect occurs (the duration timer is set to zero, as shown after the slash), the transition fires, and timing begins again. The order of evaluation in conditional state transitions is important. Given state S with transition T on event E with condition C and effect A, the following order applies.

1. Event E occurs.
2. Condition C is evaluated.
3. If C evaluates true, then transition T is triggered, and effect A is invoked.

For example, suppose that event E occurs, and condition C evaluates true, but then the execution of the exit action changes the world so that C no longer evaluates true. The state transition will still be triggered.



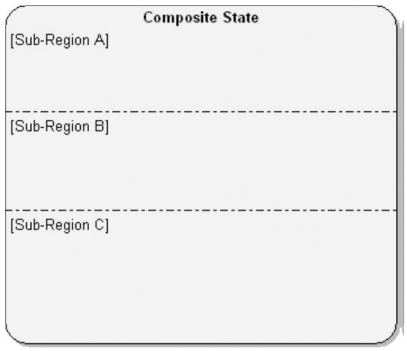
Advanced Concepts: Composite States and Nested States



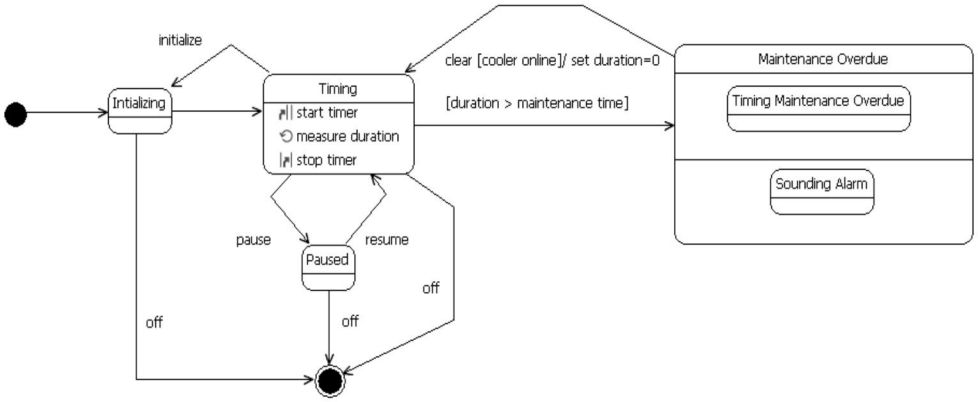
In Figure, we have nested the states Timing, Sounding Alarm, and Paused. This nesting is depicted with a surrounding boundary known as a *region*. The enclosing boundary is called a *composite state*. Thus we now have a
composite state named Operating that contains the nested states Timing, Sounding Alarm, and Paused.

Advanced Concepts: Concurrency and Control

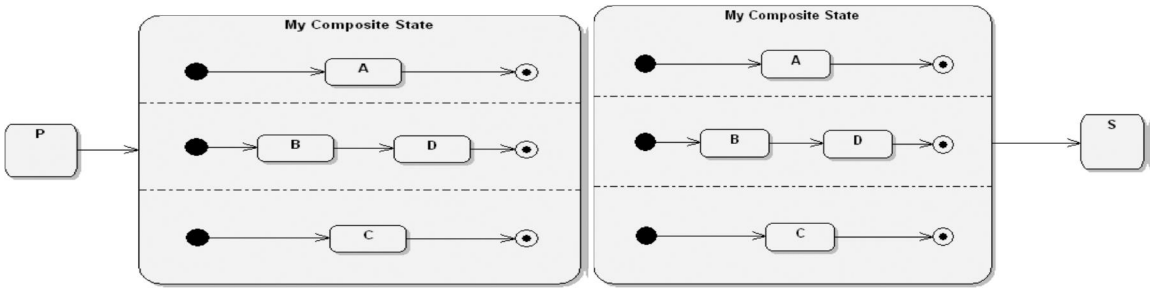
Concurrent behavior can be depicted in state machine diagrams by simply partitioning a composite state into two or more sub-regions by using dotted lines. Each sub-region within the composite state represents behavior that will occur concurrently. Figure shows a state with three concurrent sub-regions.



For our example, when the cooler is overdue for maintenance, the duration timer sounds an alarm. Earlier, in Figure, we showed this by having the state machine transition into the Sounding Alarm state. Let us say that, instead of just sounding the alarm, we want the system to also capture how long the cooler remains overdue for maintenance. We show this in Figure by replacing the Sounding Alarm state with a composite state, Maintenance Overdue, that contains two concurrent states: Sounding Alarm and Timing Maintenance Overdue (which captures how long the cooler has been overdue for maintenance). Thus, when the transition into the Maintenance Overdue composite state occurs, both substates start and run concurrently, that is, the overdue maintenance timer starts and the alarm sounds.

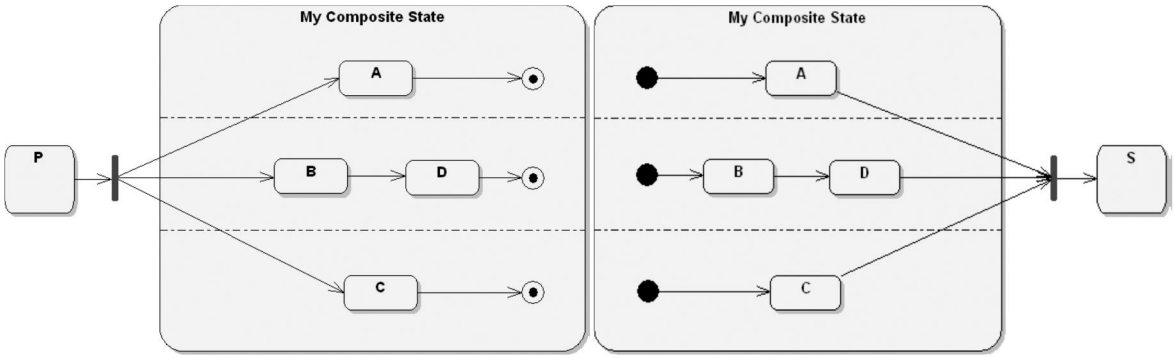


You must be careful to make sure that how you diagram the flow correctly represents your intent. Let us examine this with a generic composite state, as shown in Figure. You can transition into a composite state in various ways. For example, you can have a transition to the composite state, as shown in Figure. In this case, the concurrent submachines would all activate and begin to operate concurrently. Each individual sub-region would execute beginning at the default initial state (pseudostate) for that sub-region. While it is not necessary, we recommend overtly showing the initial states in the sub-region for clarity.

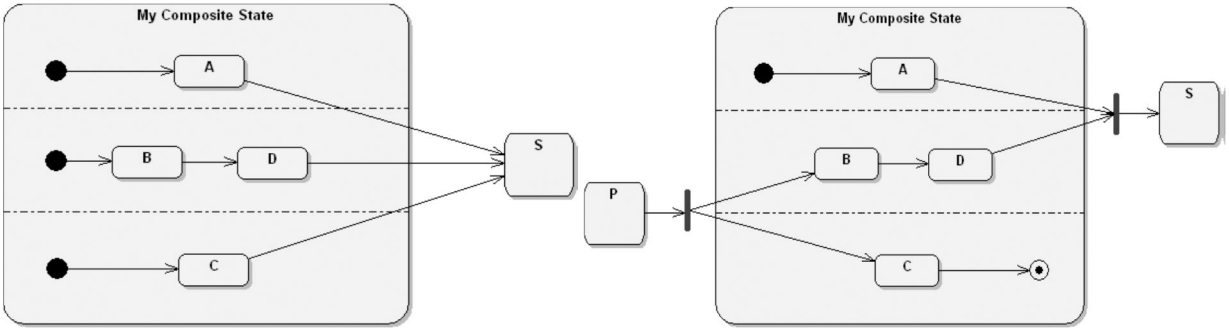


Another way to move into a composite state is to use a fork. A fork vertex is actually another pseudostate (like initial and final states) and is used to split an incoming transition into two or more outgoing transitions. Figure shows the use of a fork in this manner. The single incoming transition may have an event and or a guard condition on it. The multiple outgoing transitions may not. In our case, the fork splits the flow of control, and concurrent execution within the composite state would begin with substates A, B, and C, as these are the target substates of the multiple outgoing transitions from the fork. You can also use a join vertex (another pseudostate) to perform a similar merging of control. Figure shows multiple transitions from the individual concurrent sub-regions to a vertical bar, where they are merged into one

outgoing transition. This single outgoing transition may also have an event and a guard condition on it. In this case, transition to state S would occur when all the joined substates (A,
D, and C) are active, the event occurs, and the condition is satisfied.



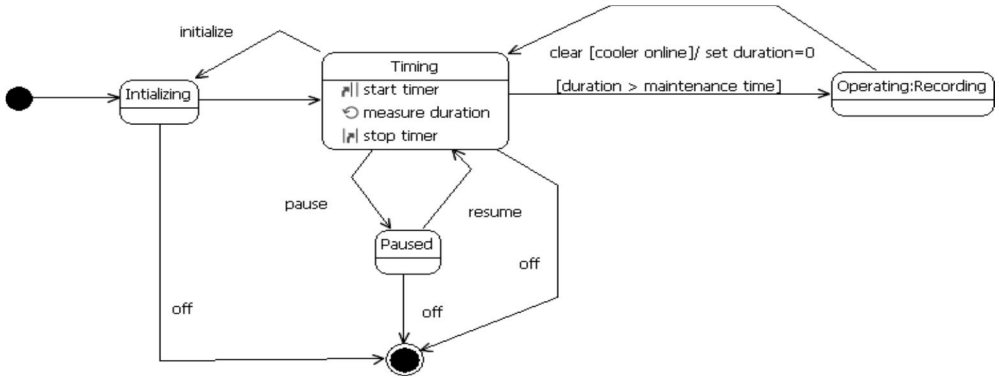
In Figure, there is no merging of control as there is when joins are used. In this case, if any one of the transitions from substates A, D, or C fire, all the other concurrent substates will terminate. Along those same lines, if some sub-regions do not explicitly participate in a join (see Figure), when the join transition fires, the nonparticipating sub-regions will be forced to terminate. Conversely, if some sub-regions do not explicitly participate in a fork, the nonparticipating sub-regions will execute starting at their initial states.



Advanced Concepts: Submachine State

Submachine states are used to simplify state machine diagrams or to contain common behavior that may be reusable in various diagrams. Let us suppose that much more needs to be done when the maintenance time is exceeded than just sounding the alarm. Let us say that a second timer needs to run to count how much time the cooler has exceeded its maintenance cycle and that the system needs to record temperature, coolant pressure, on/off cycles, and humidity, making that information available as graphs showing the values over time. These new recording requirements could result in a quite complex state machine diagram.

In order to keep the diagrams simple, we could simply replace the Sounding Alarm substate with a submachine state named Operating:Recording (see Figure). This submachine state represents *an entirely separate state machine diagram* that would depict all the detailed recording requirements just mentioned. In this manner, submachines enable us to organize complex state machine diagrams into understandable structures.

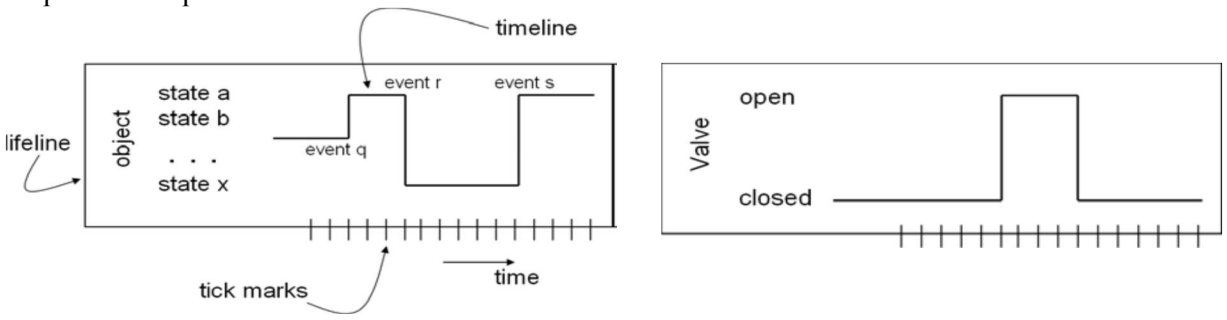


❖ Timing Diagrams

Timing diagrams are a type of interaction diagram. Their purpose is to show how the states of an element or elements change over time and how events change those states.

Essentials: Layout

The general layout of a timing diagram is reminiscent of a sequence diagram laid on its side. Timing diagrams have one or more lifelines (which look like a horizontal partition) for each object in the diagram. The lifeline name (i.e., the object name) is shown in the lifeline. The possible states of the object are listed inside the lifeline. Also, a timeline shows how the object changes from one state to another. Events that drive the state changes are shown along the timeline. The horizontal axis shows time and may also show tick marks. For example, Figure shows a simple timing diagram for a Valve object that is controlled to fill the WaterStorageTank object in our Hydroponics Gardening System. Valve has two simple states: open and closed.

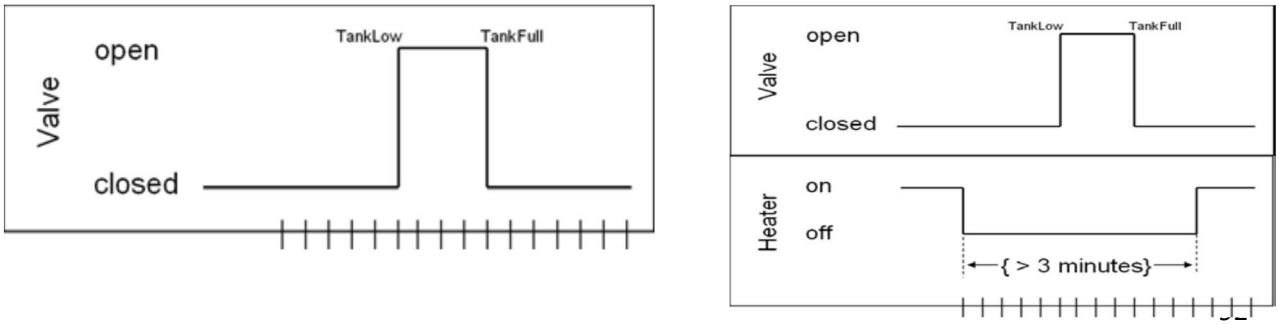


Essentials: Events

Events (or other stimuli such as messages) that cause state changes are shown in the lifeline near the timeline of the object. In Figure, two events have been added, that is, TankLow and TankFull, which cause changes in the state of the valve.

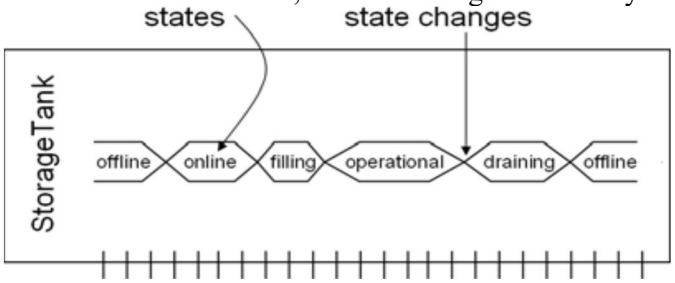
Essentials: Constraints

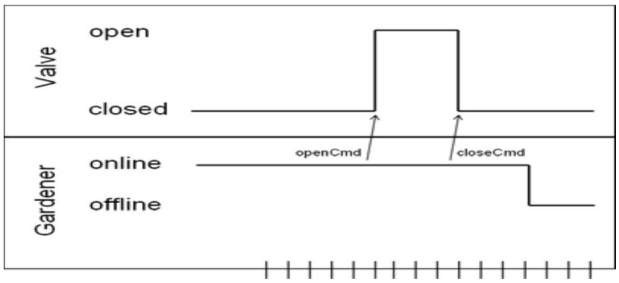
Constraints can be used to specify conditions or limits that restrict the change of state shown on a timing diagram. In Figure, we show a timing diagram with both the Valve and the Heater objects represented. This diagram shows the relationship between the state of Heater and the state of Valve. Here we see a constraint on Heater that restricts how quickly the heater can be turned back on. The constraint indicates that once the heater is turned off, at least three minutes must pass before the heater is turned back on.



Advanced Concepts: Alternate Representations

In cases where timing diagrams have many lifelines, or objects that have many states, instead of using a timeline, we can use a more compact representation, as shown in Figure. States are shown within the hexagons, and the state changes occur between them. Instead of the change of state being indicated as a rise and fall of a timeline, the state changes are merely shown progressing along the horizontal axis.





Advanced Concepts: Events versus Messages

As stated earlier, not only can events drive state changes, but other stimuli such as messages can, too. So which should be used when? The subtle difference in this case is that an event has a specific location in space and time, which is not true for messages. For example, the two events shown earlier in above Figure, TankLow and TankFull, physically occur in the actual storage tank. Instead of using these events, we could use messages to open or close the valve. Say that the gardener decides to add more water to the water storage tank, even though the tank's level is not physically low. The gardener simply wants to increase the amount of water in the tank. In this case, using a message would be better than using an event. Above Figure shows two messages (openCmd and closeCmd) that command the valve to open, thus filling the tank, and close, to stop the filling, respectively.

❖ Communication Diagrams

A communication diagram is a type of interaction diagram that focuses on how objects are linked and what messages they pass as they participate in a specific interaction.

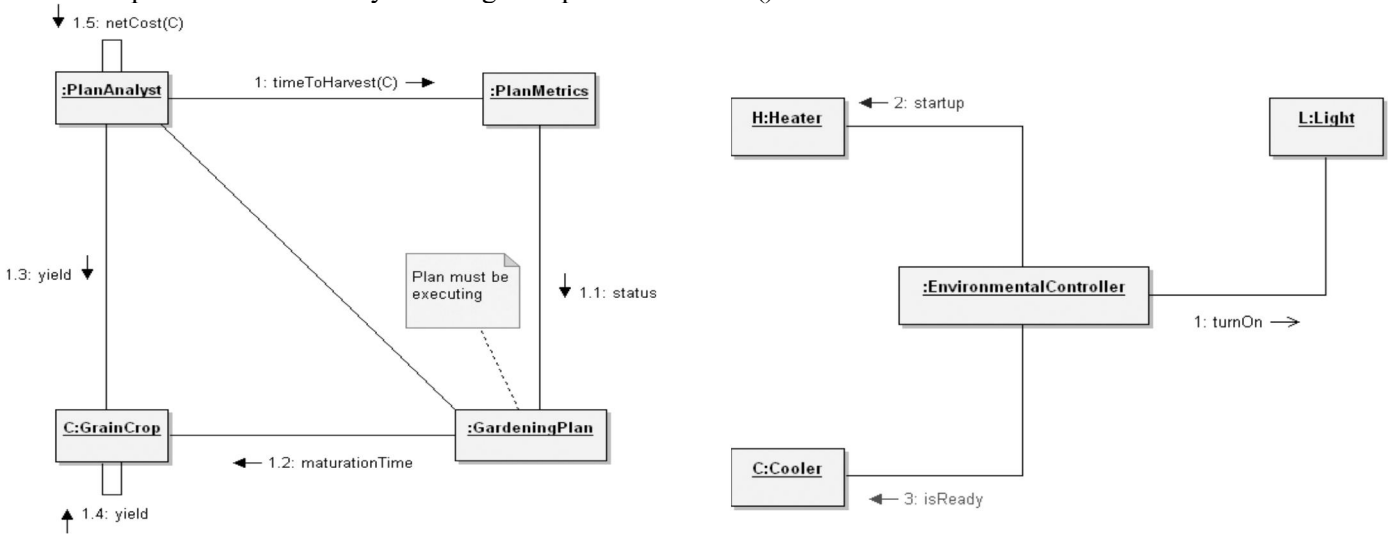
Essentials: Objects, Links, and Messages

A link may exist between two objects if and only if there is an association between their corresponding classes. The existence of an association between two classes denotes a path of communication (i.e., a link) between instances of the classes, whereby one object may send messages to another. Given object A with a link L to object B, A may invoke any operation that is applicable to B's class and accessible to A; the reverse is true for operations invoked by B on A. We will refer to the object that invokes the operation as the *client* and whichever object provides the operation as the *supplier*. Figure shows an example of a communication diagram for the Hydroponics Gardening System. The intent of this diagram is to illustrate the interaction for the execution of a common system function, namely, the determination of a predicted net cost-to-harvest for a specific crop. As shown in Figure, we may adorn a link with one or more messages. We indicate the direction of a message by adorning it with a directed line, pointing to the destination object. An operation invocation is the most common kind of message (the other type would be a signal). An operation invocation may include actual parameters that match the signature of the operation, such as the timeToHarvest message that appears in Figure.

Essentials: Sequence Expressions

Carrying out the predicted net cost-to-harvest system function requires the collaboration of several different objects. To show an explicit ordering of events, we prefix a sequence number (starting at 1) to a message. This sequence expression indicates the relative ordering of messages. Messages with lower sequence numbers are dispatched before messages with higher sequence numbers. The sequence numbers in Figure specify the order of messages for that example. Using a nested decimal numbering scheme (e.g., 4.1.5.2), we can show how some messages are nested within the next higher-level procedure call. Each integer term indicates the level of nesting within the interaction. Integer terms at the same level indicate the sequence of the messages at that level. In Figure, message 1.3 follows message 1.2, which follows message 1.1, and all are nested calls within the timeToHarvest call activation (i.e., message 1). We see from this diagram that the action of the scenario begins with some PlanAnalyst object invoking the

operation timeToHarvest() on PlanMetrics. Note that the object C is passed as an actual argument to this operation. Subsequently, PlanMetrics calls status() on a certain unnamed GardeningPlan object; our diagram includes a development note indicating that we must check that the given plan is in fact executing. The GardeningPlan object in turn invokes the operation maturationTime() on the selected GrainCrop object, asking for the time the crop is expected to mature. After this selector operation completes, control then returns to the PlanAnalyst object, which then calls yield(), which in turn propagates this operation to the C:GrainCrop object. Control again returns to the PlanAnalyst object, which completes the scenario by invoking the operation netCost() on itself.

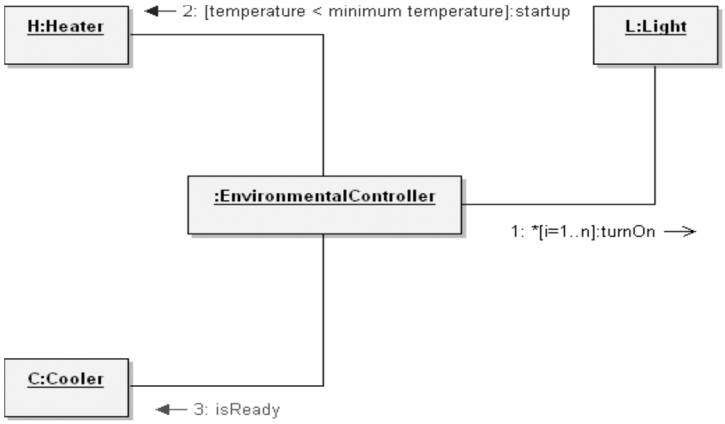


Advanced Concepts: Messages and Synchronization

The message startup() is an example of a simple call and is represented with a directed line with a solid arrowhead. This indicates a synchronous message. In the cases of the startup( ) and isReady( ) messages, the client must wait for the supplier to completely process the message before control can resume. In the case of the message turnOn( ), the semantics are different. This is an example of an asynchronous message, indicated by the open arrowhead. Here the client sends the event to the supplier for processing, the supplier queues the message, and the client then proceeds without waiting for the supplier. Asynchronous message passing is akin to interrupt handling.

Advanced Concepts: Iteration Clauses and Guards

An iteration clause optionally can be added to indicate a series of messages to be sent. The manner in which the iteration clause is specified is up to the individual, although using pseudocode would seem a good choice. Figure shows an iteration clause added to the turnOn( ) message. The adornment is shown as an asterisk followed by the iteration clause in brackets. This example indicates that the turnOn message is to be sent sequentially, 1 to n times. If the messages were to be sent concurrently, the asterisk would be followed by a double bar (i.e., *||[i=1..n]). Guard conditions can also adorn messages. The notation is similar to an iteration clause, but without the asterisk. The guard condition is placed within brackets, as shown in Figure for the startup message. This condition indicates that the message will be executed when the guard condition is true, in this case, when the temperature is below the minimum temperature desired. The manner in which the guard is expressed is up to the individual.


H:Heater
2: [temperature < minimum temperature]:startup
L:Light
:EnvironmentalController
1: *[i=1..n]:turnOn
C:Cooler
3: isReady