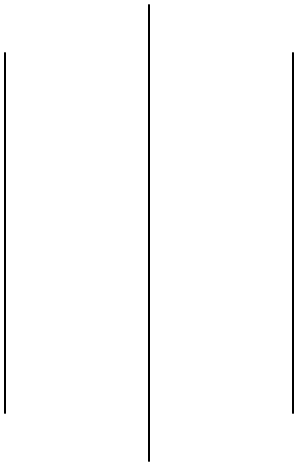
Purbanchal University

College of Information Technology & Engineering (CITE)

Department of Electronics & Computer

Course Manual for Object Oriented Analysis & Design

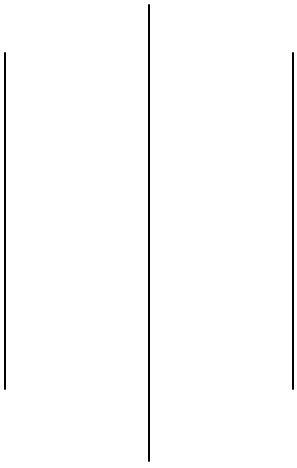
[BCA – V Semester]



Compiled By:

Hari Prasad Aryal [haryal4@gmail.com]

Hari Prasad Pokhrel [hpokhrel24@gmail.com]



Kathmandu, Nepal

October 2011

Chapter 1

Complexity

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.

Software Systems: Software systems are not any different from other systems with respect to these characteristics. Thus, they are also embedded within some operational environment, and perform operations which are clearly defined and distinguished from the operations of other systems in this environment. They also have properties which emerge from the interactions of their components and/or the interactions of themselves with other systems in their environment. A system that embodies one or more software subsystems which contribute to or control a significant part of its overall behavior is what we call a software intensive system. As examples of complex software-intensive systems, we may consider stock and production control systems, aviation systems, rail systems, banking systems, health care systems and so on.

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

1.1 The Inherent Complexity of Software

The Properties of Complex and Simple Software Systems: Software may involve elements of great complexity which is of different kind.
Some software systems are simple.

- These are the largely forgettable applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation.
- Such systems tend to have a very limited purpose and a very short life span.
- We can afford to throw them away and replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality, Such applications are generally more tedious than difficult to develop; consequently, learning how to design them does not interest us.

Some software systems are complex.

- The applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries; and

systems for the command and control of real-world entities, such as the routing of air or railway traffic.

- Software systems such as world of industrial strength software tend to have a long life span, and over time, many users come to depend upon their proper functioning.
- The frameworks that simplify the creation of domain-specific applications, and programs that mimic some aspect of human intelligence.
- Although such applications are generally products of research and development they are no less complex, for they are the means and artifacts of incremental and exploratory development.

Why Software is inherently Complex

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements; the complexity of the problem domain, the difficultly of managing the developmental process, the flexibility possible through software and the problems of characterizing the behavior of discrete systems.

1. The complexity of the problem domain

- Complex requirements
- Decay of system

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.

Consider the requirement for the electronic systems of a multi-engine aircraft, a cellular phone switching system or a cautious (traditional) robot. The row functionality of such systems is difficult enough to comprehend. External complexity usually springs from the impedance mismatch that exists between the users of a system and its developers. Users may have only vague ideas of what they want in a software system. Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the system. A further complication is that the requirement of a software system is often change during its development. Once system is installed, the process helps developers master the problem domain, enabling them to ask better questions that illuminate the done existing system every time its requirements change because a large software system is a capital investment. It is software maintenance when we correct errors, evolution when we respond to changing environments and preservations, when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation.

2. The Difficulty of Managing the Development Process

- Management problems
- Need of simplicity

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 row materials, few such standards exist in the software industry.

4. The problem of characterizing the behavior of discrete systems

- Numerous possible states
- Difficult to express all states

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.

Within a large application, there may be hundreds or even thousands of variables well as more than one thread of control. The entire collection of these variables as well as their current values and the current address within the system constitute the present state of the system with discrete states. Discrete systems by their very nature have a finite numbers of possible states.

The Consequences of Unrestrained Complexity

The more complex the system, the more open it is to total breakdown. Rarely would a builder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition the software crisis, but frankly, a malady that has carried on this long must be called normal. Sadly, this crisis translates into the squandering of human resources - a most precious commodity - as well as a considerable loss of opportunities. There are simply not enough good developers around to create all the new software that users need. Furthermore, a significant number of the developmental personnel in any given organization must often be dedicated to the maintenance or preservation of geriatric software. Given the indirect as well as the direct contribution of software to the economic base of most industrialized countries, and considering the ways in which software can amplify the powers of the individual, it is unacceptable to allow this situation to continue.

1.2 The structure of Complex Systems

Examples 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.

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 can not be done by made of divisions which in turn contain branches which in turn encompass local offices and so on.

The five Attributes of a complex system: There are five attribute common to all complex systems. They are as follows:

1. **Hierarchical and interacting subsystems**
   Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elementary components is reached.
2. **Arbitrary determination of primitive components**
   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. **Stronger intra-component than inter-component link**
   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. **Combine and arrange common rearranging subsystems**
   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. **Evolution from simple to complex systems**

   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.

Booch has identified five properties that architectures of complex software systems have in common.

Firstly, every complex system is decomposed into a hierarchy of subsystems. This decomposition is essential in order to keep the complexity of the overall system manageable. These subsystems, however, are not isolated from each other, but interact with each other.

Very often subsystems are decomposed again into subsystems, which are decomposed and so on. The way how this decomposition is done and when it is stopped, i.e. which components are considered primitive, is rather arbitrary and subject to the architects decision.

The decomposition should be chosen, such that most of the coupling is between components that lie in the same subsystem and only a loose coupling exists between components of different subsystem. This is partly motivated by the fact that often different individuals are in charge with the creation and maintenance of subsystems and every additional link to other subsystems does imply an higher communication and coordination overhead.

Certain design patterns re-appear in every single subsystem. Examples are patterns for iterating over collections of elements, or patterns for the creation of object instances and the like.

The development of the complete system should be done in slices so that there is an increasing number of
subsystems that work together. This facilitates the provision of feedback about the overall architecture.

Organized and Disorganized Complexity

Simplifying Complex Systems

- Usefulness of abstractions common to similar activities
  e.g. driving different kinds of motor vehicle
- Multiple orthogonal hierarchies
  e.g. structure and control system
- Prominent hierarchies in object-orientation
  " class structure "
  " object structure "
  e. g. engine types, engine in a specific car

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.

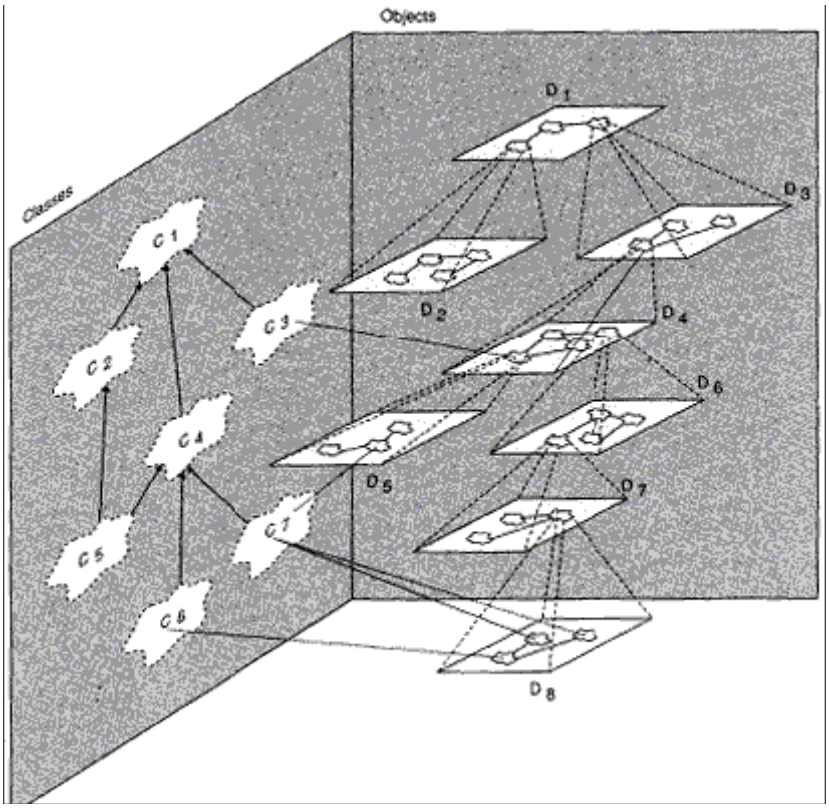


Figure 1.1 : Canonical form of a complex system

The figure 1.1 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.

Approaching a Solution

Hampered by human limitations

- dealing with complexities
- memory
- communications

When we devise a methodology for the analysis and design of complex systems, we need to bear in mind the limitations of human beings, who will be the main acting agents, especially during early phases. Unlike computers, human beings are rather limited in dealing with complex problems and any method need to bear that in mind and give as much support as possible.
Human beings are able to understand and remember fairly complex diagrams, though linear notations expressing the same concepts are not dealt with so easily. This is why many methods rely on diagramming techniques as a basis. The human mind is also rather limited. Miller revealed in 1956 that humans can only remember 7 plus or minus one item at once. Methods should therefore encourage its users to bear these limitations in mind and not deploy overly complex diagrams.
The analysis process is a communication intensive process where the analyst has to have intensive communications with the stakeholders who hold the domain knowledge. Also the design process is a communication intensive process, since the different agents involved in the design need to agree on decompositions of the system into different hierarchies that are consistent with each other.

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.

1.3 Bringing Order to chaos

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

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.

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'.

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.

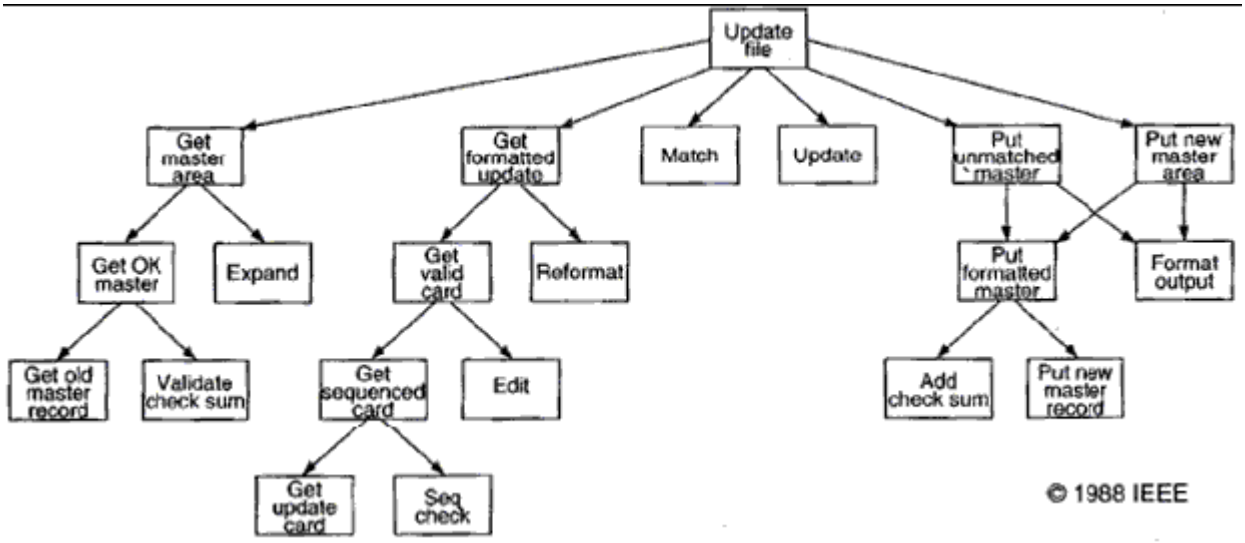


Figure 1.2: Algorithmic decomposition

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.

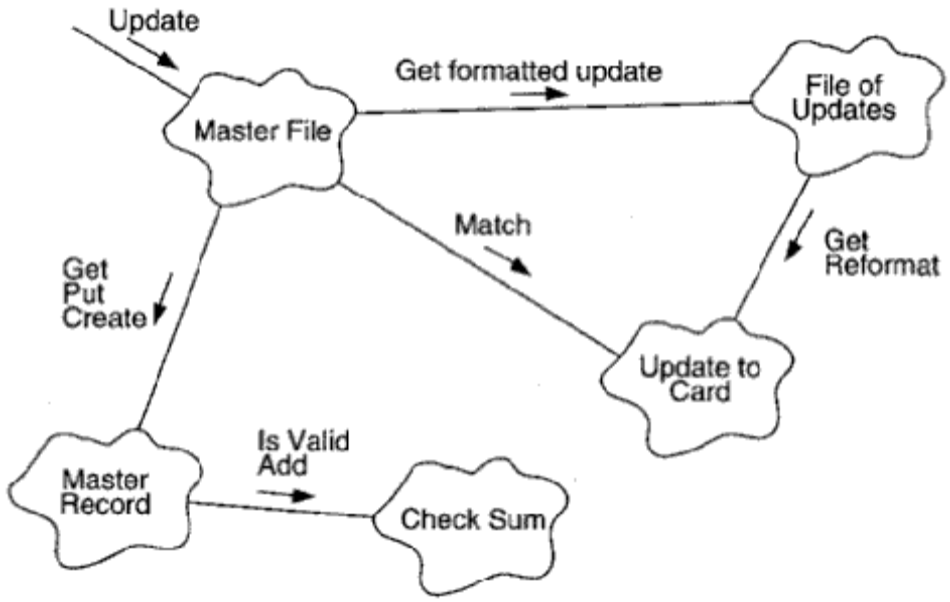


Figure 1.3: Object Oriented decomposition

Algorithmic versus object oriented decomposition: The algorithmic view highlighst 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.

1.4 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 fordoing 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.

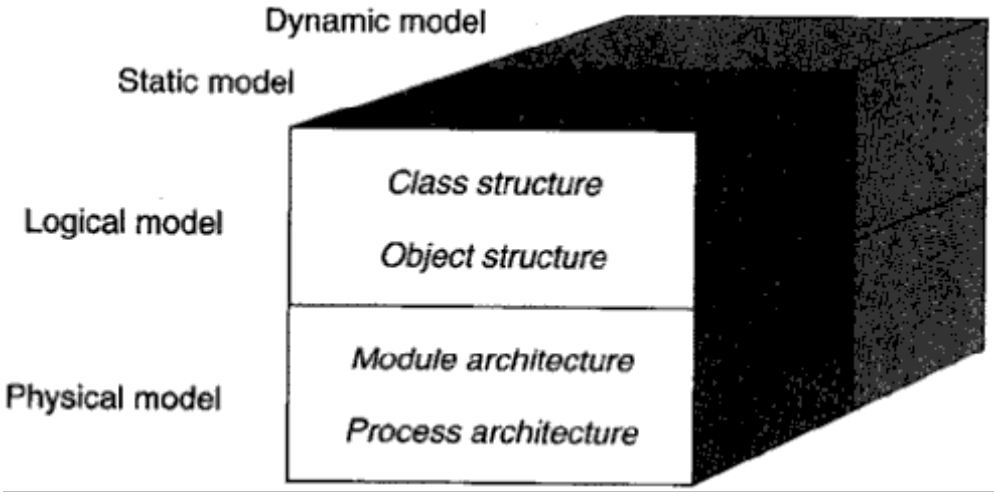


Figure 1.4: Models of object oriented development

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.

Chapter 2
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.

2.1 The Evolution of the object Model

- The shift in focus from programming-in-the-small to programming-in-the-large.
- The evolution of high-order programming languages.
- New industrial strength software systems are larger and more complex than their predecessors.
- Development of more expressive programming languages advances the decomposition, abstraction and hierarchy.
- Wegner has classified some of more popular programming languages in generations according to the language features.

The generation of programming languages

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.

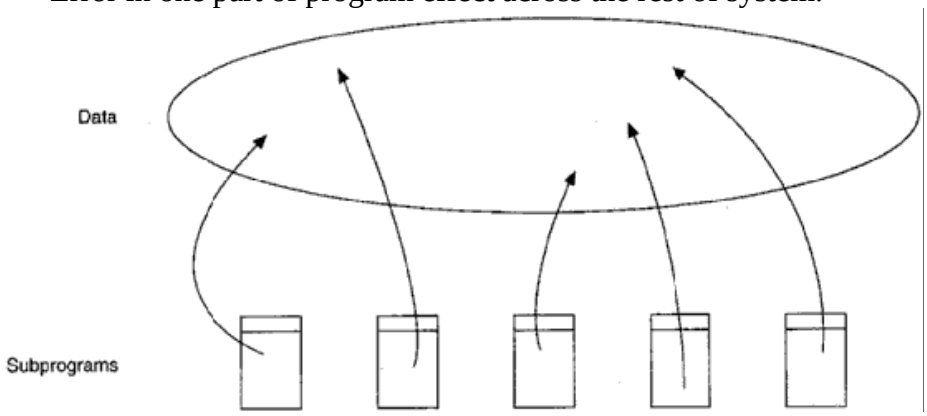


Fig 2.1: The Topology of First- and Early Second-Generation Programming Languages

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.

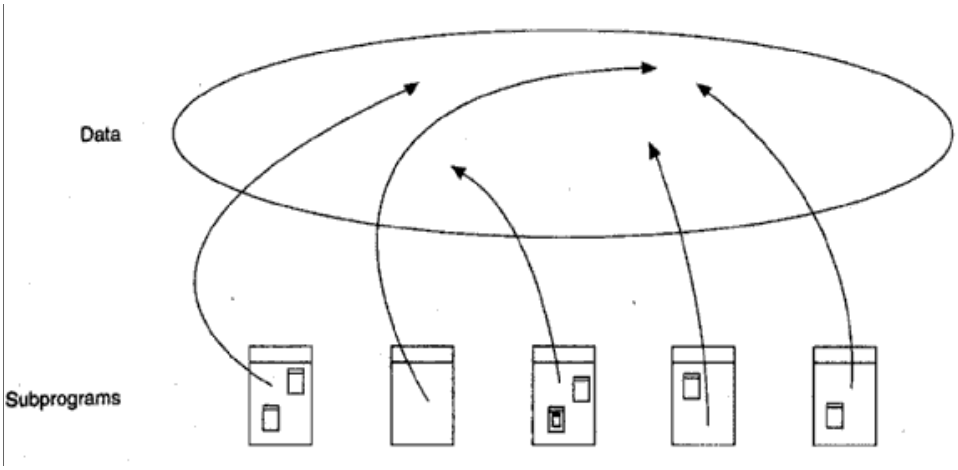


Fig 2.2: The Topology of Late Second- and Early Third-Generation Programming Languages

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.

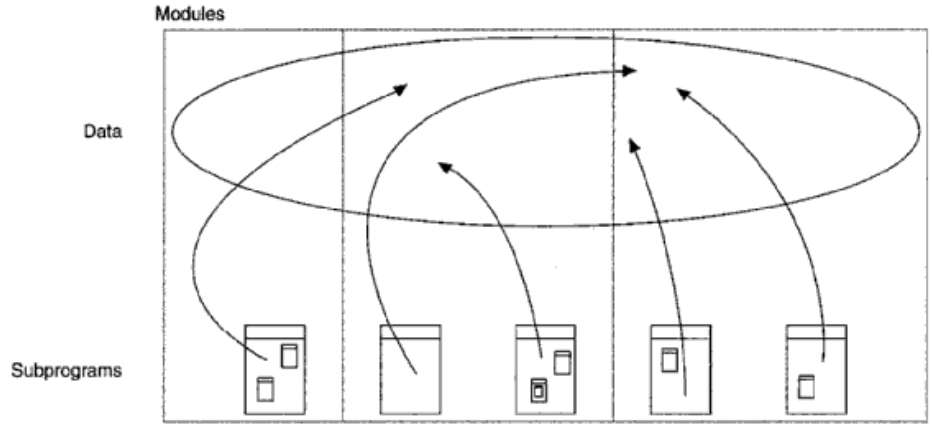


Fig 2.3: The Topology of Late Third-Generation Programming Languages

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.

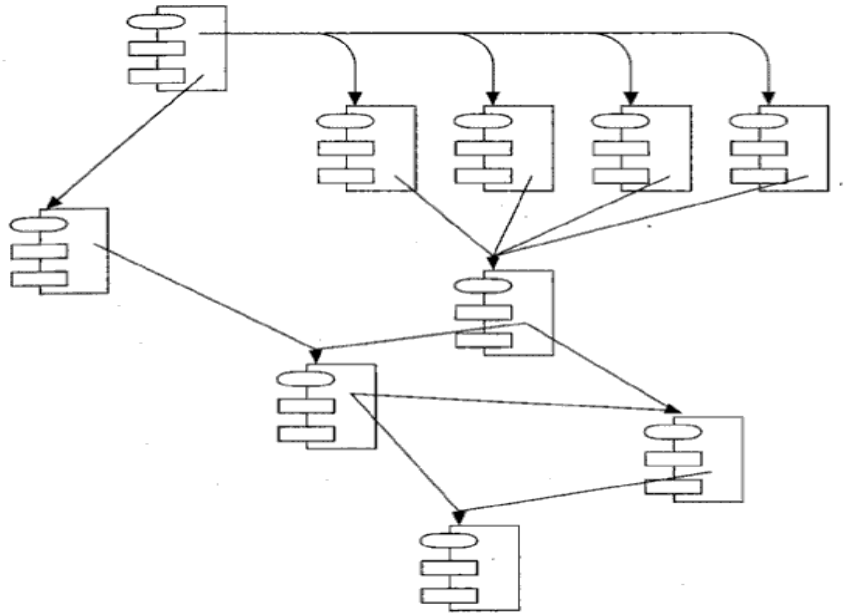


Fig 2.4: The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

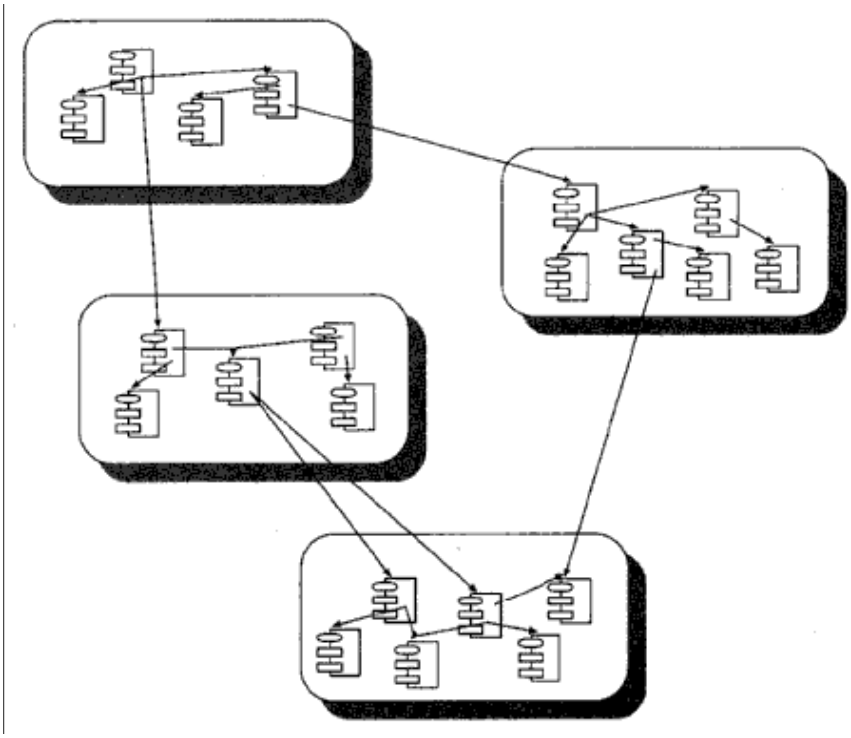


Fig 2.5: The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

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.

2.2 Elements of Object Model

Kinds of Programming Paradigms: According to Jenkins and Glasgow, most programmers work in one language and use only one programming style. They have not been exposed to alternate ways of thinking about a problem. Programming style is a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear.

There are five main kinds of programming styles:

1. Procedure oriented – Algorithms for design of computation
2. Object oriented – classes and objects
3. Logic oriented – Goals, often expressed in a predicate calculus
4. Rules oriented – If then rules for design of knowledge base
5. Constraint orient – Invariant relationships.

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

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

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.

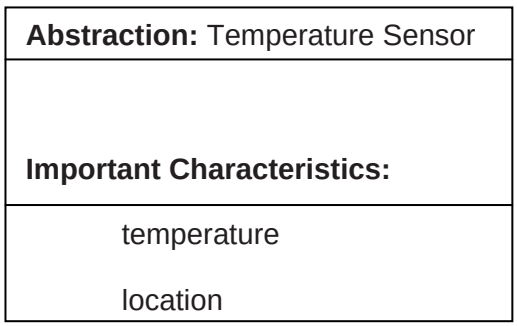
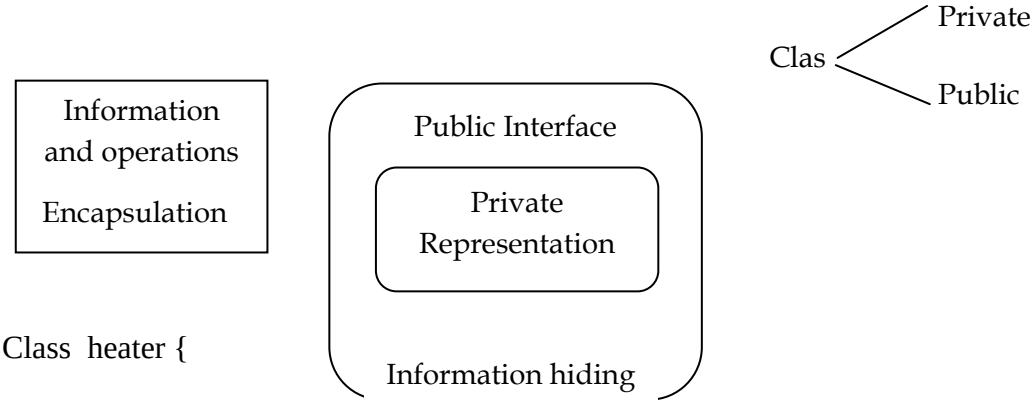


Figure 2.6: Abstraction of a Temperature Sensor

Encapsulation

The act of grouping data and operations into a single object.



Class heater {

Public:

heater (location):

~ heater ( ):

void turnon ( );

void turnoff ( );

Boolean ison () const

private:

Abstraction: Heater
Important Characteristics: location
status

Figure 2.7: Abstraction of a Heater

Modularity

The act of partitioning a program into individual components is called modularity. It is reusable component which reduces complexity to some degree. Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program. In some languages, such as Smalltalk, there is no concept of a module, so the class forms the only physical unit of decomposition. Java has packages that contain classes. In many other languages, including Object Pascal, C++, and Ada, the module is a separate language construct and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system; we place these abstractions in modules to produce the system's physical architecture. 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.
- A poor design is to define each message class in its own module; so difficult for users to find the classes they need. Sometimes modularization is worse than no modulation at all.
- Developer must balance: desire to encapsulate abstractions and need to make certain abstractions visible to other modules.
- Principles of abstraction, encapsulation and modularity are synergistic (having common effect)

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.

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. An earlier section described our abstraction of a very generalized growing plan. Different kinds of crops, however, demand specialized growing plans. For example, the growing plan for all fruits is generally the same but is quite different from the plan for all vegetables, or for all floral crops. Because of this clustering of abstractions, it is reasonable to define a standard fruitgrowing plan that encapsulates the behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert that FruitGrowingPlan "is a" kind of GrowingPlan.

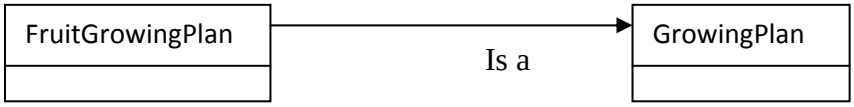


Fig 2.8: Class having one superclass (Single Inheritance)

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.

As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses. This is why we often speak of inheritance as being a generalization/specialization hierarchy. Superclasses represent generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden.

Examples of Hierarchy: Multiple Inheritance

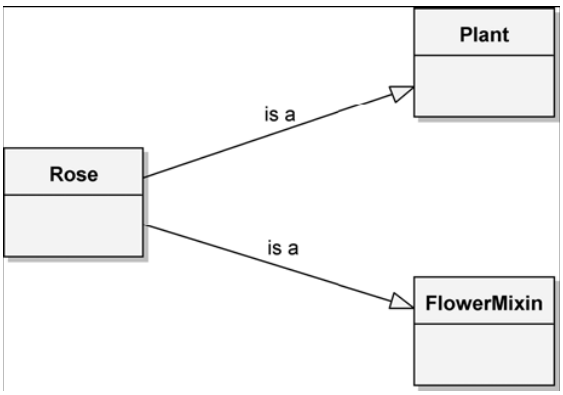


Figure 2.9: The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)

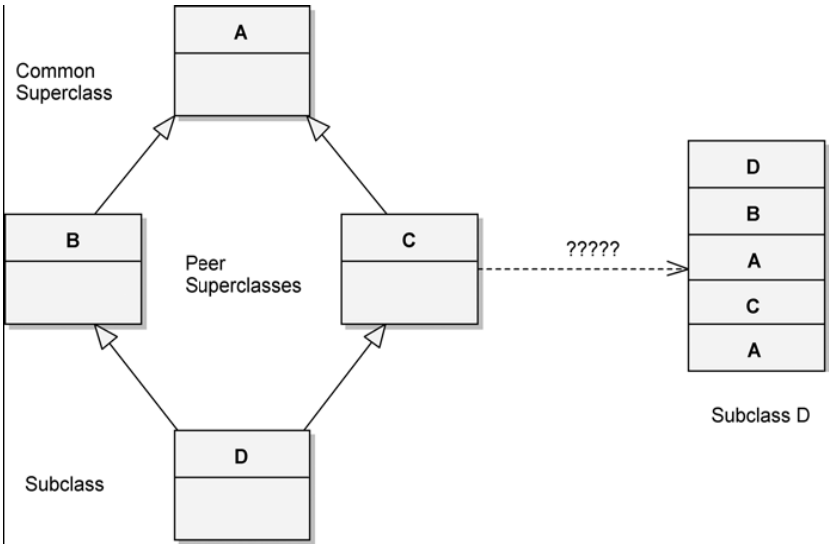


Figure 2.10: The Repeated Inheritance

Repeated inheritance occurs when two or more peer superclasses share a common superclass.

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.

Aggregation raises the issue of ownership. Our abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted). In other words, the lifetime of a garden and its plants are independent. In contrast, we have decided that a GrowingPlan object is intrinsically associated with a Garden object and does not exist independently. Therefore, when we create an instance of Garden, we also create an instance of GrowingPlan; when we destroy the Garden object, we in turn destroy the GrowingPlan instance.

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.

A given programming language may be strongly typed, weakly typed, or even untyped, yet still be called object-oriented. A strongly typed language is one in which all expressions defined in super class are guaranteed to be type consistent. When we divide distance by time, we expect some value denoting speed, not weight. Similarly, dividing a unit of force by temperature doesn't make sense, but dividing force by mass does. These are both examples of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions.

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).

Polymorphism is a condition that exists when the features of dynamic typing and inheritance interact. Polymorphism represents a concept in type theory in which a single name (such as a

variable declaration) may denote objects of many different classes that are related by some common superclass. The opposite of polymorphism is monomorphism, which is found in all languages that are both strongly and statically typed.

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.

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.

Traditional Programming Languages usually address only the first three kind of object persistence. Persistence of last three kinds is typically the domain of database technology. Introducing the concept of persistence to the object model gives rise to object oriented databases. In practice, such databases build upon some database models (Hierarchical, network relational). Database queries and operations are completed through the programmer abstraction of an object oriented interface. Persistence deals with more than just the lifetime of data. In object oriented databases, not only does the state of an object persist, but its class must also transcend only individual program, so that every program interprets this saved state in the same way.
In most systems, an object once created, consumes the same physical memory until it classes to exist. However, for systems that execute upon a distributed set of processors, we must sometimes be concerned with persistence across space. In such systems, it is useful to think of objects that can move from space to space.

2.3 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.

Chapter 3
Classes and Objects

When we use object-oriented methods to analyze or design a complex software system, our basic building blocks are classes and objects.

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

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

Classes represent groups of objects which have the same behavior and information structures.

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

3.1 The nature of an object

The ability to recognize physical objects is a skill that humans learn at a very early age. From the perspective of human, cognition, an object is any of the following.

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

Informally, object is defined as a tangible entity that exhibits some well defined behavior. During software development, some objects such as inventions of design process whose collaborations with other such objects serve as the mechanisms that provide some higher level behavior more precisely.

An object represents an individual, identifiable item, until or entity either real or abstract, with a well defined role in the problem domain. E.g. of manufacturing plant for making airplane wings, bicycle frames etc. A chemical process in a manufacturing plant may be treated as an object; because it has a crisp conceptual boundary interacts with certain other objects through a well defined behavior. Time, beauty or colors are not objects but they are properties of other objects. We say that mother (an object) loves her children (another object).

An object has state, behavior and identify; the structure and behavior similar objects are defined an their common class, the terms instance and object 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.

A property is a distinctive characteristic that contributes to making an object uniquely that object properties are usually static because attributes such as these are unchanging and fundamental to the nature of an object. Properties have some value. The value may be a simple quantity or it might denote another object. The fact that every object has static implies that every object has state implies that every object takes up some amount of space be it in the physical world or in computer memory.

We may say that all objects within a system encapsulate some state and that all of the state within a system is encapsulated by objects. Encapsulating the state of an object is a start, but it is not enough to allow us to capture the full intent of the abstractions we discover and invent during development

e.g. consider the structure of a personnel record in C++ as follows

```
struct   personnelRecord{
char     name[100];
int      socialsecurityNumber;
char     department[10];
float    salary;
};
```

This denotes a class. Objects are as personnel Record Tom, Kaitlyn etc are all 2 distinct objects each of which takes space in memory. Own state in memory class can be declared as follows.

```
Class   personnelrecord{
public: char*employeename()const;
 int      SSN()  const;
 char*  empdept         const;
 protected:
char     name[100];
int      SSN;
 char    department[10];
 float   salary;
};
```

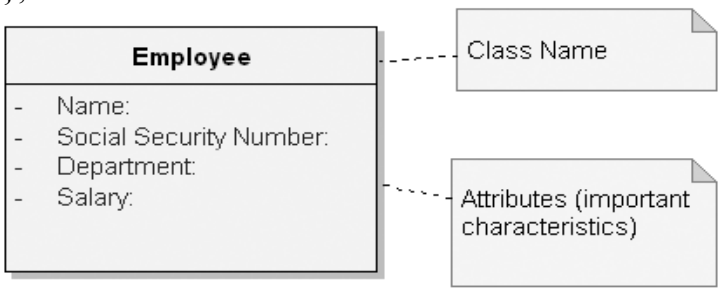


Figure 3–1 Employee Class with Attributes

Figure 3–2 Employee Objects Tom and Kaitlyn

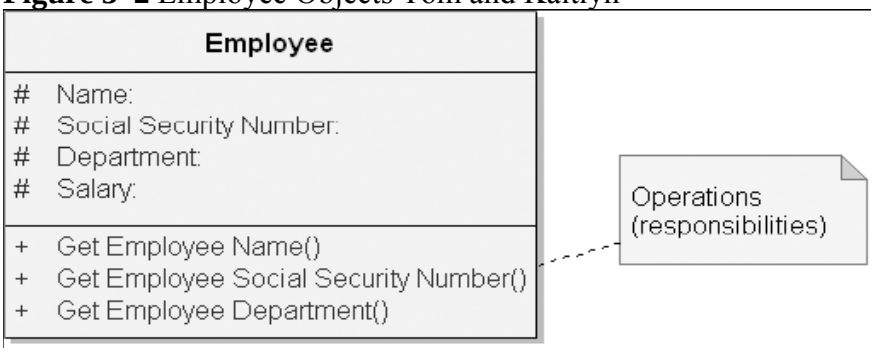


Figure 3–3 Employee Class with Protected Attributes and Public Operations

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

Behavior

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.

e.g. consider the declaration of queue in C++

```
Class Queue{
public:
Queue();
Queue(constQueue);
virtual ~Queue();
virtual Queue&operator = (ConstQueue);
Virtual int operator == (constQueue&)const;
int operator = (constQueue)const;
virtual voidclear();
Virtual voidappend(constvoid*);
virtual voidPOP();
virtual void remove (int at);
virtual int length();
virtual int isempty ( ) const;
```

```
virtual const void * Front ( ) const;
 virtual int location (const void*);
protected..
};

queue a, b;
a. append (& Tom);
a.append (& Kaitlyn);
b = a;
a. pop( );
```

Operations

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

- Modifier: An operation that alters the state of an object.
- Selector: An operation that accesses the state of an object but does not alter the state.
- Iterator: An operation that permits all parts of an object to be accessed in some well defined order. In queue example operations, clear, append, pop, remove) are modifies, const functions (length, is empty, front location) are selectors.
- 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.

Identity

Identity is that property of an object which distinguishes it from all other objects.
Consider the following declarations in C++.

```
struct point {
int x;
int y;
point ( ) : x (0), y (0){}
 point (int x value, int y value) : x (x value), (y value) {}
};

Next we provide a class that denotes a display items as follows.
Class DisplayItem{
Public: DisplayItem ();
displayitem (const point & location);
 virtual ~ displayitem ().
Virtual void draw();
Virtual void erase();
Virtual void select();
Virtual void Unselect ( );
virtual void move (const point & location);
int isselected ();
virtual void unselect ();
 virtual void move (const point & location);
```

```
 virtual void point location ( ) const;
int isunder (const point & location) const;
Protected …….
};
```

To declare instances of this class:

```
displayItem  item1;
item2 = new displayItem (point (75, 75);
Display item *  item 3 = new Display Item (point (100, 100)) ;
display item *  item 4 = 0
```

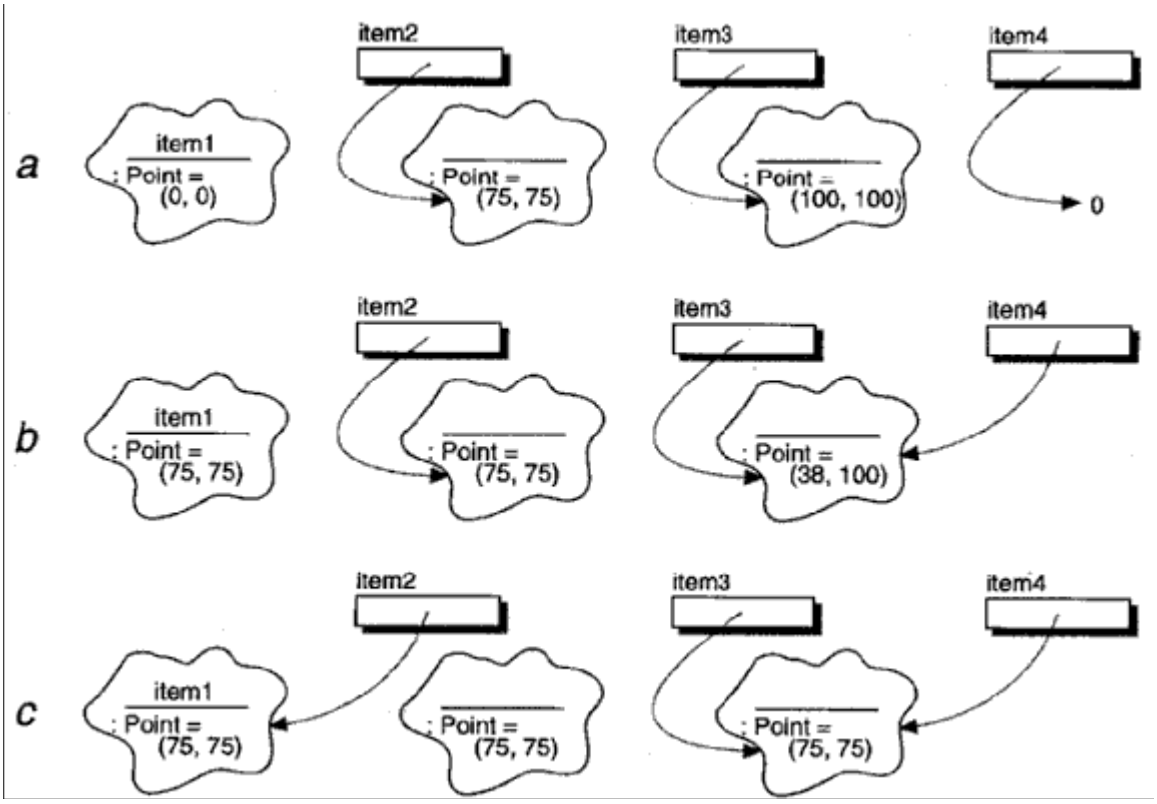


Figure 3–4 Object Identity

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.

Object life span

The lifeline of an object extends from the time it is first created (and this first consumes space) until that space is recalled, whose purpose is to allocate space for this object and establish an

initial stable state. Often objects are created implicitly in C++ programming an object by value creates a new objection the stack that is a copy of the actual parameters.
In languages such as smalltalk, an object is destroyed automatically as part of garbage collection when all references to it have been lost. In C++, objects continuous exist and consume space even if all references to it are lost. Objects created on the stack are implicitly destroyed wherever control panels beyond the block in which the object can declared. Objects created with new operator must be destroyed with the delete operator. In C++ wherever an object is destroyed either implicitly or explicitly, its destructor is automatically involved, whose purpose is to declared space assigned to the object and its part.

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.
In other words, we may say that 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.
Most interesting objects play many different roles during their lifetime such as:

- 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.

Objects as Machines

The existence of state within an object means that the order in which operations are invoked is important. This gives rise to the idea that each object is like a tiny, independent machine. Continuing the machine metaphor, we may classify objects as either active or passive. An active object is one that encompasses its own thread of control, whereas a passive object does not. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated on by another object. Passive objects, on the other hand, can undergo a state change only when explicitly acted on. In this manner, the active objects in our system serve as the roots of control. If our system involves multiple threads of control, we will usually have multiple active objects. Sequential systems, on the other hand, usually have exactly one active object, such as a main object responsible for managing an event loop that dispatches messages. In such architectures, all other objects are passive, and their behavior is ultimately triggered by messages from the one active object. In other kinds of sequential system architectures (such as transaction-processing systems), there is no obvious central active object, so control tends to be distributed throughout the system's passive objects.

3.2 Relationship 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. Two kinds of objects relationships are links and aggregation.

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.

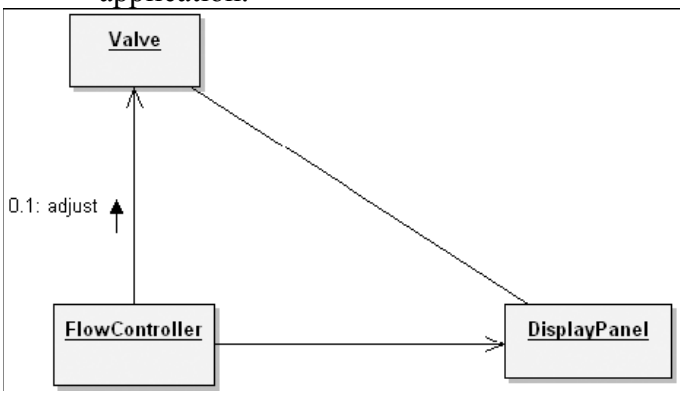


Figure 3–5 Links
In the above figure, FlowController acts as a controller object, DisplayPanel acts as a server object, and Valve acts as a proxy.

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.

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.

There are clear trade-offs between links and aggregation. Aggregation is sometimes better because it encapsulates parts as secrets of the whole. Links are sometimes better because they permit looser coupling among objects.

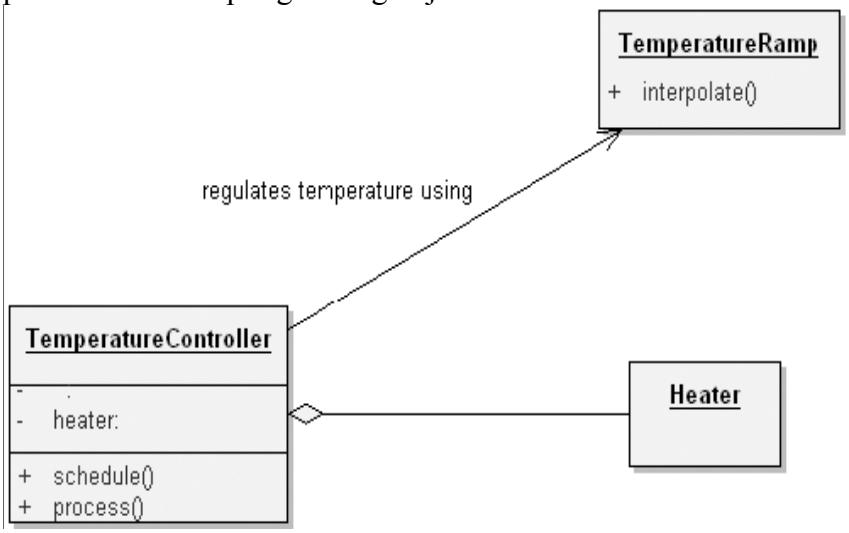


Figure 3–6 Aggregation

3.3 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

3.4 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 three basic kinds of class relationships.

- The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower.
- The second is whole/part, which denotes a "part of" relationship. A petal is not a kind of a flower; it is a part of a flower.
- The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers. As another example, roses and candles are largely independent classes, but they both represent things that we might use to decorate a dinner table.

Association

Of the different kinds of class relationships, associations are the most general. The identification of associations among classes is describing how many classes/objects are taking part in the relationship. As an example for a vehicle, two of our key abstractions include the vehicle and wheels. As shown in Figure 3–7, we may show a simple association between these two classes: the class Wheel and the class Vehicle.

Figure 3–7 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

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.

Space probe (spacecraft without people) report back to ground stations with information regarding states of important subsystems (such as electrical power & population systems) and different sensors (such as radiation sensors, mass spectrometers, cameras, detectors etc), such relayed information is called telemetry data. We can take an example for Telemetry Data for our illustration.

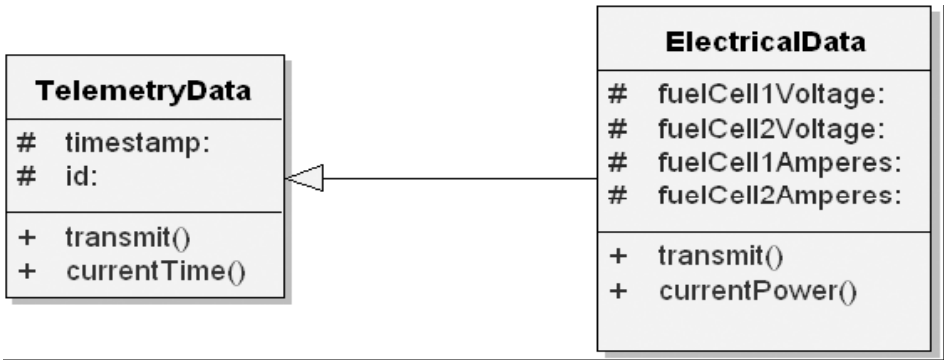


Figure 3–8 ElectricalData Inherits from the Superclass TelemetryData

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

Single Inheritence

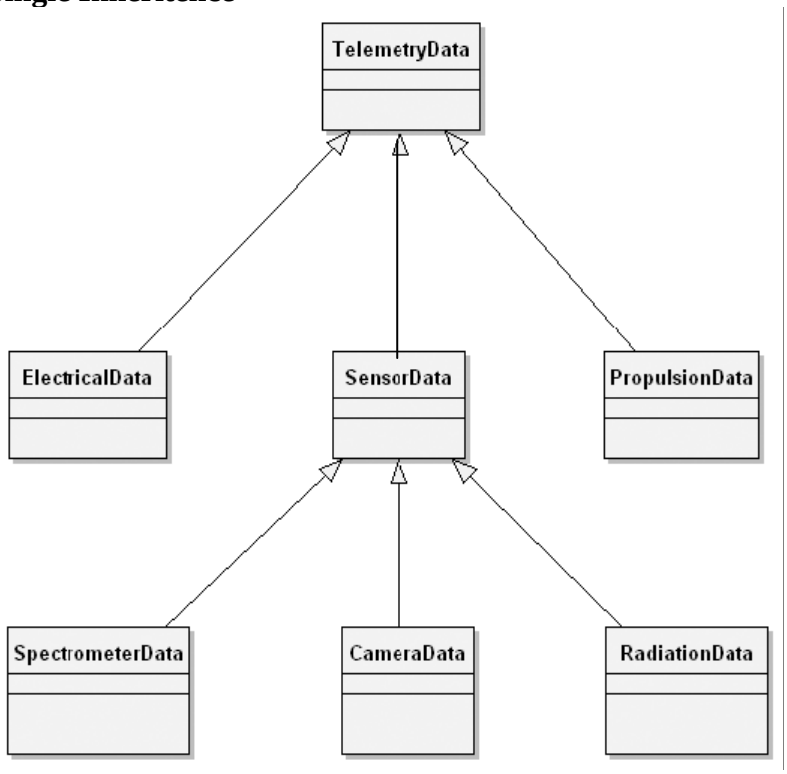


Figure 3–9 Single Inheritance

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

Multiple Inheritence

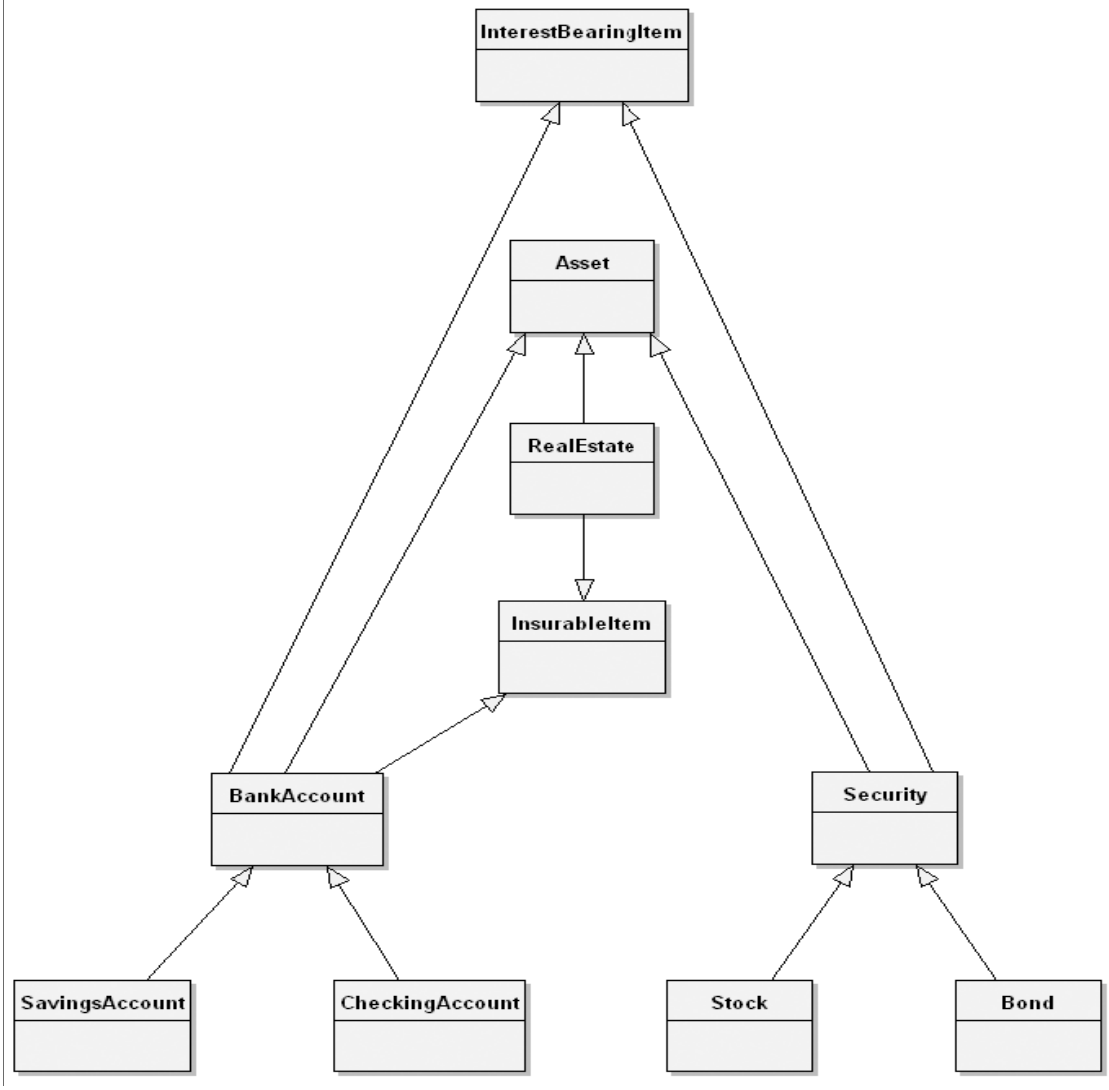


Figure 3–10 Multiple Inheritance

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

Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance. Figure 3–10 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 a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways. With polymorphism, an operation can be implemented differently by the classes in the hierarchy.

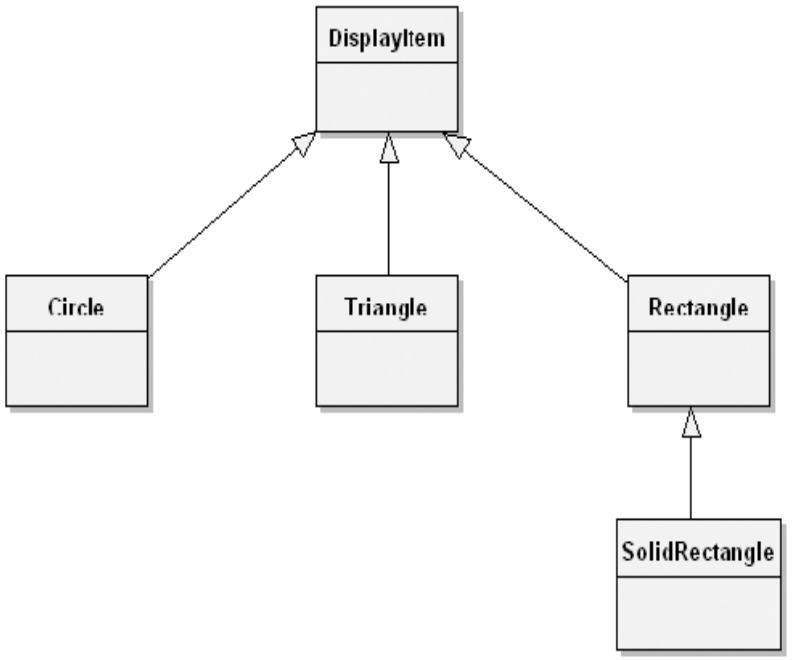


Figure 3-11 Polymorphism

Consider the class hierarchy in Figure 3–11, 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.

Aggregation

We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes. As shown in Figure 3–12, the class TemperatureController denotes the whole, and the class Heater is one of its parts.

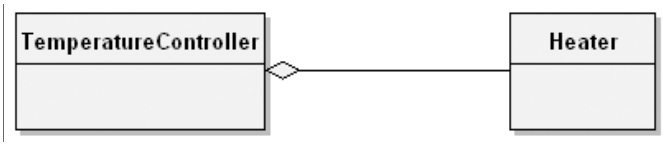


Figure 3–12 Aggregation

Physical Containment

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

Using

Using shows a relationship between classes in which one class uses certain services of another class in a variety of ways. "Using" relationship is equivalent to an association, although the reverse is not necessarily true.

Clients and Suppliers

"Using" relationships among classes parallel the peer-to-peer links among the corresponding instances of these classes. Whereas an association denotes a bidirectional semantic connection, a "using" relationship is one possible refinement of an association, whereby we assert which abstraction is the client and which is the supplier of certain services.

Instantiation

The process of creating a new object (or instance of a class) is often referred to as instantiation.

Genericity

The possibility for a language to provided parameterized modules or types. E.g. List (of: Integer) or List (of: People). There are four basic ways of genericity

- Use of Macros – in earlier versions of C++, does not work well except on a small scale.
- Building heterogenous container class: used by small task and rely upon instance of some distant base class.
- By building generalized container classes as in small task, but then using explicity type checking code to enforce the convention that the contents are all of the same clam, which is asserted when the container object is created used in object Pascal, which are strongly typed support heritance but don't support any form of parameterized class.
- Using parameterized class (Also known as generic class) is one that serves as a temperature for other classes & template that may be parameterized by other classes, objects and or operations. A parameterized class must be instantiated (i.e. parameters must be filled in) before objects can be created.

Metaclass

Metaclass is a class whose instances are themselves classes. Small task and CLOS support the concept of a metaclass directly, C++ does not. A class provides an interface for the programmer to interface with the definition of objects. Programmers can easily manipulate the class.

Metaclass is used to provide class variables (which are shared by all instances of the class) and operations for initializing class variables and for creating the metaclass's single instance.

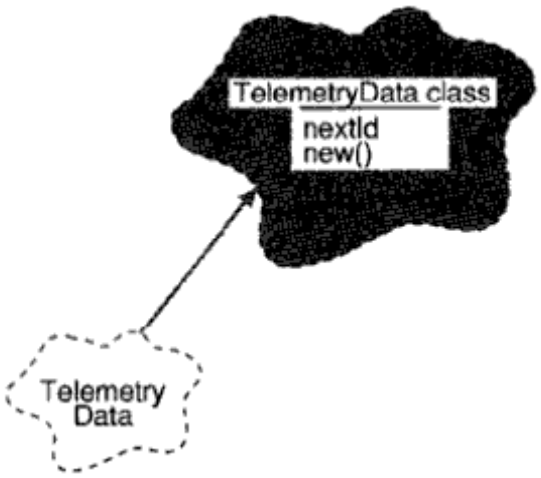


Figure 3-13 Metaclass

As shown in figure 3-13, a class variable next ID for the metaclass of telemetry data can be defined in order to assist in generating district ID's up on the creation of each instance of telemetry data. Similarly, an operation can be defined for creating new instances of the class, which perhaps generates them from some pre-allocated pool of storage. In C++, and destructors serve the purpose of metaclass creation operations. Member function and member objects as static in C++ are shared by all instances of class in C++. Static member's objects and static member function of C++ are equivalent to small task's metaclass operations.

Importance of proper classification

Classification is the means where by 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 18th 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

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 there after, the child acquires skill in classifying these objects, first using basic category such as dogs, cats and toys. Later the child develops more general categories (such as animals). In criteria for sameness among objects specifically, one can divide objects into disjoint sets depending upon the presence or absence of a particular property. Properties may denote more than just measurable characteristics. They may also encompass observable behaviors e.q. bird can fly but others can not is one property.

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.

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

Identifying key abstractions

Finding key abstractions

A key abstraction is a class or object that forms part of the vocabulary of the problem domain. The primary value of identifying such abstractions is that they give boundaries to our 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 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 grainsize 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
Finding Mechanism
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)

Key abstractions reflect the vocabulary of the problem domain and mechanisms are the soul of the design. Idioms are part of a programming culture. An idiom is an expression peculiar to a certain programming language. e.g. in CLOS, no programmer use under score in function or variable names, although this is common practice in ada.
A frame work is collection of classes that provide a set of service for a particular domain. A framework exports a number of individual classes and mechanisms which clients can use.

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.

Chapter - 4
The Notation

A diagram simply captures a statement of a system's behavior (for analysis), or the vision and details of an architecture (for design). A well-defined and expressive notation is important to the process of software development. A standard notation makes it possible for an analysis or developer to describe a scenario e.g. electrical circuit is same for all electrical engineer all round the world and also makes it possible to eliminate tension of checking the consistency and correctness of these decisions by using automated tools.

4.1 Elements of the Notation

The need for multiple views:

It is possible to capture all the details of a complex software system in just one view. One must understand the taxonomic structure of the class objects, the inheritance mechanisms used, the individual behaviors of object and the dynamic behavior of the system as a whole. So, for a given project, the products of analysis and design are expressed through different models as shown in figure.

These models help developer to make decisions during the analysis of a system. The whole notation may not be used a proper subset of this notation is sufficient to express the semantics of a large percentage of analysis and design. This subject called Booch Lite notation. One should apply only those elements of the notation that are necessary to convey the intended meaning. E.g. on a blue print, an architect may show general location of light switch in a room but its exact location if found out after the house has been framed. The notation is largely language independent.

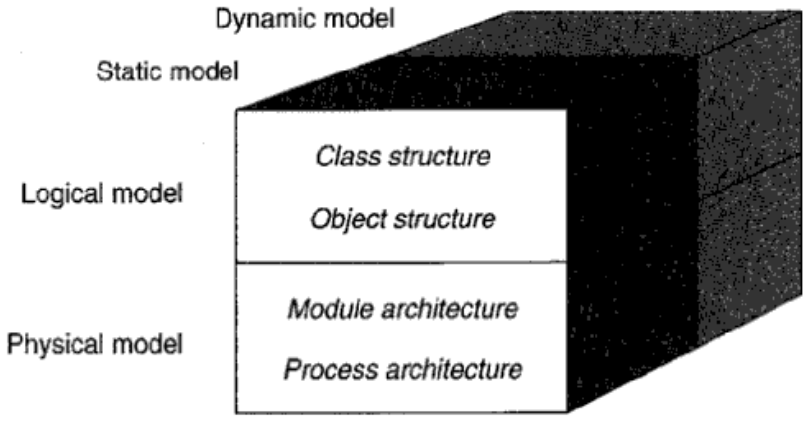


Figure 5-1: The Models of Object-Oriented Development

Models and views

A shown is figure we may capture our analysis and design decisions regarding classes and objects and their collaborations according to two dimensions are necessary to specify the structure and behavior of an object oriented system. For each dimension, a number of diagrams are defined to denote a view of system's models. E.g. consider an application comprising several hundred classes. The classes form part of the application's model. We might view this model through several class diagrams.

Logical versus physical models

The logical view of a system serves to describe the existence and meaning of key abstractions and mechanisms that define the system architecture. The physical model of a system describes the concrete software and hardware imposition of the system's context or implementation.

During analysis, we must desired behavior of the system.

- What is the desired behavior of the system?
- What are the roles and responsibilities of the objects that carry out this behavior in the logical model, object diagrams serve as primary vehicles for describing scenarios. Class diagram is also used to capture abstraction of objects during analysis.

During design, we must address the following central questions relative to the system's architecture:

- What classes exist and how are those classes related?
- What mechanisms are used to regulate how objects collaborate?
- Where should each class and object be declared?

To answer above questions, following diagrams are used.

Class diagrams, object diagrams, process diagrams, all diagrams are largely static.

Static versus dynamic semantics

Dynamic semantics of a problem or its implementations are expressed using state transition diagrams and interactions diagram. Each class may have an associated state transition diagram that indicate the event ordered behavior of the class's instances. A script or interaction diagram is sued in conjunction with an object diagram to show the time or event ordering of messages.

The Role of Tools

Great designs come from great designers not from great tool. Tools simply empower the individual in analysis or design. Tools simply empower the individual in analysis or design. there are some things that tools can do well.

- Tools can ensure that the message is in fact part of object's protocol when we use an object diagram to show a scenario with a message being passed from one object to another. This is consistency checking.
- Constraint checking: A tool can enforce the convention that there are no more than 3 instances of class.
- Completeness checking: A tool can tell us if certain classes or methods of a given class are never used.
- Analysis: A sophisticated tool might tell how long it takes to take a certain operation. Expert system can be used as a tool.

Class diagrams

Class diagram is used to show the existence of classes and their relationships in the logical view of a system. A single class diagram represents a new of the class structure of a system.

During analysis, we use class diagrams to indicate the common roles and responsibilities of the entities that provide the systems behavior. During the design, we use class diagrams to capture the structure of the classes that form the system's architecture.

The two essential elements of a class diagram are classes and their basic relationships. Figure shows the icon which is used to represent class in a class diagram. Its shape is that of a cloud; some call it an amorphous blob.

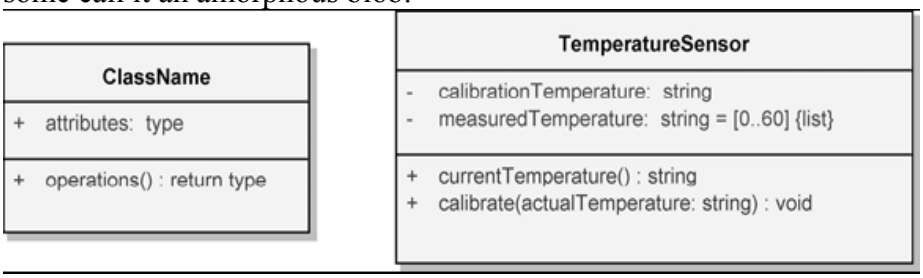


Fig 5-2: A General Class Icon and an Example for the Gardening System

A name is required for each class of the name is particularly long, it can either be elided or the icon magnified every class name must be unique to its enclosing class category. For certain class diagram, it is useful to expose some of the attributes and operations associated with a class. We say "some" because for all but most trivial class, it is clumsy and indeed unnecessary to how all such members in a diagram, even when using rectangular icon.

If we choose to show no such members at all, we may drop the separating line and show only the class name. An attribute denotes a part of an aggregate object and so is used during analysis as well as design to express a singular property of the class. Using the following language independent syntax, an attribute may have a name, a class or both and optionally a default value.

- A attribute name only
- **:** C attribute class only.
- A **:** C attribute name and class
- A **:** C =E attribute name, class and default expression.

An attribute name must be unambiguous in the context of the class.

An operation denotes some service provided by the class. Operations are usually just named when shown inside a class icon and are distinguished from attributes by appending parentheses or by providing the operations complete signature.

- NC – operation name only
- R N (Arguments) – Operation return class, name and formal arguments (if any) an abstract class is one for which no instances may be created such classes is shown as shown in figure:

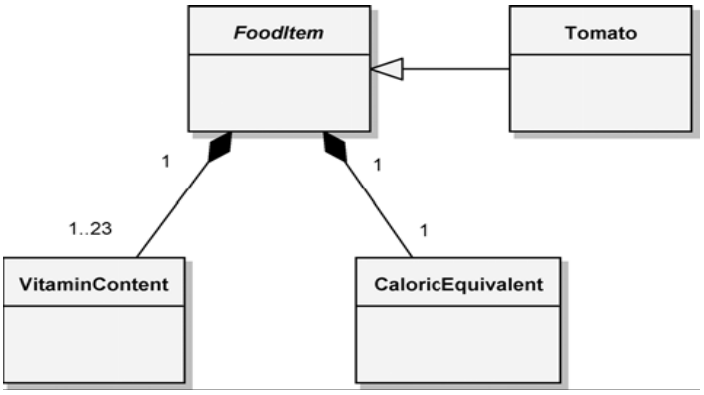


Fig 5-3: Abstract Class Adornment

Class Relationships

The essential connections among classes include association, inheritance, "has" and using relationships whose icons are shown is figure. Each such relationship may include a textual label that documents the name of the relationship or suggests its purpose.

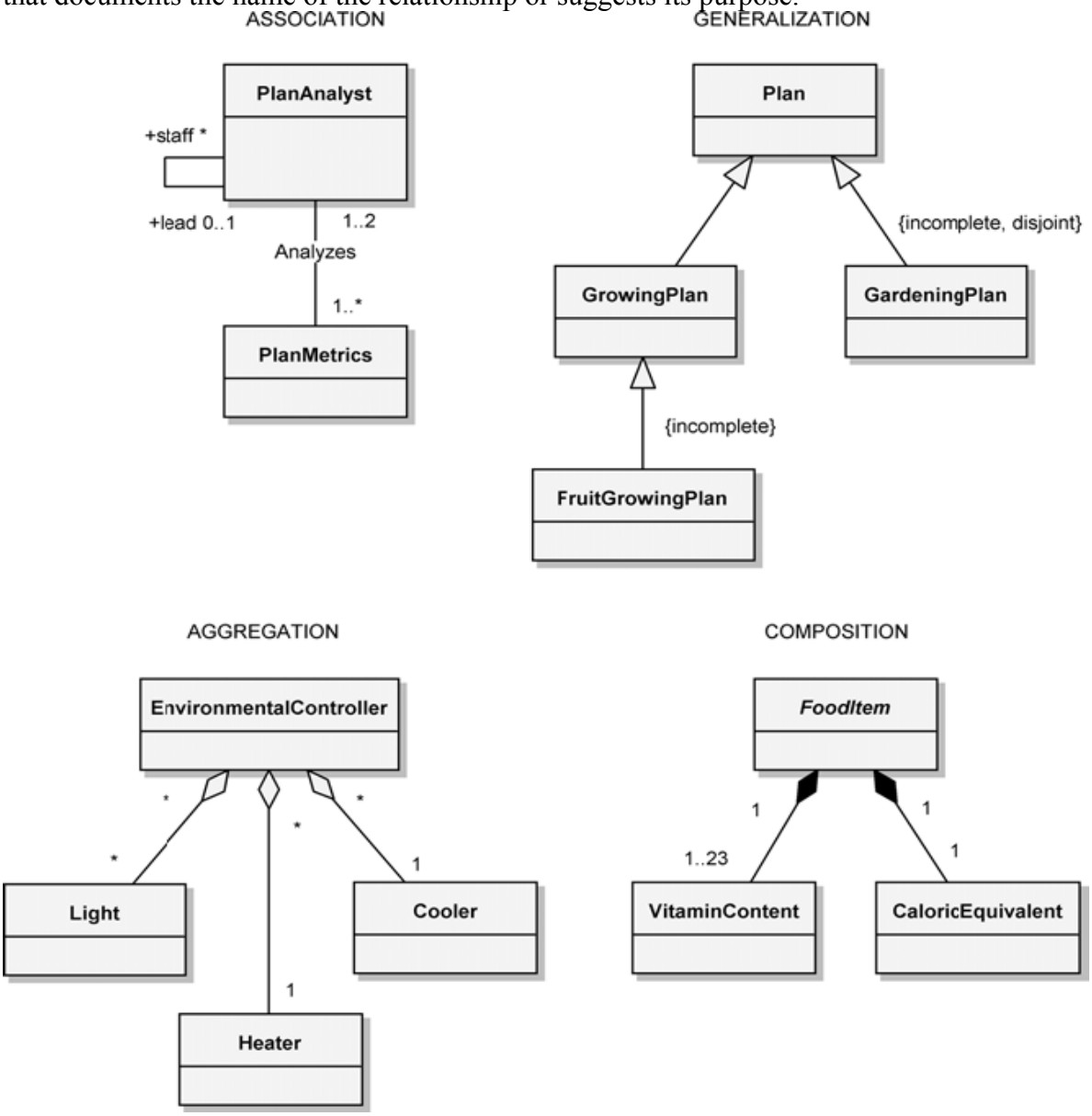


Fig 5 – 4: Class Relationship Icons

Association icon connects two classes and denotes a semantic connection. Associations are often labeled with noun phrases such as employment denoting the nature of the relationship. A class may have association to itself (called a reflexive association). Association may be further adorned with their cardinality using the syntax in the following examples.

- 1 Exactly one
- * Unlimited number
- 0..* Zero or more

- 1..* One or more
- 0..1 Zero or one
- 3..7 Specified range (from three through seven, inclusive)
- 1..3,7 Specified range or exact number

The cardinality adornment denotes the number of links between each instance of the source class and instance of the target class. The arrowhead of inheritance points to the super class and opposite end to subclass. So, inheritance icon is represented by association with an arrowhead. The "has" icon (whole/part relationship i.e. aggregation) is denoted by association with a filled circle at the end denoting the aggregate and the class at the other end denotes the part whose instances are contained by association with on open circle at the end denoting the client.

Class growing plan with attribute named crop together with one modifier operation execute, and one selector operation, conHarvest. There is an association between this class and the environment controller. Wherein instances of the plan define the climate that instances of the controller monitor and modify. This diagram also indicates that the class environment controller is an aggregate whose instances contain exactly one heather, one cooler and any number of lights. The heather and cooler classes in turn are both subclasses of the abstract class actuator, which provides the protocol startup and shout down and which uses the class temperature.

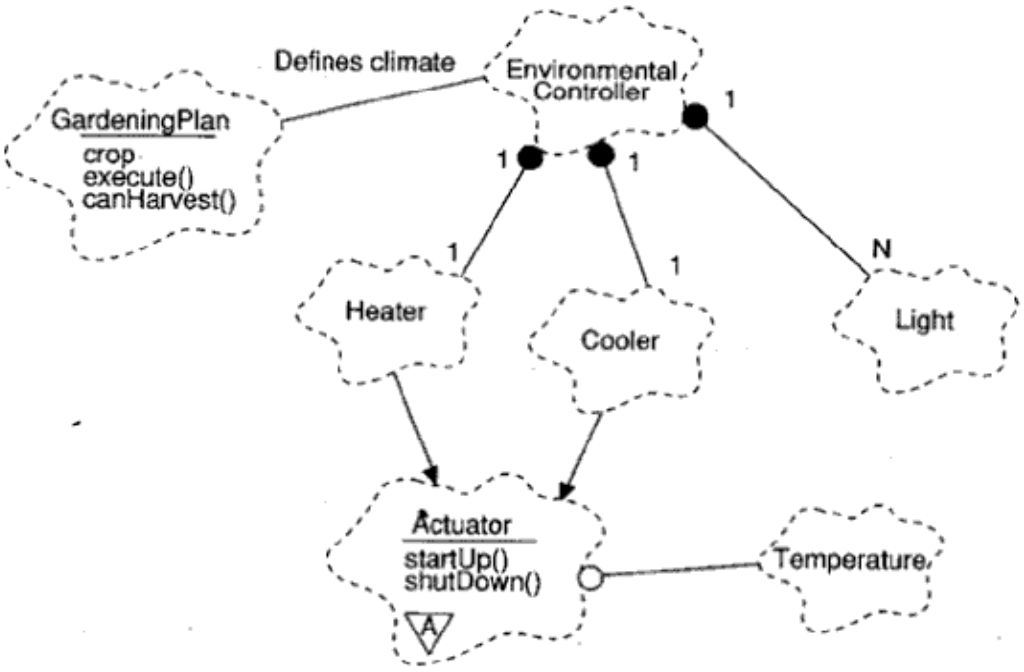


Fig: Hydroponics Gardening System Class Diagram

Class categories: A class category is aggregate containing classes and other class categories, in the same sense that a class is an aggregate containing operations and other classes. Each class in the system must live in a single class category or at the top level of the system. Unlike a class, a class category does not directly contribute state or operations to the model. For contain class diagrams, it is useful to expose some of the classes certain in a particular class category. The list of classes in the class diagram represents an elided new of the class category's specification.

A class represents an encapsulated name space e.g. given the class C contained in class category A, its fully qualified name is A::C as classes and class category may be public meaning that they are usable outside the class category. Other classes may not be usable by any other class outside of the class category. By convention, every class in a class category is considered public, unless explicitly defined otherwise.
Top level class diagrams containing only class categories represent the high level architecture of our system. Such diagrams are externally useful in visualizing the layers and partitions of our system. Layers represent grouping of class categories, just as class categories represent clusters of classes. A common use of layers is to insulate higher layers from lower layer details.

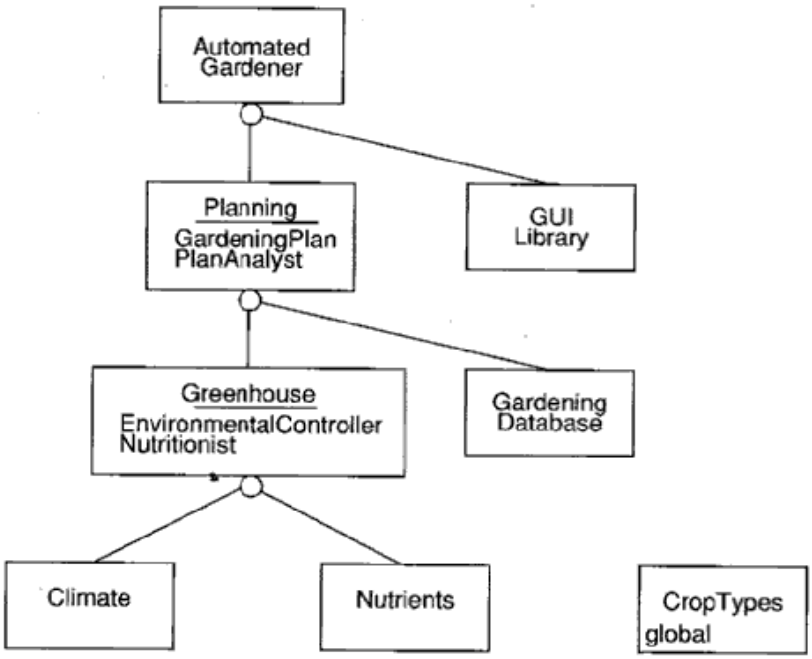


Fig: Hydroponics Gardening System Top-Level Class Diagram

Figure shows an example of a top level class diagram for the hydroponics gardening system. The class category named crop types is global, indicating that its services are available to all other class categories. It should also be noted that the class category planning exposes two of its interesting classes, gardening plan and plan analyst.

Class Utilities

Class utility may denote one or more free subprograms and may name a class that only has class instance variables and operations. Figure shows a common motivation for class utilities.
Class utility is represented as an icon for a plain class and adorned with a shadow. The class utility plan metrics provides two interesting operations expected upon the services of the lower level classes Gardening plan and crop database. As the diagram indicates, plan metrics depends upon crop database for retrieving historical information or certain interesting crops. In turn, the class plan analyst uses the services of plan matrices. Class utility may associate with use of contain static instance of other classes but not inherit from them. Similarly, classes may associate with or use but not inherit from or contain instance of class utility.

NESTING: Classes may be physically nested in other classes and categories as well, to any depth of resting, typically to achieve some control over the namespace. Nesting is indicated by physically nesting icons qualified name of nested class.

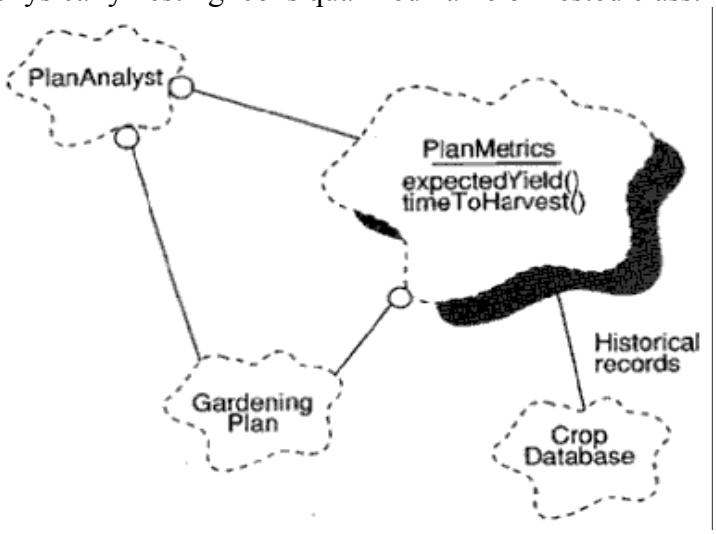


Fig: Nesting

Nutritionist: Nutrient profile.

Export Control: We may specify access by adoring the appropriate relationship with the following symbols:

- <no adornment> public access (the default)
- | protected Access
- || Private access
- ||| Implementation access

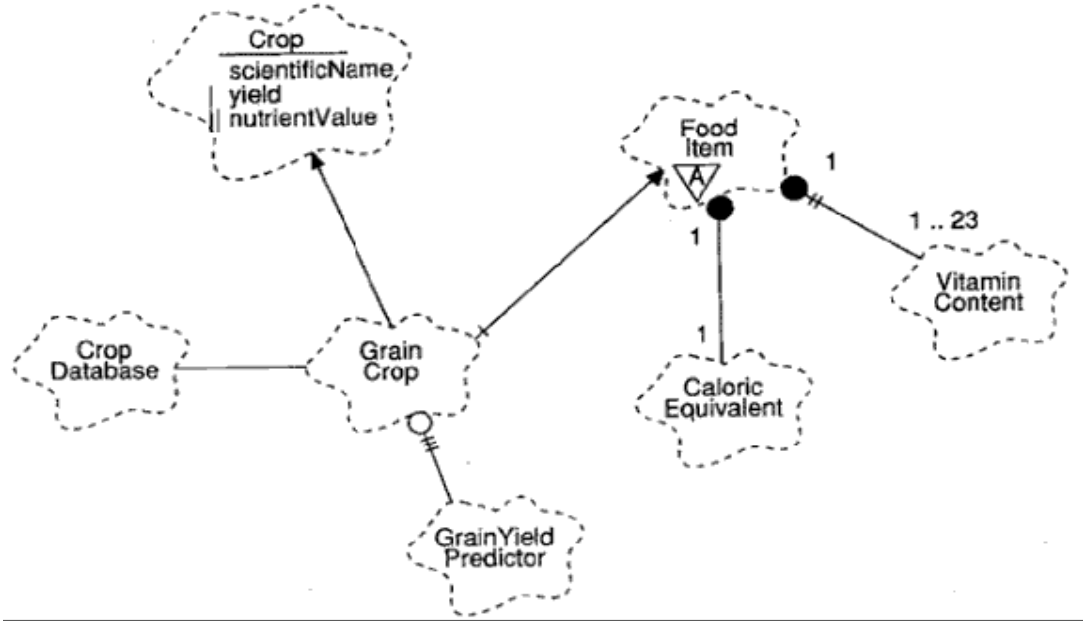


Fig: Export Control

We place these hash marks at the source end of a relationship. In figure class grain crop multiply inherits from the class crop (a public super class) and the abstract class food item (a protected super class) food item in turn contains from 1 to 23 private instances of the class vitamin content, and one public instance of the class caloric equivalent could have been written as an attribute to the class food item, because attributes are equivalent to aggregation whose cardinality is exactly 1:1 and class grain crop uses the class grain viewed predictor as part of its implementation. Class crop has one public attribute (specific name), one protected attribute (yield) and one private attribute (nutrient-value).

Properties: In C++ properties may be as follows:
Static – The designation of a class member object or function
Virtual – The designation of a showed base class in a diamond shaped inheritance lattice.
Friend – The designation of a class that grants rights to another to access its nonpublic parts.

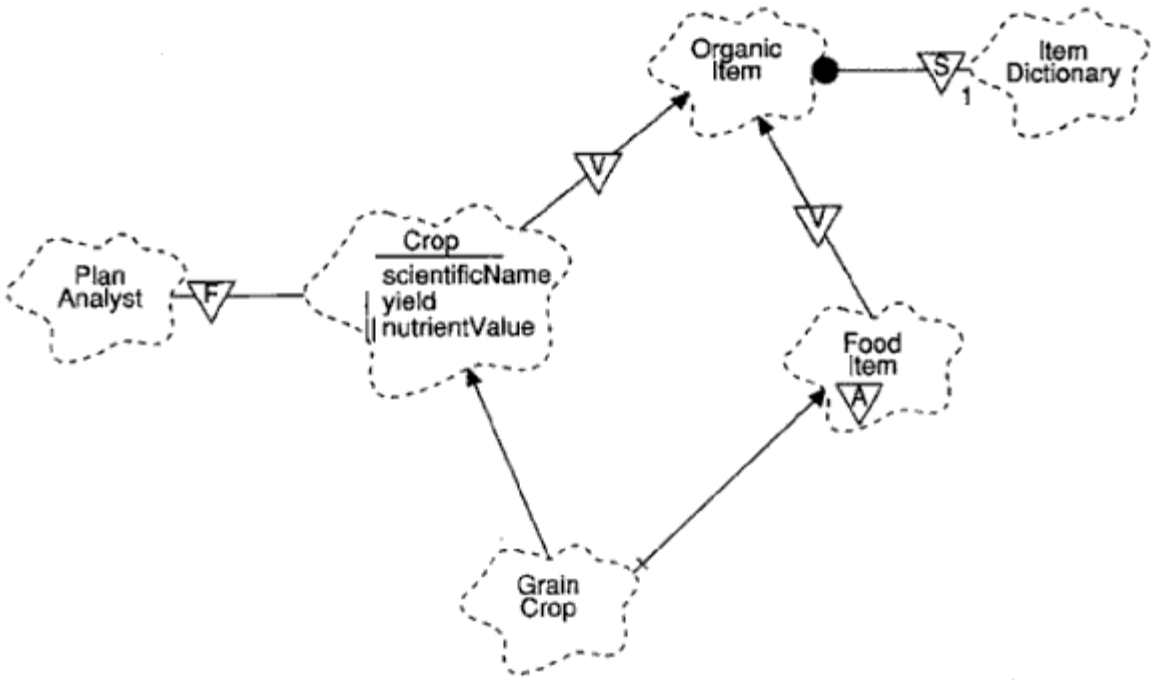


Fig: Properties

Figure provides a different new of the classes. The base class organic item contains one instance of the class item dictionary and thus this instance is owned by the class itself, not by its individual's instances. Class graincrop's inheritance takes on a diamond (for join) shape. in order to have the class Grain Crop share a single copy of the multiply inherited state from organic item, we must specify virtual inheritance as shown in figure.
Friendship may be applied to the supplier of any relationship, denoting that the supplier has granted the right of friendship to the client. For example, we see that the class plan analyst is a friend of the class crop and therefore has access to its nonpublic members, including both the attributes yield and scientific name.

Physical containment:

Two types of physical containment in aggregation are:

- By value: denotes physical containment of a value of the part.
- By reference: Denotes physical containment of a pointer or a reference to the part.

Containment by value as in climate event and nutrient schedule implies that the construction and destruction of these parts occurs as a consequence of the aggregate (Crop history). Containment by reference implies that lifetimes of aggregate (crop history) and part (crop) are independent. Each instance of crop history physically contains only a reference or pointer to are instance of crop.

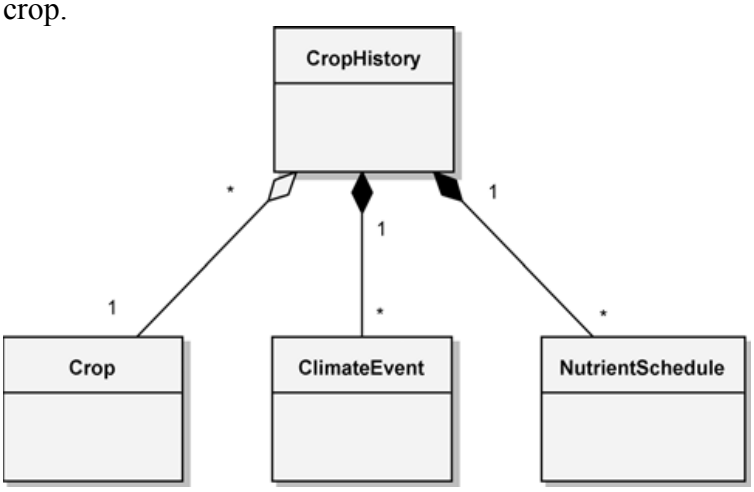


Fig: Physical Containment

Role and Keys:

A role denotes the purpose or capacity where in one class associates with another instances of the class Plan Analyst and Nutritionist are both contributors to the Crop Encyclopedia object (meaning that they both add information to encyclopedia.

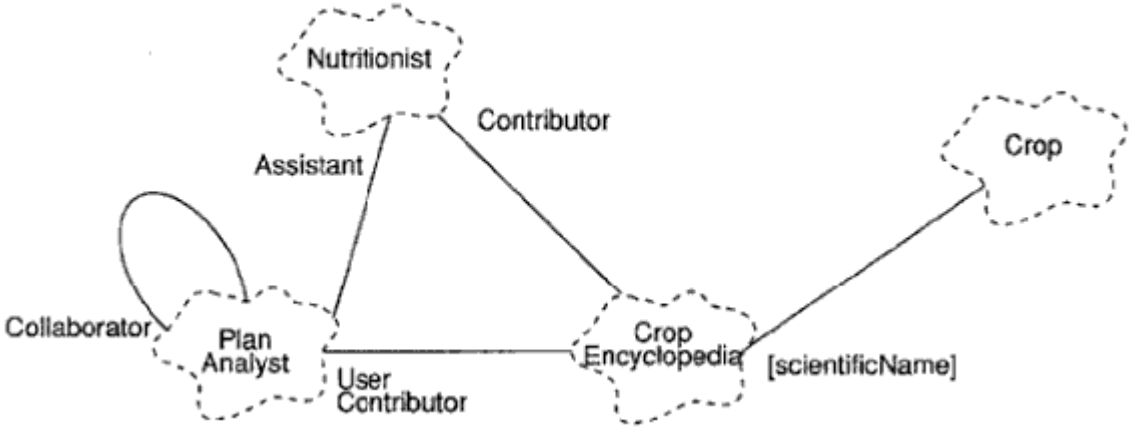


Fig: Roles and Keys

A key is an attribute whose values uniquely identifies a single target object the class crop Encyclopedia uses the attribute scientific name as a key to navigate to individual entries in the set of items managed by instances of crop Encyclopedia.

Constraints:

Constraint is an invariant semantic condition of a class or relationship that must be preserved while the system is in a steady state. Constraints are shown by placing an expression, surrounded by braces, adjacent to the class or relationship. For which the constraint applies. Here is cardinality constraint upon the class environmental controller and light, we require that individual lights be uniquely indexed with respect to one another in the context of this association.

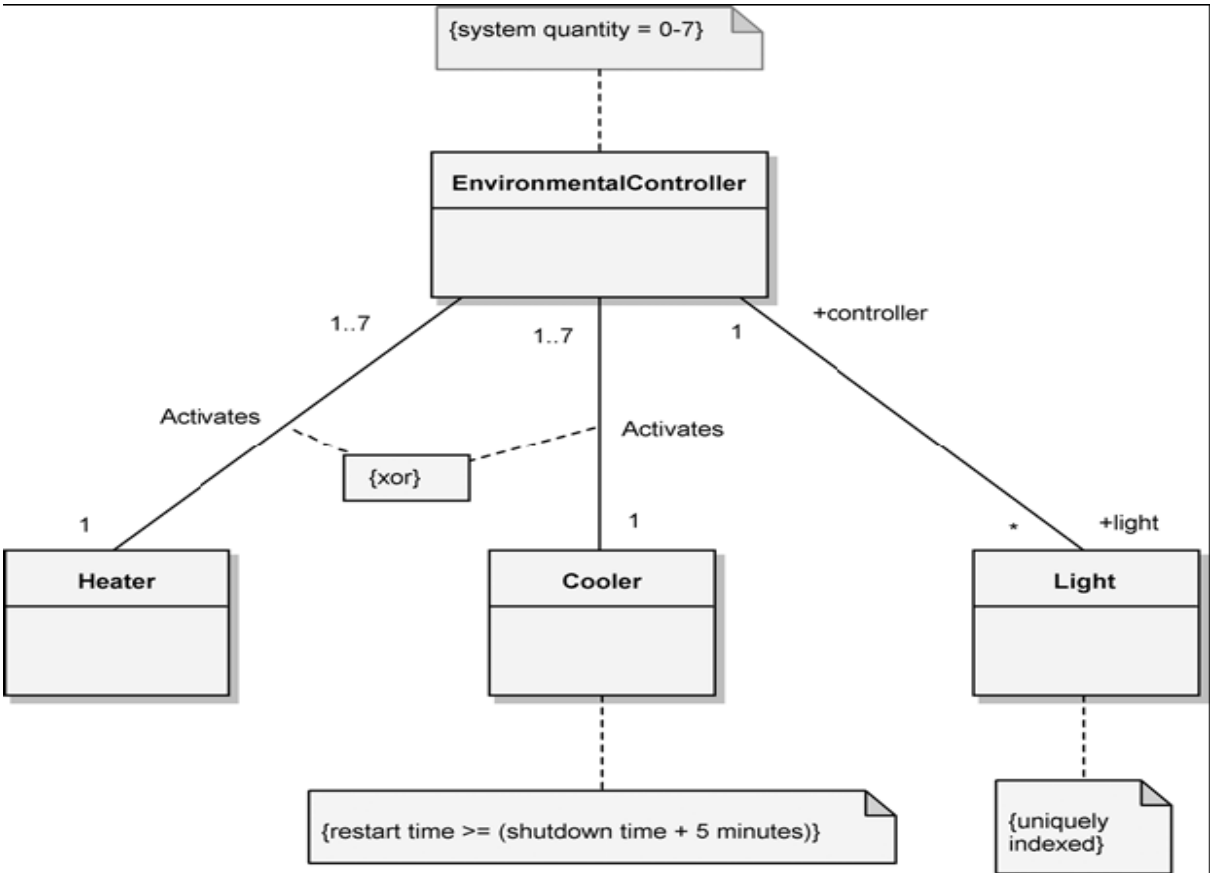


Fig: Constraints

Attributed Association and Notes:

Final advanced concept specific to class diagram concerns itself with the problem of modeling properties of associations. Note attached to a specific operation of the class Nutrient captures our expectation of how this operation will be implemented. Note shaped icon and dashed line used in the diagram.

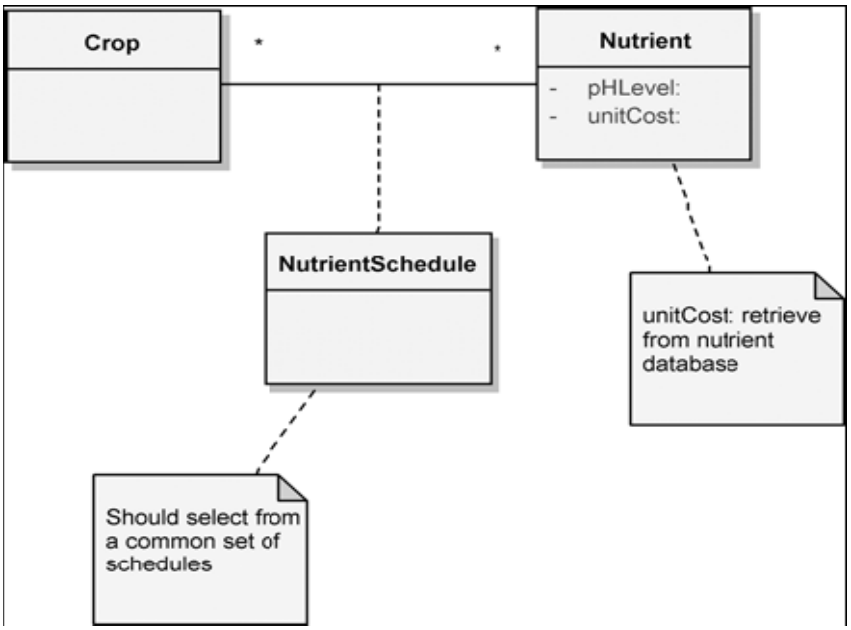


Fig : Association Classes and Notes

Specifications

A specification is a nongraphical form used to provide the complete definition of an entity in the notation, such as a class, an association, an individual cooperation or even an entire diagram. Specification contain some information that is best expressed textually, and so have no graphical representation.

Common Elements: All specifications have at least the following entities.

Name: Identifier.

Definition: Text

Name of entities must be unique. Class names must at least be unique to their enclosing class category whereas operation names have a scope that is local to their enclosing class.

A definition is text that identifies the concept or function represented by the entity.

Class specifications: Each class in the model has exactly one class specification that provides at least the following entities.

Responsibilities text

Attributes: List of attributes

Operations: List of operations

Constraints: list of constraints

Operation Specifications: For each operation that is a member of a class and for all free subprograms, we define one operation specification that provides at least the following entries.

Return class: reference to class

Arguments: List of formal arguments

State transition diagram

A state transition diagram is used to show the state space of a given class, the events that cause a transition from one state to another and the actions that result from a state change. Every class has significant event ordered behavior and so we supply state transition diagram only for the classes that a habit such behavior.

State transition diagram used to indicate dynamic behavior of the system during analysis and to capture the dynamic behavior of individual classes or of collaborations of classes. Two essential elements of a state transition diagram are states and state transitions.

States: At any given point in time, the state of an object encompasses all of its (usually static) properties, together with the current (usually dynamic) values of each of these properties. The state of an object represents the cumulative results of its behavior. E.g. when a telephone is first installed, it is in the idle state, and the phone is ready to initiate or receive calls. When someone picks up the handset, we say that the phone is now off hook and in the dialing state. In this state, we don't expect the phone to ring. We expect to be able to initiate a conversation with a party. When the telephone is now of hook, if the rings and then we pick up the handset, the phone is now in the receiving state and we expect to be able to converse with the party that initiated the conversation.

We can generalize the concept of an individual objects state to apply to the object's class, because all instances of the same class live in the same state space, which encompasses an infinite yet finite number of possible states.

A unique state name is given and for certain states, it is useful to expose the actions associated with a state

State transitions: An event is some occurrence that may cause the state of a system to change. This change of state is called a state transition. Each state transition connects two states. A state may have a transition to itself, and it is common to have many different state transitions from the same state.

e.g. 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 indement weather.
- A cooler fails.

Time passes.

Each of the first four events above 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. An action typically denotes the invocation of a method, the triggering of another event, or the starting or stopping of an action. An activity is some operation that takes some time to complete. An event is simply defined as an operation.

An action may be written using the syntax shown in the following examples.

- Heater start (UPC) – An operation
- Device failure – Triggering of an event
- Start Heating – Begin some activity
- Stop Heating – Terminate some activity

In every state transition diagram, there must be exactly to the state from a special icon, shown as filled circle objects of this class start in the idle state: then they change state upon receipt of the event define climate, for which there is no explicit action. The dynamic behavior of this class

then toggles between the states Dynamic and Nighttime, triggered by the events sunset and sunrise, respectively, whose action is to change the lighting accordingly. In either state, a drop or rise in temp. event invokes an action to adjust the temperature. We return to the idle state whenever we receive a climate event.

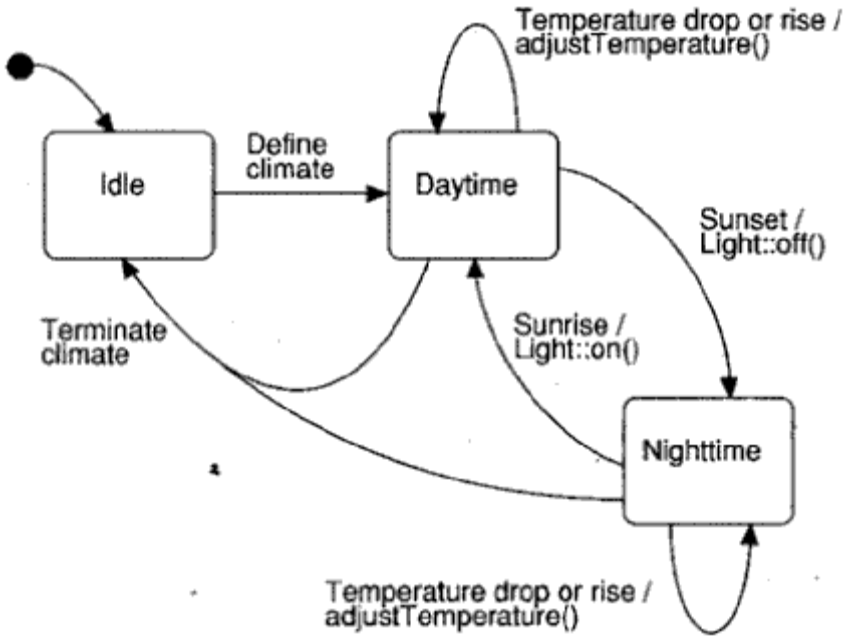


Fig: Environmental Controller State Transition Diagram

Advanced Concepts:

State actions and contributions: Some actions can be specified that is to be carried out upon entry of exit of a state, using the syntax in the following examples.

- Entry start Alarm – start an activity upon entry
- Exit Shout Down ( ) – Invoke an operation upon exit.

Activities may be associated with a state using the syntax in the following example. do cooling – carry out an activity while in the state.

As showing in fig entering the cooling state, we invoke the operation cooler:: startup ( ), and upon exiting this state we invoke the operation cooler:: shout down ( ) in the case of entering and exiting the state failure, we start and stop an alarm respectively consider also the state transition from idle to heating, here we transition if the temperature is too cool, but only if it has been more than five minutes, since we last shout down the heater. This is an example of conditional state transition. We represent a condition as a Boolean expression placed inside brackets. If a condition evaluates false, the state transition may not be triggered until the event occurs again and the condition is re-evaluated.

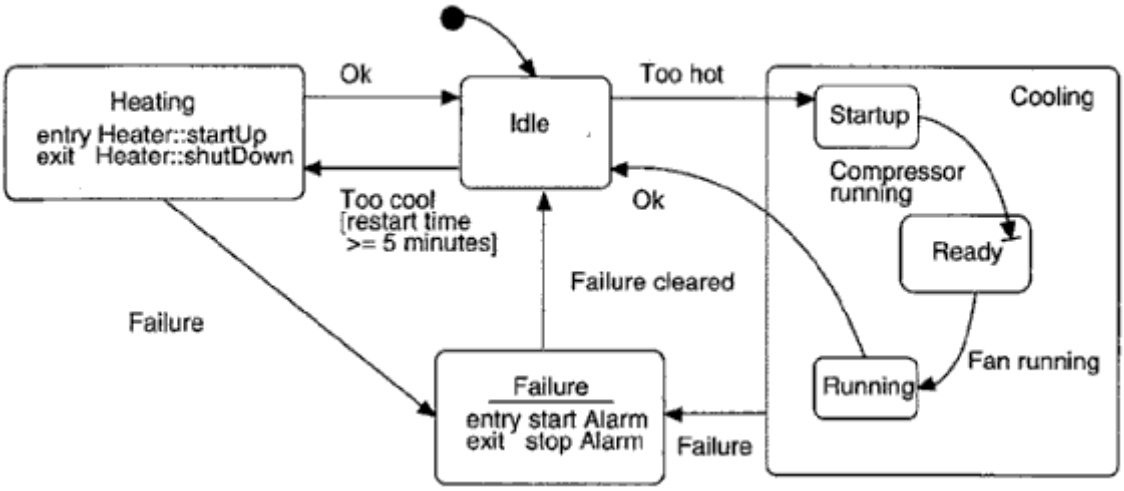


Fig: Actions, conditional transitions and rested states

Nested States: The ability to nest states gives depth to state transition diagram. Enclosing states in figure such as cooling are called super states and its nested states such as running are called sub states. Nesting may be to any depth. If the system is in the cooling state (the super state), then it must also be in exactly one of the three sub states, startup, ready or running.
State transitions are allowed to originate and terminate at any level. Consider then the different forms of state transition:

- Transitioning from one state to a peer state (such as from Failure to Idle or from Ready to Running) is the simplest form of transition; it follows the semantics described in the previous section on state actions and conditional state transitions.
- One may transition directly to a substate (such as from Idle to Startup) or directly from a substate (such as the transition from Running to Idle) or both.
- Specifying a transition directly from a superstate (such as from Cooling to Failure via the Failure event) means that the state transition applies to every substate of the superstate. The transition is passed through all levels, until overridden. These semantics greatly reduce the clutter of common state transitions from substates.
- Specifying a transition directly to a state with substates (such as perhaps to Failure) indeed moves to the new state, but also implies moving to this superstate's default substate.

History: Failure state can be expanded to reveal it sub states. The very first tine, we transition into this state, we also move to its default start state create log, indicated by unlabeled transition from the filled circle. After that the log is created and we move into the log ready state. After positing the failure, we return to this state. The next time we transition into the failure state, we don't want to create the log again.

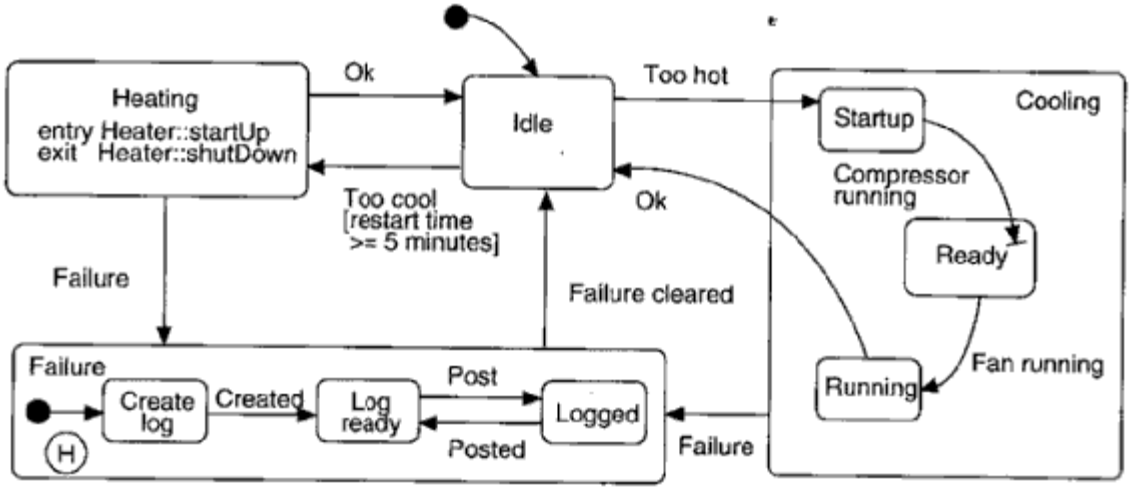


Fig: History

OBJECT DIAGRAM

An object diagram is used to show the existence of objects and their relationships in the logical design of a system. a single object diagram represents a new of the object structure of a system. Object diagram is used to indicate the semantics of primary and secondary scenarios that provide a trace of the system's behavior during analysis and also used to illustrate the semantics of mechanisms in the logical design of a system.

Essential elements of an object diagrams are objects and their relationships.

Objects: Object icon is used to represent object in the object diagram.

The name of an object follows the syntax for attributes in any of the following forms.

- A - Object name only
- A : C - Object class only
- A : C - Object name and class

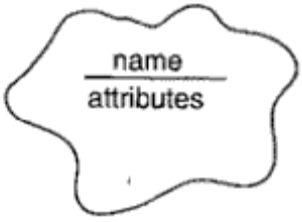


Fig: Object Icon

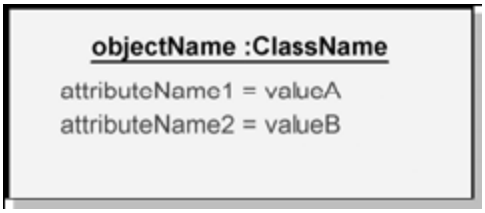


Fig: Generic Object Icon

If we never specify the class of an object, either explicitly using the above syntax, or implicitly through the object's specification, then the object's class is considered anonymous. If we only specify a class name, the object is said to be anonymous. Each such icon without an object name denotes a distinct anonymous object. An any case, the name given for an object's class must be that of the actual class. Attribute names must also include icons that denote class utilities and metaclasses.

Object Relationship: Object interact with other objects through their links which are represented by the icon as shown in figure. A link is an instance of an association.

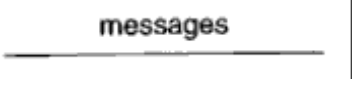


Fig: Object Relationship Icon

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. link) between instances of the classes, whereby one object may send message to another. It is possible for an object to send a message to itself. The object which invokes the operation is known as client and the object which provides the operation is known as server.

Object relation icon can consists of a collection of message and each message consists of the following three elements.

- D - A synchronization symbol denoting the direction of the invocation
- M - An operation invocation or event dispatch
- S - Optionally, a sequence number

We indicate the direction of message by adoring it with a directed line pointing to the supplier object. Operation syntax is:

- N () – operation name only
- R N (arguments) – operation return object, name and actual arguments to show an explicit ordering of events, we may optionally prefix a sequence number (starting at one) to an operation invocation or relative ordering of messages.

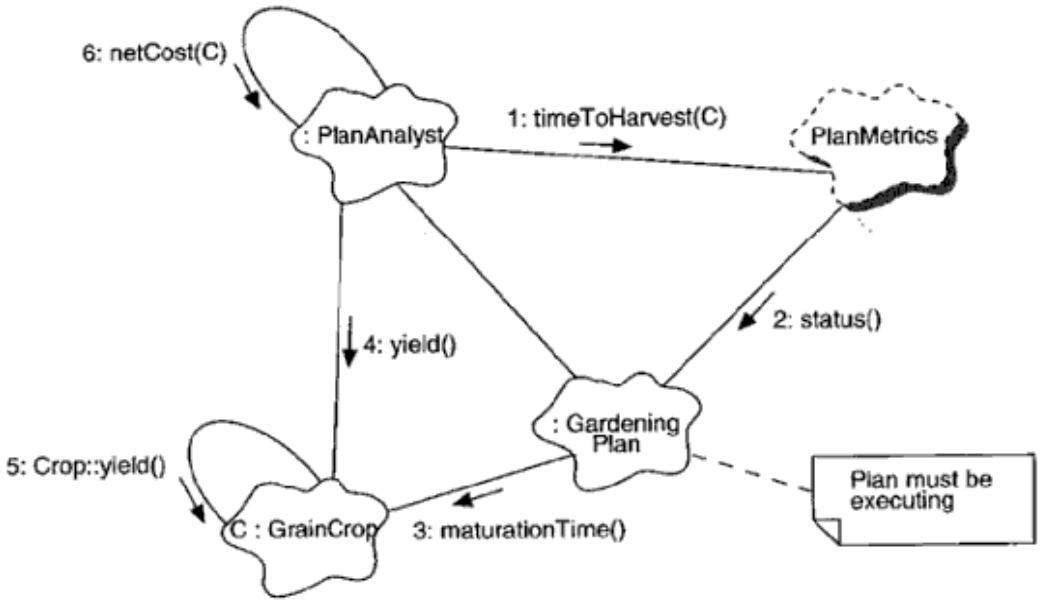


Fig: Hydroponics Gardening System Object Diagram

We see from this diagram that the action of the scenario begins with some plan analyst object invoking the operation time to Harvest () upon the class utility plan metrics. The object C is passed as an actual argument to this operation. The plan metrics class status () upon a certain unnamed gardening plan object. the diagram includes a development note indicating that we must check that the given plan is in operation maturation time () upon the selected grain crop object, asking for the time the completes, control then returns to the plan analyst object, which then calls c. yield () directly, which in turn propagates this operation to the crop's super class (the operation crop:: yield()). Control again returns to the plan Analyst object, which completes the scenario by invoking the operation netcost() upon itself.

Advanced Concepts: Roles, keys and constraints: for certain object diagrams, it is useful to restate role (as in class diagram) on the link between two objects. Some Plan Analyst object inserts a specific crop into an anonymous crop encyclopedia object and does so while acting in the role of contributor.

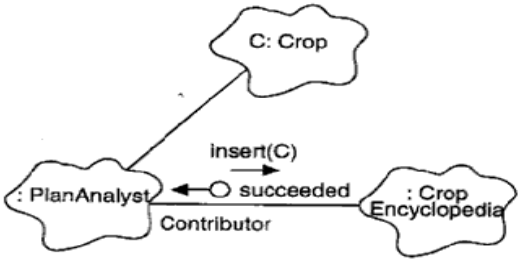


Fig: Roles

Data flow: Data may flow with or against the direction of a message. In figure it is shown that the value succeeded returns upon completion of the message inserted. We may use either an object or value in a dataflow.

Visibility: We may adorn the links in object diagrams with icons that represent the visibility of one object to another.

Adornment in figure denotes that the class utility is global to the declaration of the analyst object. The object C is visible to the Plan Analyst object and the Gardening Plan object through to different paths. From the perspective of the Plan Analyst object, the grain crop object c is visible as a parameter to some analyst operation (the padornment); from the perspective of the gardening plan object, the grain crop object c is visible as a field (i.e. as a part of the plan aggregate object).

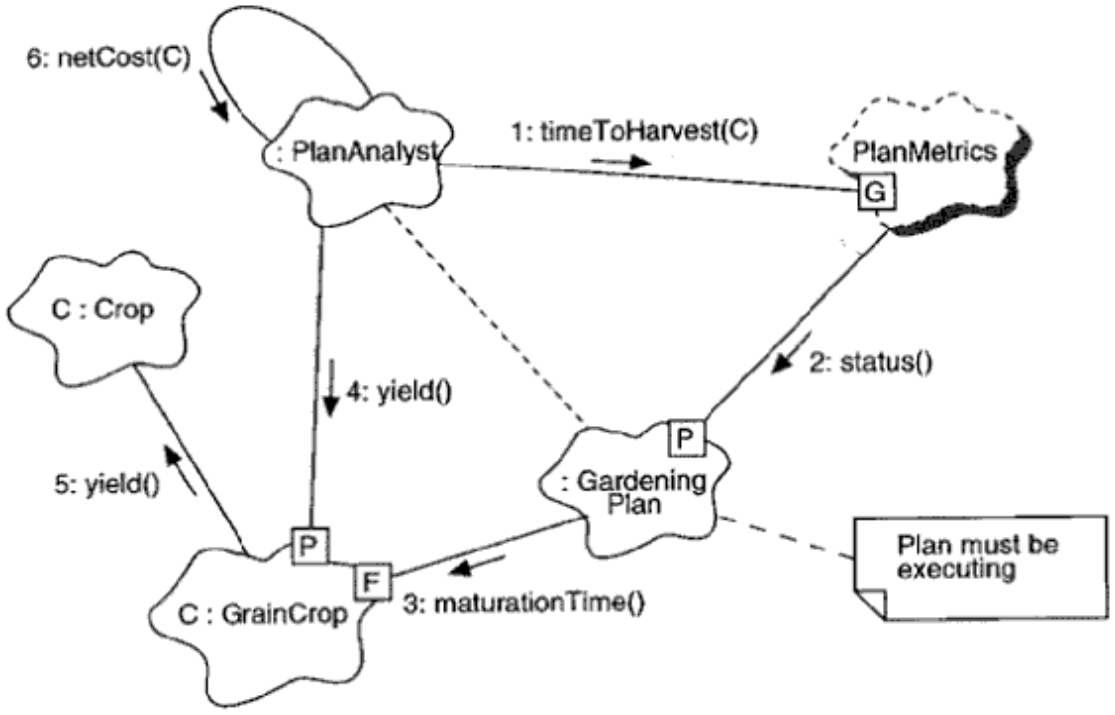


Fig: Visibility

The following adornments may be used to indicate visibility.

- G - The supplier object is global to the client
- P - The supplier object is a parameter to some operation of the client
- F - The supplier object is a part of the client object
- L - The supplier object is a locally declared object in the scope of the object diagram.

Active objects and synchronization: Active objects embody their own thread of control. Concurrency of an object in an object diagram is represented by adoring its object icon with the names sequential, guarded, synchronous or active, placed in the lower left of the icon. In figure, H. C and environmental controller are all active objects and thus embody their thread of control.

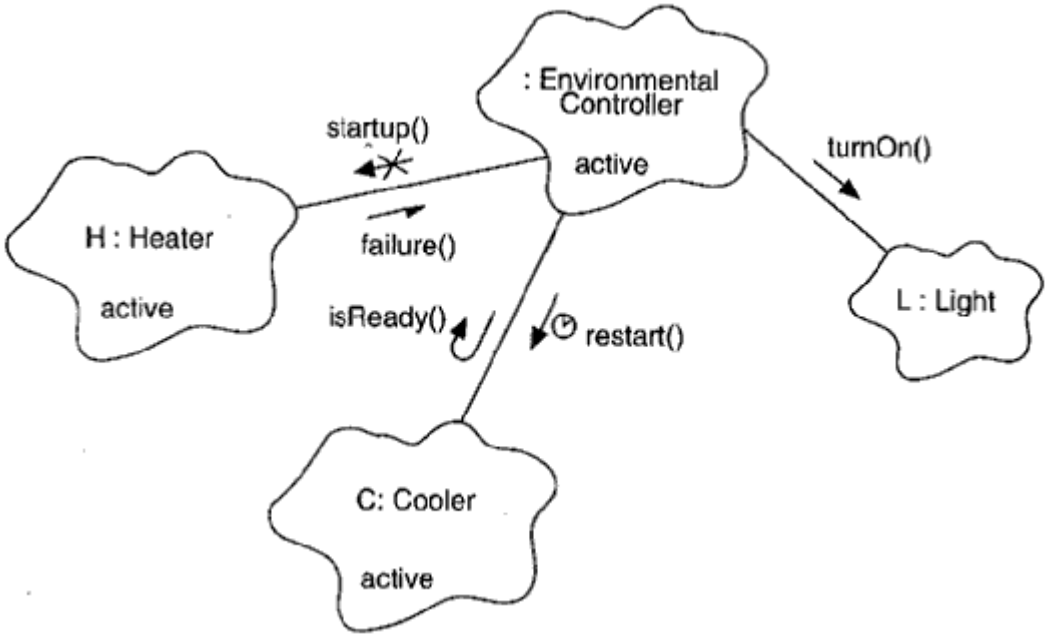


Fig: Active Objects & Synchronization

Startup () is synchronous, meaning that the client will wait forever until the supplier accepts the message. In the case of message failure (), it is asynchronous meaning that the client sends the event to the supplier for processing the supplier for processing the supplier queues the message, and the client then proceeds without waiting for the supplier

Time Budgets: In certain time critical applications, we use sequence numbers that denote time (in seconds), prefixed by the plus symbol. E.g. as in figure message startup () is first invoked 5 seconds after the start of the scenario, followed by the message ready () 6.5 seconds after the start of the scenario and there followed by the message turn on () after 7 seconds.

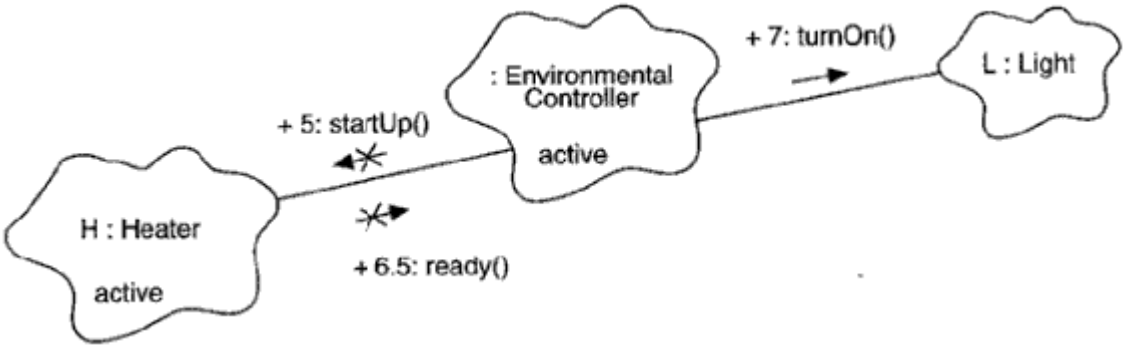


Fig: Time Budgets

Interaction diagram

Interaction diagram is another way of representing object diagram. It permits the inclusion of other information such as links, attribute values, roles, data flow, and visibility as shown in figure. The advantage of using interaction diagram is that it is easier to read the passing of messages in relative order.

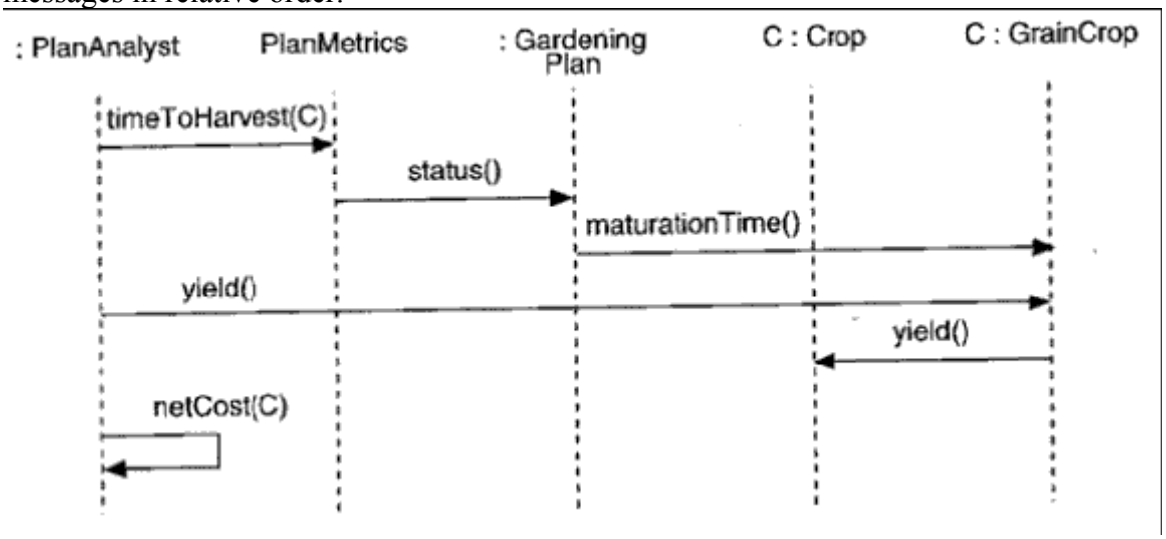


Fig: Hydroponics Gardening Interaction Diagram

Interaction diagram appears in tabular form the entities of interest are written horizontally across the top of the diagram. A dashed vertical line is drawn below object. Messages are shown horizontally using the same syntax and synchronization symbols as for object diagrams. The endpoints of the message connect with the diagram and are drawn from the client to the supplier. 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 and hence it is unnecessary to use sequence numbers.

Advanced Concept: **Scripts and focus of control** can be added in interaction diagram to make them more expressive as shown in figure.

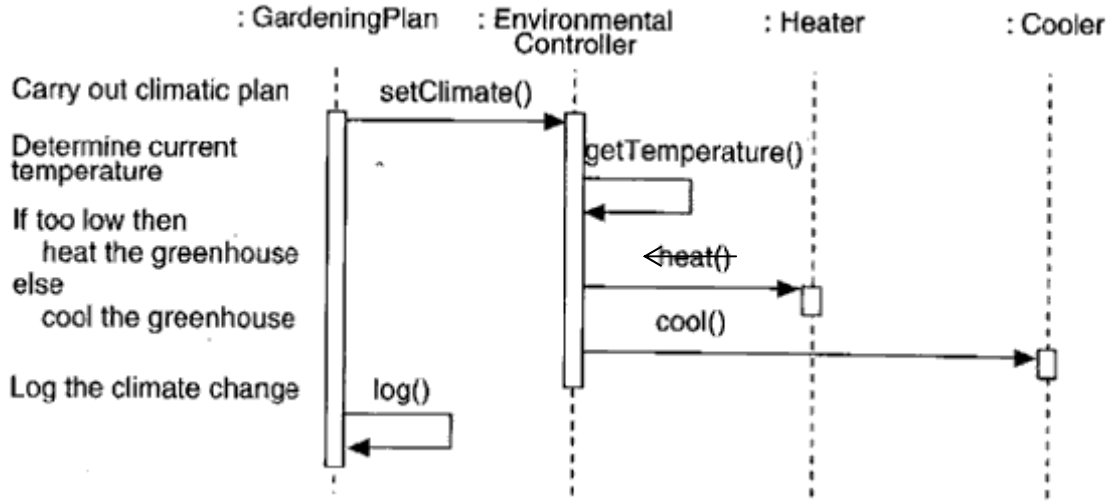


Fig: Scripts & Focus of Control

A script may be written to the left of an interaction diagram, with the steps of the script aligning with the message invocations and may be written using free form or structured English text. Focus of control can be shown by adorning the vertical lines descending from each object in an interaction diagram with a box representing the relative time that the flow of control is focused in that object.

MODULE DIAGRAM

It is used to show the allocation classes and objects to modules in the physical design of a system. a single module diagram represents a new of the module structure of a system. Two essential elements of module diagrams are modules and their dependencies.

Modules: The first three icons denote files distinguished by their functions. The main program icon denotes a file that contains the root of a program e.g. some .CPP file in C++ containing definition of privileged nonmember function called main. The specification module denotes on files and body modules denote .cpp files. A name is required for each module. Each module encompasses the declaration or definition of classes, objects, and other language details.

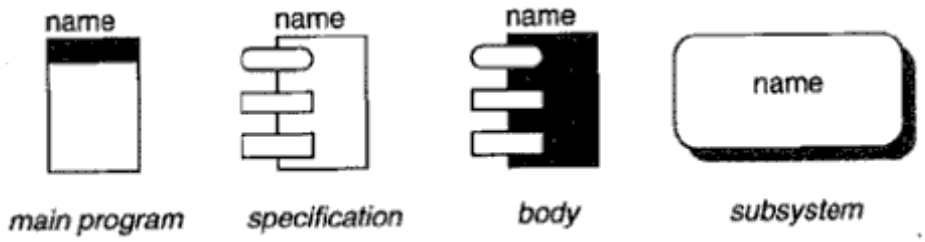


Fig: Module and Subsystem Icons

Dependencies: The relationship between two modules is a compilation dependency, represented by a directed line pointing to the module upon which the dependency exists. E.g. in C++, compilation dependency is indicated by # include directive in ada by with clauses. Here are six modules. Two of them climatedefs and cropdefs are only specifications and serves to provide common types and constants. The remaining four modules are shown with their specification and bodies grouped together as specification and body of a module are intimately related. Dependency of body upon corresponding specification is hidden because the two parts are overlaid. Name of body is kept hidden. Dependencies in diagram suggest, a partial ordering of compilation e.g. body of climate depends upon the specification of heater, which in turn depends upon the specification of climatedefs.

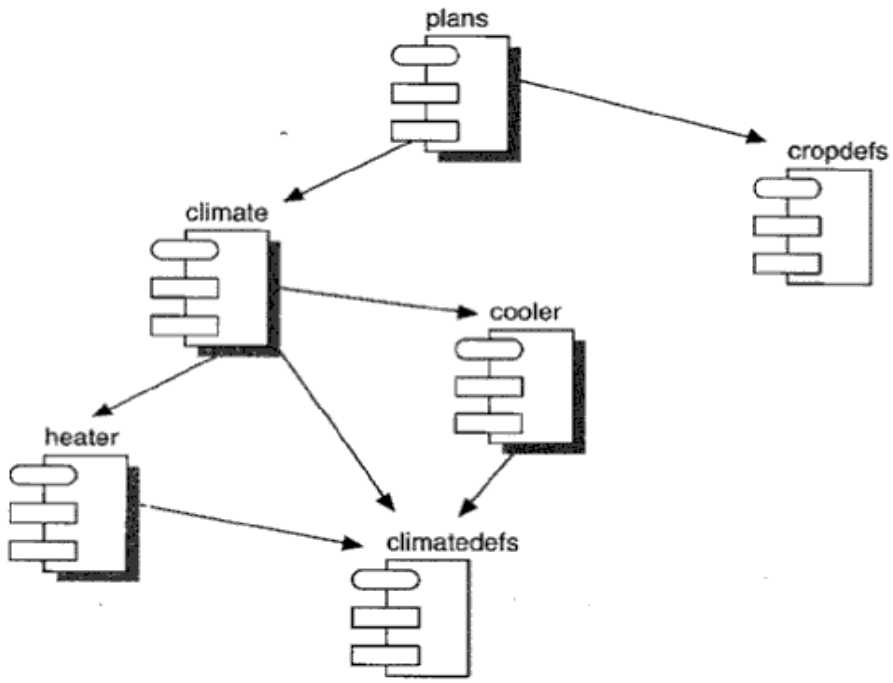


Fig: Hydroponics Gardening System Module Diagram

Subsystems: Subsystems represent clusters of logically related modules. Subsystems serve to partition the physical model of a system. A subsystem is an aggregate containing other modules and other subsystems. Each module in the system must live in a single subsystem or at the top level of the system. Some of the modules enclosed by a subsystem may be public, meaning that they are exported from the subsystem and hence usable outside the subsystem. Other modules may be part of the subsystem's implementation meaning that they are not intended to be used by any other module outside of the subsystem. By correction, every module in a subsystem is considered public.

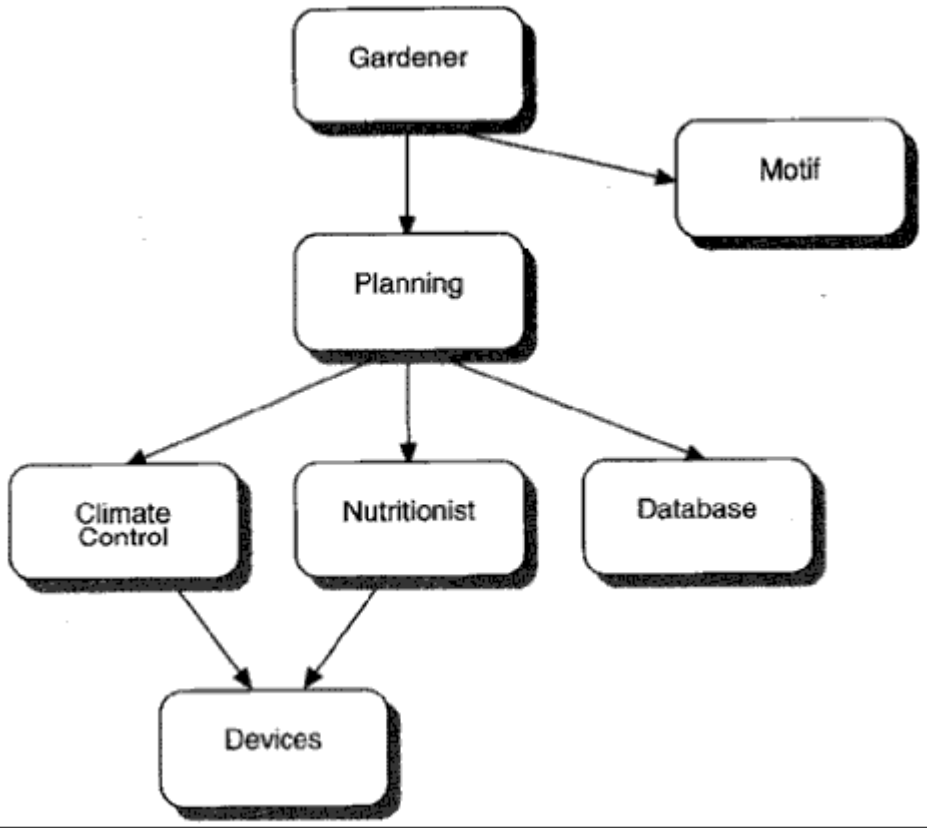


Fig: Hydroponics Gardening System top level module diagram

A system can have dependencies upon other subsystems or modules, and a module can have dependencies upon a subsystem. In figure, low level device classes from the climate and nutrients class categories and place their corresponding modules into one subsystem called devices. The greenhouse class category is also splitted into the two subsystems called climate control and nutritionist.

Advanced concepts:

Language Tailoring: certain languages, such as ada define other kinds of modules than the simple one provided in figure. In particular, ada defines generic packages, generic subprograms and tasks as separate

Segmentation: Module diagrams can be extended to help visualize this segmentation by including language adornments to each module diagram that denote its corresponding code or data segment.

Specifications: Each entity in a module diagram may have a specification, which provides its complete definition.

PROCESS DIAGRAM

Process diagram is used to show the allocation of processes to processes in the physical design of a system. It represents a new into the process structure of a system. Essential elements of process diagram are processors, devices and their connections.

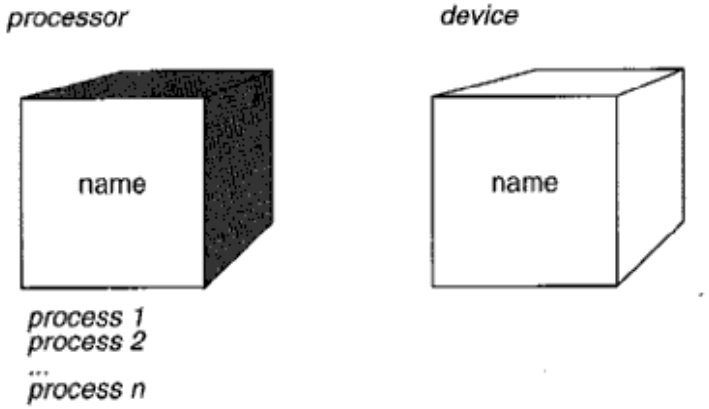


Fig: Processor and Device Icons

Processors: A processor is a piece of hardware capable of executing programs. A name is required for each processor. A list of processes below the processor icon denotes the root of a main program (from a module diagram) or the name of an active object (from an object diagram).

Devices: A device is a piece of hardware incapable of executing programs. A name is required for each devices.

Connections: Processor and devices must communicative with one another. An undirected line is used to indicate the connection between a device and a processor, a processor and a processor or a device and a device. A connection usually represents some direct hardware coupling. Such as Ethernet connection and some indirect coupling's such as satellite to ground communications. An arrow may be added to show the direction in the connection.
As shown in figure, system is decomposed into a network, of four computers, one assigned to a gardener workstation and the other allocated to individual greenhouses. Processes running on the greenhouse computers can not communicate directly with one another, although they can communicate with processes running on the gardener workstation.

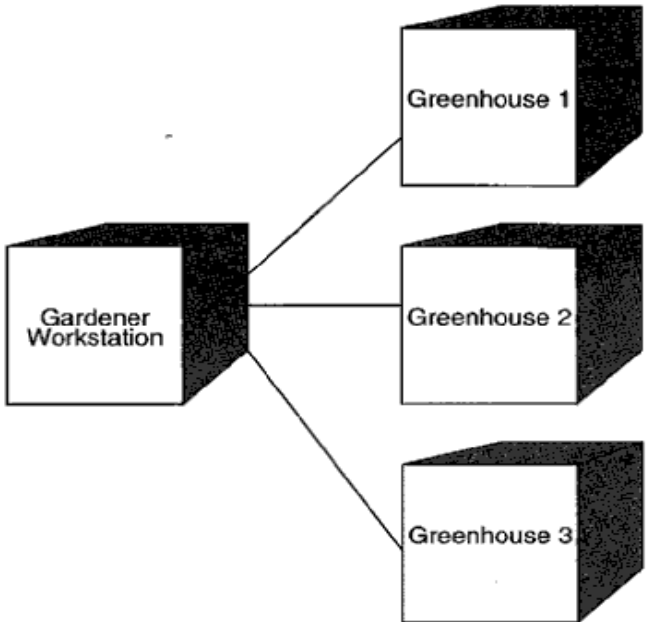


Fig: Hydroponics Gardening System

Advanced concepts:

Tailoring: Instead of using standard icons as shown above, we can use and define specific icons to graphically represent an embedded microcomputer (a processor), a disk, a terminal, and other devices and use in process diagrams.

Nesting: To represent complex hardware configuration of a system, it is necessary to represent groups of processor, devices and connections and all such groups are indicated by named icon shaped as a rounded rectangle with dashed lines. Each such icon denotes a district group to be nested entities.

Process Scheduling: Policy for how to schedule the execution of processes within a processor. Five general approaches of scheduling are:

i) Preemptive – Higher priority processes that are ready to execute may preempt lower priority ones that are currently executing.
ii) Non-preemptive – The current process continues to execute until it relinquishes control.
iii) Cycle – Control passes from one process to another and each process is given a fixed amount of processing time.
iv) Executive – Some algorithm controls process scheduling.
v) Manual – Processes are scheduled by a user outside of the system.

Specification: Textual specification of each processor, device and connection provide their complete definition.

Chapter 5: The Process

1. First Principles

1.1 Traits (Qualities) of Successful Projects

A successful software project is one whose deliverables -satisfy and possibly exceed the customer's expectations, was developed in a timely and economical fashion, and is resilient to change and adaptation. There are two traits that are common to all of the successful object-oriented systems we have encountered:

- The existence of a Strong architectural vision
- The application of a well-managed iterative and incremental development life cycle

1.1.1 Architectural Vision

A system that has a sound architecture is one that has *conceptual integrity* and, conceptual integrity is the most important consideration in system design. The architecture of an object-oriented software system encompasses its class and object structure, organized in terms of distinct layers and partitions. In some ways, the architecture of a system is largely irrelevant to its end users.

Clean internal structure is essential to constructing a system that is understandable, can be extended and reorganized, and is maintainable and testable. Fundamentally, good architectures tend to be object-oriented. This is not to say that all object oriented architectures are good, or that only object-oriented architectures are good. However, as it can be shown that the application of the principles that underlie object-oriented decomposition tend to yield architectures that exhibit the desirable properties of organized complexity.

Good software architectures tend to have several attributes in common:

- They are constructed in well-defined layers of abstraction, each layer representing a coherent abstraction, provided through a well-defined and controlled interface, and built upon equally well-defined and controlled facilities at lower levels of abstraction.
- There is a clear separation of concerns between the interface and implementation of each layer, making it possible to change the implementation of a layer without violating the assumptions made by its clients.
- The architecture is simple: common behavior is achieved through common abstractions and common mechanisms.

1.1.2 Iterative and Incremental Life Cycle

Consider two extremes:

- An organization that has no well-defined development life cycle, and
- One that has very rigid and strictly-enforced policies that dictate every aspect of development.

In the former case, we have anarchy: through the hard work and individual contributions of a few developers, the team may eventually produce something of value, but we can never reliably predict anything: not progress to date, not work remaining, and certainly not quality.

The team is likely to be very inefficient and, in the extreme, may never reach closure and so never deliver a software product that satisfies its customer's current or future expectations. This is an example of a project in free fall.
In the second case, we have a dictatorship, in which creativity is punished, experimentation that could yie1d a more elegant architecture is discouraged, and the customer's real expectations are never correctly communicated to the lowly developer who is hidden behind a veritable paper wall erected by the organization's bureaucracy.
The successful object-oriented projects we have encountered follow neither anarchic nor draconian development life cycles. Rather, we find that the process that leads to the successful construction of object-oriented architectures tends to be both iterative and incremental.

Such process is iterative in the sense that it involves the successive refinement of an object-oriented architecture, from which we apply the experience and results of each release to the next iteration of analysis and design.
The process is incremental in the sense that each pass through an analysis/design/evolution cycle leads us to gradually refine our strategic and tactical decisions, ultimately converging upon a solution that meets the end user's real (and usually unstated) requirements, and yet is simple, reliable, and adaptable.
An iterative and incremental development life cycle is the antithesis of the traditional waterfall life cycle, and so represents neither a strictly top-down nor a bottom-up process. It is reassuring to note that there are precedents in the hardware and software communities for this style of development.

1.2 Towards a rational (balanced) Design Process

Clearly, however, we desire to be prescriptive; otherwise, we will never secure development process for any organization. It is for this reason that we spoke earlier of having a well managed incremental and iterative life cycle: well-managed in the sense that the process can be controlled and measured, yet not so rigid that it fails to provide sufficient degrees of freedom to encourage creativity and innovation.
Having a prescriptive process is fundamental to the maturity of a software organization. As there are five distinct levels of process maturity defined by CMM(Capability Maturity Model):

1. ***Initial*** : The development process is *ad hoc* and often chaotic. Organizations can progress by introducing basic project controls.
2. ***Repeatable :***The organization has reasonable control over its plans and commitments. Organizations can progress by institutionalizing a well-defined process.
3. ***Defined*** :The development process is reasonably well-defined, understood, and practiced; it serves as a stable foundation for calibrating the team and predicting progress. Organizations can progress their development practices.
4. ***Managed :*** The organization has quantitative measures of its process. Organizations can progress by lowering the cost of gathering this data, and instituting practices that permit this data to influence the process.
5. ***Optimizing*** :The organization has in place a well-tuned process that consistently yields products of high quality in a predictable, timely, and cost-effective manner.

Unfortunately, we will never find a process that allows us to design software in a perfectly rational way, because of the need for creativity and innovation during the development process. However, we will come closer to a rational process if we try to follow the process rather than proceed on an *ad hoc* basis. When an organization undertakes many software projects, there are advantages to having a standard procedure.

As we move our development organizations to higher levels of maturity, how then do we bring together the need for creativity and innovation with the requirement for more controlled management practices?

The answer appears to lie in distinguishing the micro and macro elements of the development process.

The micro process is more closely related to Boehm's spiral model of development, and serves as the framework for an iterative and incremental approach to development.

The macro process is more closely related to the traditional waterfall life cycle, and serves as the controlling framework for the micro process. By reconciling these two disparate processes, we end up a fully rational development process, and so have a foundation for the defined level of software process maturity.

We must emphasize that every project is unique, and hence developers must strike a balance between the informality of the micro process and the formality of the macro process.

1.3 The Micro Development Process

To a large extent, the micro process represents the daily activities of the individual developer or a small team of developers.

The micro process of object-oriented development is largely driven by the stream of scenarios and architectural products that emerge from and that are successively refined by the macro process.

The micro process applies equally to the software engineer and the software architect.

- From the perspective of the engineer, the micro process offers guidance in making the many tactical decisions that are part of the daily production and adaptation of the architecture;
- From the perspective of the architect, the micro process offers a framework for evolving the architecture and exploring alternative designs

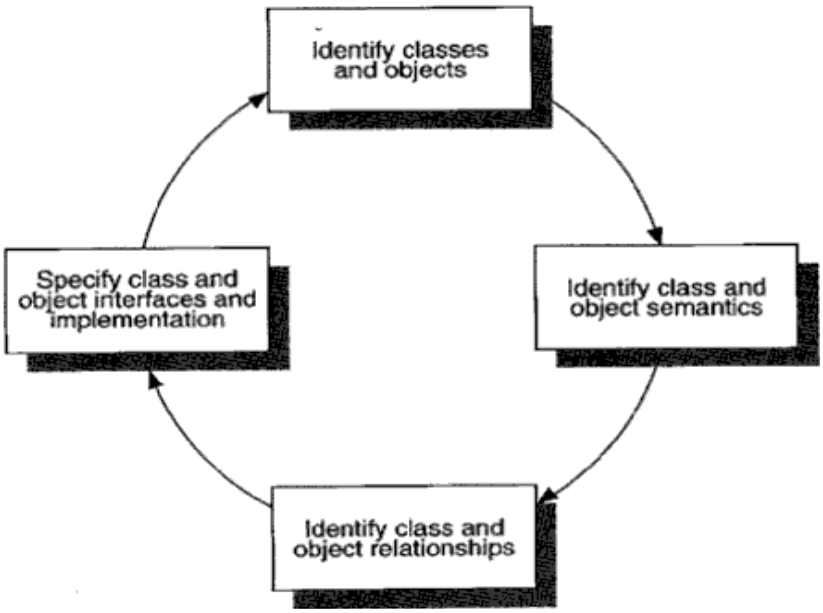


Fig: Micro Development Process

The different phases of a software project, such as design, programming, and testing, cannot be strictly separated. The micro process tends to track the following activities:

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Specify the interface and then the implementation of these classes and objects.

1.3.1 Identifying Classes and Objects

1.3.1.1 Purpose

The purpose of identifying classes and objects is to establish the boundaries of the problem at hand, additionally; this activity is the first step in devising an object-oriented decomposition of the system under development.

As part of analysis, we apply this step to discover those abstractions that form the vocabulary of the problem domain, and by so doing, we begin to constrain our problem by deciding what is and what is not of interest.

As part of design, we apply this step to invent new abstractions that form elements of the solution.

As implementation proceeds, we apply this step in order to invent lower-level abstractions that we can use to construct higher-level ones, and to discover commonality among existing abstractions, which we can then exploit in order to simplify the system's architecture.

1.3.1.2 Products

The central product of this step is a data dictionary that is updated as development proceeds..

1.3.1.3 Activities

The identification of classes and objects involves two activities: *discovery and invention*.

In each case, we carry out these activities by applying any of the various approaches to classification.
A typical order of events might be the following:

- Apply the classical approach to object-oriented analysis to generate a set of candidate classes and objects. .
- Apply the techniques of behavior analysis to identify abstractions that: are directly related to system function points.
- From the relevant scenarios generated as part of the macro process, apply the techniques of use-case analysis.

1.3.1.4 Milestones and Measures

We successfully complete this phase when we have a reasonably stable data dictionary. Because of the iterative and incremental nature of the micro, process, we don't expect to complete or freeze this dictionary until very late in the development process. Rather, it is sufficient that we have a dictionary containing an ample set of abstractions, consistently named and with a sensible separation of responsibilities.

1.3.2 Identifying the Semantics of Classes and Objects

1.3.2.1 Purpose

The purpose of identifying the semantics of classes and objects is to establish the behavior and attributes of each abstraction identified in the previous phase. Here we refine our candidate abstractions through an intelligent and measurable distribution of responsibilities.

- As part of analysis, we apply this step to allocate the responsibilities for different system behaviors.
- As part of design, we apply this step to achieve a clear separation of concerns among the parts of our solution.
- As implementation proceeds, we move from free-form descriptions of roles and responsibilities to specifying a concrete protocol for each abstraction, eventually culminating in a precise signature for each operation.

1.3.2.2 Products

There are several products that flow from this step. The first is a refinement of the data dictionary, whereby we initially attach responsibilities to each abstraction. As development proceeds, we may create specifications for each which state the named operations that form the protocol of each class.

1.3.2.3 Activities

There are three activities associated with this step: storyboarding, isolated class design, and pattern scavenging. The primary and peripheral scenarios generated by the macro process are the main drivers of storyboarding. This activity represents a top-down identification of semantics and, where it concerns system function points, addresses strategic issues.
A typical order of events might be the following:

- Select one scenario or a set of scenarios related to a single function point; from the previous step, identify those abstractions relevant to the scenario.
- Walk though the activity of the scenario, assigning responsibilities to each abstraction sufficient to accomplish the desired behavior. As needed, assign attributes that represent structural elements required to carry out certain responsibilities.

- As the storyboarding proceeds, reallocate responsibilities so that there is a reasonably balanced distribution of behavior. Where possible, reuse or adapt existing responsibilities.
- Splitting large responsibilities into smaller ones is a very common action; less often, but still rarely, trivial responsibilities are assembled into larger behaviors.

1.3.2.4 Milestones and Measures

We successfully complete this phase when we have a reasonably sufficient, primitive, and complete set of responsibilities and/or operations for each abstraction.

1.3.3 Identifying the Relationships among Classes and Objects

1.3.3.1 Purpose

The purpose of identifying the relationships among classes and objects is to set the boundaries of and to recognize the collaborators with each abstraction identified earlier in the micro process. This activity formalizes the conceptual as well as physical separations of concern among abstractions begun in the previous step.

As part of analysis, we apply this step to specify the associations among classes and objects (including certain important inheritance and aggregation relationships)..

As part of design, we apply this step to specify the collaborations that: form the mechanisms of our architecture, as well as the higher-level clustering of classes into categories and modules into subsystems.

As implementation proceeds, we refine relationships such as associations into more implementation-oriented relationships, including instantiation and use.

1.3.3.2 Products

Class diagrams, object diagrams, and module diagrams are the primary products of this step.

During analysis, we produce class diagrams that state the associations among abstractions, and add detai1s from the previous step (the operations and attributes for certain abstractions) as needed to capture the important subtleties of our decisions.

During design, we refine these diagrams to show the tactical decisions we have made about inheritance, aggregation, instantiation, and use. We must focus on the "interesting" ones, where our measure of interesting encompasses any set of related abstractions whose relationships are an expression of some fundamental architectural decision, or that express a detail necessary to complete a blueprint for implementation.

As implementation proceeds, we must make decisions about the physical packaging of our system into modules, and the allocation of processes to processors. These are both decisions of relationships, which we can express in module and process diagrams. Our data dictionary is updated as part of this step as well, to reflect the allocation of classes and objects to categories and modules to subsystems.

1.3.3.3 Activities

There are three activities associated with this step: the specification of associations, the identification of various collaborations, and the refinement of associations.

The identification of associations is primarily an analysis and early design activity.. A typical order of events for this activity might be the following:

- Collect a set of classes at a given level of abstraction, or associated with a particular family of scenarios;.

- Consider the presence of a semantic dependency between any two classes, and establish an association if such a dependency exists.
- For each association, specify the role of each participant, as well as any relevant cardinality or other kind of constraint.
- Validate these decisions by walking through scenarios and ensuring that associations are in place that are necessary and sufficient to provide the navigation and behavior among abstractions required by each scenario.

Class diagrams are the primary model generated by this activity.

1.3.3.4 Milestones and Measures

We successfully complete this phase when we have specified the semantics and relationships among certain interesting abstractions sufficiently to serve as a blueprint for their implementation. Measures of goodness include cohesion, coupling, and completeness.

1.3.4 Implementing Classes and Objects

1.3.4.1 Purpose

During analysis, the purpose of implementing classes and objects is to provide a refinement of existing abstractions sufficient to unveil new classes and objects at the next level of abstraction, which we then feed into the following iteration of the micro process.

During design, the purpose of this activity is to create tangible representations of our abstractions in support of the successive refinement of the executable releases in the macro process.

1.3.4.2 Products

Decisions about the representation of each abstraction and the mapping of these representations to the physical model drive the products from this step. Early in the development process, we may capture these tactical representation decisions in the form of refined class specifications. Where these decisions are of general interest or represent opportunities for reuse, we also document them in class diagrams (showing their static semantics) and finite state machines or interaction diagrams (showing their dynamic semantics). As development proceeds, and as we make further bindings to the given implementation language, we begin to deliver pseudo- or executable code.

As development proceeds, we may use method-specific tools that automatically forward-engineer code from these diagrams, or reverse engineer them from the implementation.

As part of this step, we also update our data dictionary, including the new classes and objects that we discovered or invented in formulating the implementation of existing abstractions.

1.3.4.3 Activities

There is one primary activity associated with this step: the selection of the structures and algorithms that provide the semantics of the abstractions we identified earlier in the micro process. Whereas the first three phases of the micro process focus upon the outside view of our abstractions, this step focuses upon their inside view .

Typical order of events for this activity might be the following:

- For each class, identify the patterns of use among clients, in order to determine which operations are central, and hence should be optimized.

- Before choosing a representation from scratch, consider the use of protected or private inheritance for implementation, or the use of parameterized classes.
- Consider the objects to which we might delegate responsibility.
- If the abstraction's semantics cannot be provided through inheritance, instantiation, or delegation, consider a suitable representation from primitives in the language.
- Select a suitable algorithm for each operation.

1.3.4.4 Milestones and Measures

During analysis, we successfully complete this phase once we have identified all the interesting abstractions necessary to satisfy the responsibilities of higher-level abstractions identified during this pass through the micro process.

During design, we successfully complete this phase when we have an executable or near-executable model of our abstractions.

The primary measure of goodness for this phase is simplicity. Implementations that are complex, awkward, or inefficient are an indication that the abstraction itself is lacking, or that we have chosen a poor representation.

1.4 The Macro Development Process

Overview

The macro process serves as the controlling framework for the micro process. This broader procedure dictates a number of measurable products and activities that permit the development team to meaningfully assess risk and make early corrections to the micro process, so as to better focus the team's analysis and design activities. The macro process represents the activities of the entire development team on the scale of weeks to months at a time.

Many elements of the macro process are simply sound software management practice, and so apply equally to object-oriented as well as non-object-oriented systems. These include basic practices such as configuration management, quality assurance, code walkthroughs, and documentation. In the next chapter, we will address a number of these pragmatic issues in the context of object-oriented software development.

The macro process is primarily the concern of the development team's technical management, whose focus is subtly different than that of the individual developer. Both are interested in delivering quality software that satisfies the customer's needs.

However, end users could generally care less about the fact that the developers used parameterized classes and polymorphic functions in clever ways; customers are much more concerned about schedules, quality, and completeness, and rightfully so. For this reason, the macro process focuses upon risk and architectural vision, the two manageable elements that have the greatest impact upon schedules, quality, and completeness.

In the macro process, the traditional phases of analysis and design are to a large extent retained, and the process is reasonably well ordered. Fig Below illustrates, the macro process tends to track the following activities:

- Establish the core requirements for the software (conceptualization).
- Develop a model of the system's desired behavior (analysis).
- Create architecture for the implementation (design).
- Evolve the implementation through successive refinement (evolution).
- Manage post delivery evolution (maintenance).

For all interesting software, the macro process repeats itself after major product releases. The basic philosophy of the macro process is that of incremental development.. This approach is extremely well-suited to the object-oriented paradigm, and offers a number of benefits relative to risk management.

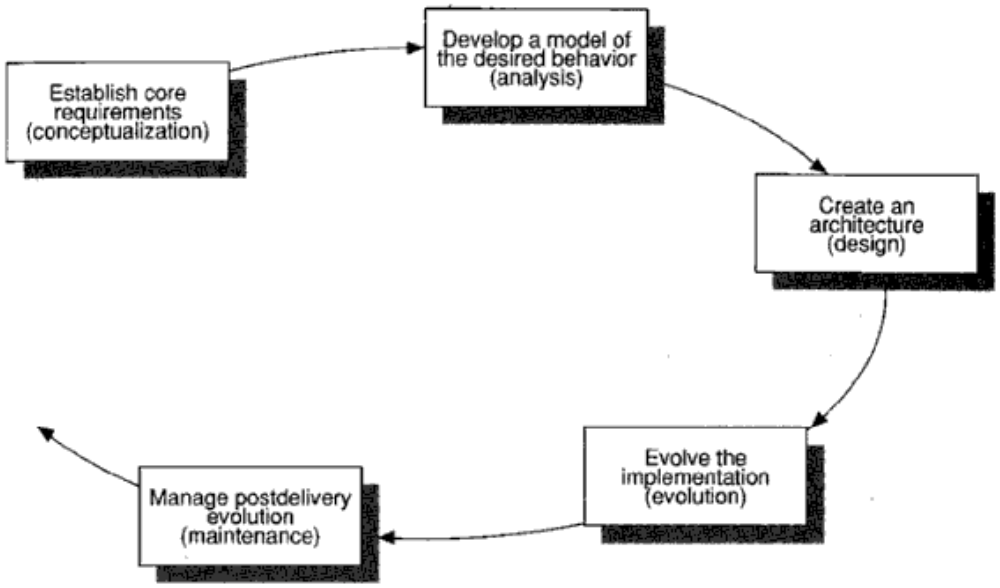


Fig: Macro Process

1.4.1 Conceptualization

1.4.1.1 Purpose

Conceptualization seeks to establish the core requirements for the system. For any truly new piece of software, or even for the novel adaptation of an existing system, there exists some moment in time where, in the mind of the developer, the architect, the analyst, or the end user, there springs forth an idea for some application.

This idea may represent a new business venture, a new complementary product in an existing product line, or perhaps a new set of features for an existing software system. It is not the purpose of conceptualization to completely define these ideas. Rather, the purpose of conceptualization is to establish the vision for the idea and validate its assumptions.

1.4.1.2 Products

Prototypes are the primary products of conceptualization. Specifically, for every significant new system, there should be some proof of concept, manifesting itself in the form of a quick-and-dirty prototype. Such prototypes are by their very nature incomplete and only marginally engineered.

It must be emphasized that all such prototypes are meant to be thrown away. Prototypes should not be allowed to directly evolve into the production system, unless there is a strongly compelling reason. Convenience for the sake of meeting a short-term schedule is distinctly not a compelling reason: this decision represents a false economy that optimizes for short term development, and ignores the cost of ownership of the software.

1.4.1.3 Activities

A typical order of events is the following:

- Establish a set of goals for the proof of concept, including criteria for when the effort is to be finished.
- Assemble an appropriate team to develop the prototype. Often, this may be a team of one (who is usually the original visionary). The best thing the development organization can do to facilitate the team's efforts is to stay out of its way.
- Evaluate the resulting prototype, and make an explicit decision for product development or further exploration. A decision to develop a product should be made with a reasonable assessment of the potential risks, which the proof of concept should uncover.

1.4.1.4 Milestones and Measures

It is important that explicit criteria be established for completion of a prototype. Proofs of concept are often schedule-driven (meaning that the prototype must be delivered on a certain date) rather than feature-driven. This is not necessarily bad, for it artificially limits the prototyping effort and discourages the tendency to deliver a production system prematurely.
Upper management can often measure the health of the software development organization by measuring its response to new ideas. Any organization that is not itself producing new ideas is dead, or in a moribund business. The most prudent action is usually to diversify or abandon the business. In contrast, any organization that is overwhelmed with new ideas and
Yet is unable to make any intelligent prioritization of them is out of control. Such organizations often waste significant development resources by jumping to product development too early, without exploring the risks of the effort though a proof of concept.

1.4.2 Analysis

1.4.2.1 Purpose

The purpose of analysis is to provide a description of a problem. The description must be complete, consistent, readable, and reviewable by diverse interested parties, and testable against reality. The purpose of analysis is to provide a model of the system's behavior. We must emphasize that analysis focuses upon behavior, not form. Analysis must yield a statement of what the system does, not how it does it. Any intentional statements of "how" during analysis should be viewed as useful only for the purpose of exposing the behavior of the system, and not as testable requirements of the design.

In this regard, the purposes of analysis and design are quite different. In analysis, we seek to model the world by identifying the classes and objects (and their roles, responsibilities, and collaborations) that: form the vocabulary of the problem domain. In design, we invent the artifacts that provide the behavior that the analysis model requires. In this sense, analysis is the phase that first brings together the users and developers of a system, uniting them with a common vocabulary drawn from the problem domain.

1.4.2.2 Products

The output of analysis is a description of the function of the system, along with statements about performance and resources required. In object-oriented development, we capture these descriptions through scenarios, where each scenario denotes some particular function point. We

use primary scenarios to illustrate key behaviors, and secondary scenarios to show behavior under exceptional conditions.
A secondary product of analysis is a risk assessment that identifies the known areas of technical risk that may impact the design process. Facing up to the presence of risks early in the development process makes it far easier to make pragmatic architectural trade-offs later in the development process.

1.4.2.3 Activities

Two primary activities are associated with analysis: domain analysis and scenario planning. domain analysis seeks to identify the classes and objects that are common to a particular problem domain..
Scenario planning is the central activity of analysis. Interestingly, there appears to be a confluence of thought about this activity among other methodologists, especially Rubin and Goldberg, Adams, Wirfs-Brock, Coad, and Jacobson.
A typical order of events for this activity follows:

- Identify all the primary function points of the system and, if possible, group them into clusters of functionally related behaviors.
- For each interesting set of function points, storyboard a scenario, using the techniques of use-case and behavior analysis . CRC card techniques are effective in brainstorming about each scenario.
- As needed, generate secondary scenarios that illustrate behavior under exceptional conditions.
- Where the life cycle of certain objects is significant or essential to a scenario, develop a finite state machine for the class of objects.
- Scavenge for patterns among scenarios, and express these patterns in terms of more abstract, generalized scenarios, or in terms of class diagrams showing the associations among key abstractions.
- Update the evolving data dictionary to include the new classes and objects identified for each scenario, along with their roles and responsibilities.

1.4.2.4 Milestones and Measures

We successfully complete this phase when we have developed and signed off on scenarios for all fundamental system behaviors. By signed off we mean that the resulting analysis products have been validated by the domain expert, end user, analyst, and architect; by fundamental we refer to behaviors that are central to the application's purpose.
Another important milestone of analysis is delivery of a risk assessment, which helps the team to manage future strategic and tactical tradeoffs.

1.4.3 Design

1.4.3.1 Purpose

The purpose of design is to create architecture for the evolving implementation, and to establish the common tactical policies that must be used by disparate elements of the system. We begin the design process as soon as we have some reasonably complete model of the behavior of the system.

1.4.3.2 Products

There are two primary products of design: a description of the architecture, and descriptions of common tactical policies.
We may describe an architecture through diagrams as well as architectural releases of the system. At the architectural level, it is most important to show the clustering of classes into class categories (for the logical architecture) and the clustering of modules into subsystems (for the physical architecture). We may deliver these diagrams as part of a formal architecture document, which should be reviewed with the entire team and updated as the architecture evolves.

1.4.3.3 Activities

There are three activities associated with design: architectural planning, tactical design, and release planning.
Architectural planning involves devising the layers and partitions of the overall system.
Architectural planning encompasses a logical decomposition, representing a clustering of classes, as well as a physical decomposition, representing a clustering of modules and the allocation of functions to different processors.
A typical order of events for this activity is as follows:

- Consider the clustering of function points from the products of analysis, and allocate these to layers and partitions of the architecture. Functions that build upon one another should fall into different layers; functions that collaborate to yield behaviors at a similar level of abstraction should fall into partitions, which represent peer services.
- Validate the architecture by creating an executable release that partially satisfies the semantics of a few interesting system scenarios as derived from analysis.
- Instrument that architecture and assess its weakness and strengths. Identify the risk of each key architectural interface so that resources can be meaningfully allocated as evolution commences.

The focus of architectural planning is to create very early in the life cycle a domain-specific application framework that we may successively refine.
Tactical design involves making decisions about the myriad of common policies.

- Relative to the given application domain, enumerate the common policies that must be addressed by disparate elements of the architecture. Some such policies are foundational, meaning that they address domain-independent issues such as memory management, error handling, and so on. Other policies are domain-specific, and include idioms and mechanisms that are germane to that domain, such as control policies in real-time systems, or transaction and database management in information systems. For each common policy, develop a scenario that describes the semantics of that policy.
- Further capture its semantics in the form of an executable prototype that can be instrumented and refined.
- Document each policy and carry out a peer walkthrough, so as to broadcast its architectural vision.

Release planning sets the stage for architectural evolution. Taking the required function points and risk assessment generated during analysis, release planning serves to identify a controlled series of architectural releases, each growing in its functionality, ultimately encompassing the requirements of the complete production system.
A typical order of events for this activity is as follows:

- Given the scenarios identified during analysis, organize them in order of foundational to peripheral behaviors. Prioritizing scenarios can best be accomplished with a team including a domain expert, analysis, architect, and quality-assurance personnel.
- Allocate the related function points to a series of architectural releases whose final delivery represents the production system.
- Adjust the goals and schedules of this stream of releases so that delivery dates are sufficiently separated to allow adequate development time, and so that releases are synchronized with other development activities, such as documentation and field testing.
- Begin task planning, wherein a work breakdown structure is identified, and development resources are identified that are necessary to achieve each architectural release.

A natural by-product of release planning is a formal development plan, which identifies the stream of architectural releases, team tasks, and risk assessments

1.4.3.4 Milestones and Measures

We successfully complete this phase when we have validated the architecture through a prototype and through formal review. In addition, we must have signoff on the design of all primary tactical policies, and a plan for successive releases.

The primary measure of goodness is simplicity. A good architecture is one that embodies the characteristics of organized complex systems. The main benefits of this activity are the early identification of architectural flaws and the establishment of common policies that yield a simpler architecture.

1.4.4 Evolution

1.4.4.1 Purpose

The purpose of the evolutionary phase is to grow and change the implementation through successive refinement, ultimately leading to the production system. The evolution of architecture is largely a matter of trying to satisfy a number of competing constraints, including functionality, time, and space: one is always limited by the largest constraint. For example, if the weight of the computer is a critical factor (as it is in spacecraft design), then the weight of individual memory chips must be considered, and in turn the amount of memory permitted by the weight allowance limits the size of the program that may be loaded. Evolutionary development focuses upon designing for functionality first and for local performance second..

1.4.4.2 Products

The primary product of evolution is a stream of executable releases representing successive refinements to the initial architectural release. Secondary products include behavioral prototypes that are used to explore alternative designs or to further analyze the dark corners of the systems' functionality. These executable releases follow the schedule established in the earlier activity of release planning.

Between each successive external release, the development team may also produce behavioral Prototypes.

1.4.4.3 Activities

Two activities are associated with evolution: application of the micro process, and change management.

The work that is carried out between executable releases represents a compressed development process, and so is essentially one spin of the micro process. This activity begins with an analysis

of the requirements for the next release, proceeds to the design of architecture, and continues with the invention of classes and objects necessary to implement this design.
A typical order of events for this activity is as follows:

- Identify the function points to be satisfied by this executable release, as well as the areas of highest risk, especially those identified through evaluation of the previous release.
- Assign tasks to the team to carry out this release, and initiate one spin of the micro process..
- As needed to understand the semantics of the system's desired behavior, assign developers to produce behavioral prototypes.
- Force closure of the micro process by integrating and releasing the executable release.

Change management exists in recognition of the incremental and iterative nature of object oriented systems. It is tempting to allow undisciplined change to class hierarchies, class protocols, or mechanisms, but unrestrained change tends to rot the strategic architecture and leads to thrashing of the development team.

In practice, we find that the following kinds of changes are to be expected during the evolution of a system:

- Adding a new class or a new collaboration of classes
- Changing the implementation of a class
- Changing the representation of a class
- Reorganizing the class structure
- Changing the interface of a class

1.4.4.4 Milestones and Measures

We successfully complete this phase when the functionality and quality of the releases are sufficient to ship the product. The releases of intermediate executable forms are the major milestones we use to manage the development of the final product.

The primary measure of goodness is therefore to what degree we satisfy the function points allocated to each intermediate release, and how well we met the schedules established during release planning. Two other essential measures of goodness include tracking defect discovery rates, and measuring the rate of change of key architectural interfaces and tactical policies.

1.4.5 Maintenance

1.4.5.1 Purpose

Maintenance is the activity of managing post delivery evolution. This phase is largely a continuation of the previous phase, except that architectural innovation is less of an issue. Instead, more localized changes are made to the system as new requirements are added and lingering bugs stamped out.

- A program that is used in a real-world environment necessarily must change or become less and less useful in that environment (the law of continuing change).
- As an evolving program changes, its structure becomes more complex unless active efforts are made to avoid this phenomenon

1.4.5.2 Products

Since maintenance is in a sense the continued evolution of a system, its products are similar to those of the previous phase. In addition, maintenance involves managing a punch list of new tasks. Additionally, as more users exercise the system, new bugs and patterns of use will be uncovered that quality assurance could not anticipate. A punch list serves as the vehicle for collecting bugs and enhancement requirements, so that they can be prioritized for future releases.

1.4.5.3 Activities

Maintenance involves activities that are little different than those required during the evolution of a system. Especially if we have done a good job in the original architecture, adding new functionality or modifying some existing behavior will come naturally.

In addition to the usual activities of evolution, maintenance involves a planning activity that prioritizes tasks on the punch list. A typical order of events for this activity is as follows:

- Prioritize requests for major enhancement or bug reports that denote systemic problems, and assess the cost of redevelopment.
- Establish a meaningful collection of these changes and treat them as function points for the next evolution.
- If resources allow it, add less intense, more localized enhancements (the so-called low hanging fruit) to the next release.
- Manage the next evolutionary release.

1.4.5.4 Milestones and Measures

The milestones of maintenance involve continued production releases, plus intermediate bug releases.

We know that we are still maintaining a system if the architecture remains resilient to change; we know we have entered the stage of preservation when responding to new enhancements begins to require excessive development resources.

Chapter - 6
PRAGMATICS

Software development today remains a very labour tritensive business. Different designers can produce different enterprise models of the same enterprise.

The pragmatics of a method identifies suggestions and heuristics of how a notation should be used. It identifies which concept of the notation should be used to express different concerns (such as operations should express behavior and attributes should express states) and may suggests orders for the development (such as identify the different classes, then establish the state they capture in terms of attributes and finally determine the operations).

Management and Planning

It is necessary to have strong project leadership that actively manages and directs a project's activities.

Risk Management: The responsibility of software development manager is to manage technical as well as nontechnical risk. Technical risks in object oriented systems include problems such as the selection of an inheritance lattice that offers the best compromise between usability and flexibility or the choice of mechanisms that field acceptable performance while simplifying the system's architecture . Nontechnical risks encompass issues such as supervising the timely delivery of software from a third party vendor or managing the relationship between the customer and the development teams, so as to facilitate the discovery of the system's real requirements during analysis.

Task Planning: Object oriented software development requires that individual developers have unscheduled critical masses of time in which they can think, innovate and develop and meet informally with other team members as necessary to discuss detailed technical issues. The management team must plan for this unstructured time. These meetings may result in small adjustments to work assignments, so as to ensure steady progress. No project can afford for any of its developer to sit idle while waiting for other team members to stabilize their part of architecture. In task planning, first, the management team directs the energies of a developer to a specific part of the system, say, for example, the design of a set of classes for interfacing to a relational database. The developer considers the scope of the effort, and returns with an estimate of time to complete; which management team relies upon to schedule other developer's activities. The problem is that these estimates are not always reliable, because they usually represent best case conditions. One developer may quote a week of effort the same task. When the work is actually carried out, it might take both developers three weeks. The first developer having under-estimates. Management must help its developers learn to do effective planning.

Walkthroughs: Management must take steps to strike a balance between too many and too few walkthroughs. It is simply ton economical to review every line of code. Therefore, management must direct the scarce resources of its team to review those aspects of the system that represent strategic development issues. For object oriented systems, this suggests conducting formal renews upon system's architecture. Scenarios are a primary product of analysis phase of object

oriented development. Formal reviews of scenarios are led by the team's analysis, together with domain expert or other end users and are witnessed by other developers. Even nonprogrammers can understand scenarios presented through conpits or the formalisms of object diagrams. Such reviews help to establish a common vocabulary among a system's developers and its users.
Architecture reviews should focus upon the overall structure of the system, including its class structure and mechanisms. Visibility of architecture creates opportunities for discovering patterns of classes or collaborations of objects.

Staffing

Resource allocation: One aspect of managing object oriented projects is that is the steady state, there is usually a reduction in the total amount of resources needed and a shift in the timing of their deployment relative to more traditional methods. The first object oriented project undertaken by on organization will require slightly more resources than for non-object oriented methods. Resources benefits more in second or third or rater projects at which time development team and management team are comfortable to work. Testing may require fewer resources primarily because adding new functionality to a class or mechanism is achieved mainly by modifying a structure that is known to behave correctly in the first place. Thus, in the steady state, the net of all human resources required for traditional approaches.
Development Team Roles: In software development, developers are not interchangeable parts, and the successful development of any complex system requires the unique and varied skills of a focused team of people. Following three rules are found to be central to an object oriented project.

- Project architect
- Subsystem lead
- Application engineer

The project architect is the visionary and is responsible for evolving and maintaining the system's architecture. The project architect is not necessary the most senior developer but rather is best qualified to make strategic decisions, as a request of his experience in building similar kinds of systems. Architects are not necessarily the best programmer but should have adequate programming skills, well versed in the notation and process of object oriented development. Instead of hiring, architect should be same throughout the system.
Subsystem leads are the primary abstractionists of the project. A subsystem lead is responsible for the design of an entire class category or subsystem. Subsystem lead must work in conjunction with project architect and must be well versed in the notation and process of object oriented development and are faster and better programmers than project architect.
Application engineers are the less senior developers and are responsible for the implementation of a category or subsystem under the supervision if its subsystem lead and other application engineers are responsible for the taking the classes designed by architects. Application engineers are familiar with but not necessarily experts in the notation and process of object oriented development. Most software development organization have really good designers and many more less experienced ones. Junior developers work under guidance of senior developers. In large project, there may be a no of other roles
Some of the roles are:

i) Project Manager: Responsible for active management of the project's deliverables, tasks, resources and schedules.
ii) Analyst: Responsible for evolving and interpreting the end user's requirements must be an expert in the problem domain.
iii) Reuse Engineer: Responsible for managing the project's repository of components and designs. Acquires produces and adapts components for general use within the project or the entire organization.
iv) Duality assurance: Responsible for measuring the products of the development process: generally directs system level testing of all prototypes and production releases.
v) Integration manager: Responsible for assembling compatible versions of released categories and subsystems in order to form a deliverable release. Also responsible for maintaining the configurations of released products.
vi) Documenter: Responsible for producing end user documentation of the product and its architecture.
vii) Toolsmith: Responsible for creating and adapting software tools that facilitate the production of the project's deliverables, especially with regard to generated code.
viii) System administration: Responsible for managing the physical computing resources used by the project.

RELEASE MANAGEMENT

Integration: At any given time in the development process, there will be multiple prototypes and production releases as well as development and test scaffolding. Most often each developer will have his or her own executable new of the system under development there will be generally many smaller integrations events, each marking the creation of another prototype or architectural release. Each release is generally incremental in nature, having evolved from an earlier stable release. For larger projects, an organization may produce an internal release of the system every few weeks and then release a running version to its customer for review every few months, according to the needs of the project. In the steady state, a release consists of a set of compatible subsystems along with their associated documentation. Building a release is possible whenever the major subsystems of a project are stable enough and work together well enough to provide some new level of functionality.

Configuration Management and Version Control: An individual developer must have a working version of a particular subsystem. i.e. a version under development. The interfaces of all imported subsystems must be available in order to proceed with further development. As this working version becomes stable, it is released to an integration team, responsible or collecting a set of compatible subsystems for the entire system. This forms the internal release which becomes the current operational release. The release is visible to all active developers who need to further refine their particular part of its implementation. In the meantime, individual developer can work on a newer version of his or her subsystem and development can proceed in parallel. At any given point in the evolution of a system, multiple versions of a particular subsystem may exist. There might be a version for the current release under development; one for the current internal release and one for the latest customer release.

Testing: In context of object oriented architecture, testing must encompass at least three dimensions.
Unit testing: Involves testing individual classes and mechanisms by application engineer (who implemented the structure).
Subsystem testing: Involves testing a compute category or subsystem by subsystem lead and it is regression tests for newly released version of the subsystem.
System testing: Involves testing the system as a whole by quality assurance team.
Testing should focus upon the system's external behavior.

REUSE

Elements of Reuse: Code, designs, scenarios and documentation can be reused. Classes, objects and designs in the form of idioms, mechanisms and frameworks, can be reused. Reuse factors are up to 70% in successful projects. Any amount of reuse is better than none because reuse represents a saving of resources.
Institutionalizing Reuse: Reuse must be institutionalized i.e. opportunities for reuse program is best achieved by making specific individuals responsible for the reuse activity. Reuse activity involves identifying opportunities for commonality, usually discovered through architectural reviews and exploiting these opportunities, usually by producing new components or adapting existing ones and implementing their reuse among developers. A reuse activity will only be successful in an organization that takes a long term new of software development and optimizes resources for more than just the current project.

Quality Assurance and Metrics

Software Quality: Software quality is the fitness for use of the total software product. Object oriented technology does not automatically lead to quality software. It is still possible to write very bad software using object oriented programming language. A simple, adaptable software architecture is control to any quality software. Its quality is made compute by carrying out simple and consistent tactical design decisions.
Software quality assurance involves the systematic activities providing evidence of the fitness for use of the total software product. Quality assurance seeks to give us quantifiable measures of goodness for the quality of a software system. The most quantifiable measure of goodness is the defect-discovery rate. During the evaluation of the system, we track software defects according to their severity and location. The defect discovery rate (i.e. how quickly errors are being discovered) is plotted against time. A project that is under control will have a bell shaped curve, with defect discovery rate peaking at around the midpoint of the test period and then falling off to zero. A project that is out of control will have a curve that tails off very slowly or not at all.
Defect density is another quality measure. Measuring approach and is still generally applicable to object oriented systems. In healthy projects, defect density tends to reach a stable value after approximately 10,000 lines of code have been infected and will remain almost unchanged no matter how large the code volume is thereafter. Defect density in terms of the numbers of defects per class category or per class 80/20 rule seems to apply i.e. 80% of the software defects will be found in 20% of the system's classes.

Object oriented Metrics: Manager can be measure the progress by measuring the lines of code produced. Source code has no correlation to its completeness or complexity. We tend to measure process by counting the classes in the logical design or the modules in the physical design, that are completed and working.

The metrics that are directly applicable to object oriented systems are:

i) Weighted methods per class
ii) Depth of inheritance tree
iii) Number of Children
iv) Coupling between objects
v) Response for a class
vi) Lack of cohesion in methods

Weighted methods per class gives a relative measure of the complexity of an individual class; if all methods are considered to be equally complex, this becomes a measure of the number of methods per class. The depth of inheritance tree and number of children are measures of the shape and size of the class structure. Well structured object oriented systems tend to be architected as forests of classes, rather than as one very large inheritance lattice coupling between objects is a measure of their correctness to other objects. Response for a class is a measure of the methods that its instances can call. Cohesion in methods is a measure of the unity of the class's abstraction. A class with low cohesion among its methods suggests an accidental or in appropriate abstraction.

Documentation

Development Legacy: The development of software system involves much more than the writing of its row source code. Products of object oriented development in general include sets of class diagram, object diagrams, module diagrams and process diagrams. All diagrams leads to the development of the system.

Documentation Contents: Documentation is an essential, and product of the development process. The documentation of a system's architecture and implementation is important. Documents are living product that should be allowed to evolve together with the iterative and incremental evaluation of the project's releases. End user documentation must be produced instructing the user on the operation and installation of each release. Analysis documentation must be produced to capture the semantics of the system's function points as viewed through scenarios. We must also generate architectural and implementation documentation.

The essential documentation of a system's architecture and implementation should include the following:

i) Documentation of the high level system architecture.
ii) Documentation of the key abstractions and mechanisms in the architecture.
iii) Documentation of scenarios that illustrate the as-built behavior of key aspects of the system.

Chapter - 7
APPLICATIONS
DATA ACQUISITION: WEATHER MONITORING STATION

Weather Monitoring Station Requirements

This system shall provide automatic monitoring of various weather conditions. It must measure:

(i) Wind speed and direction
(ii) Temperature
(iii) Barometric pressure
(iv) Humidity

The system shall also provide following derived measurements.

(i) Wind chill
(ii) Dew point temperature
(iii) Temperature trend
(iv) Barometric Pressure trend

The system shall have display that continuously indicates all eight primary and derived measurements along with time and data.

Analysis

Analysis is begun by considering the hardware on which software must execute. We will make the following strategic assumptions.

- The processor (i.e., CPU) may take the form of a PC or a handheld device.
- Time and date are supplied by an on-board clock, assessable via memory mapped 1/0
- Temp, barometric pressure and humidity are measured by on board circuits (with remote sensors) and accessible via memory mapped 1/0.
- Wind direction and speed are measured from a boom encompassing a wind vane (capable of sensing wind from any of 16 directions) and cups (which advance a counter for each revolution)
- User i/p is provided through an off the shelf telephone keypad, managed by an on board circuit supplying audible feedback for each key press.
- The display is an off-the-self LCD graphic device, managed by an on board circuit capable of processing a simple set of graphic primitives.
- An on board timer interrupts the computer every 1/60 second.

Operations related to date and time are current time and current data. Further analysis of date and time would allow a client to close either 12 or 24 hour format

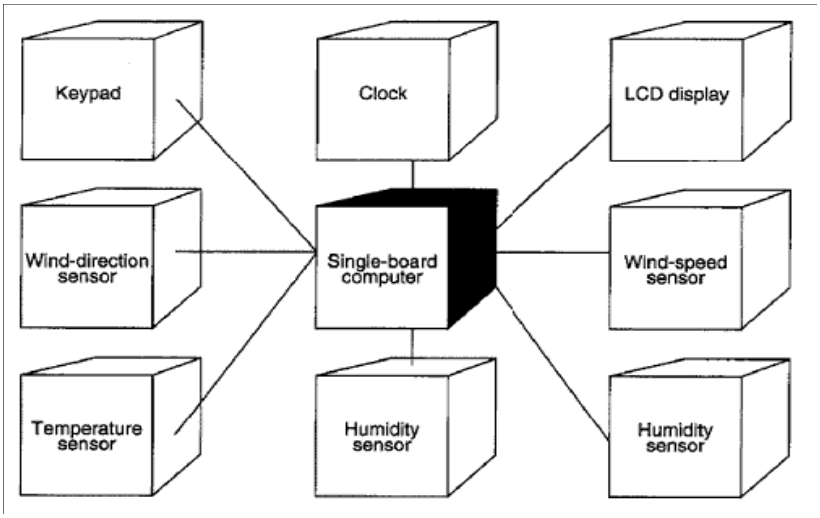


Fig: The Deployment Diagram for the Weather Monitoring System

Abstraction of time/date class can be summarized as follows:
Name: Time Date
Responsibilities: keep track of the current time and date.
Operations: current time, current date, set format, set hour, set minute, set second, set month, set day, set year.
Attribute: time date

The **class temperature sensor** serves as an along to the hardware temp sensors in the system.
Name: Temperature Sensor
Responsibilities: Keep track of the current temperature.
Operations: Current Temperature, set low temperature, Set high Temperature
Attributes: temperature

Abstraction of the barometric pressure:
Name: Pressure sensor
Responsibilities: Keep track of the current barometric pressure
Operations: Current Pressure, Set high pressure, Set low pressure
Attributes: Pressure

Abstraction of Humidity sensor:
Name: Humidity sensor
Responsibilities: Keep track of the current humidity, expressed as a percentage of saturation from 0% to 100%
Operations: current Humidity, set low humidity, set high humidity
Attributes: Pressure

A renew of system's requirements suggests some behavior common behavior is captured in the following specification.
Responsibilities: Report the highest and lowest value over a 24 hour period
Operations: high value, low value, time of high value, time of low value

Abstraction of wind speed sensor:
Name: Wind speed sensor
Responsibilities: Keep track of the current wind speed
Operations: current speed, set low speed, set high speed
Attributes: speed

Abstraction for wind direction:
Name: Wind direction sensor
Responsibilities: Keep track of the current wind direction
Operations: Current Direction
Attributes: direction.

We can translate these requirements into the following
Class Specification:
Name: LDC Device
Responsibilities: Manage the LDC device and provide service for displaying certain graphics elements.
Operations: draw text, draw line, draw circle, set text style.

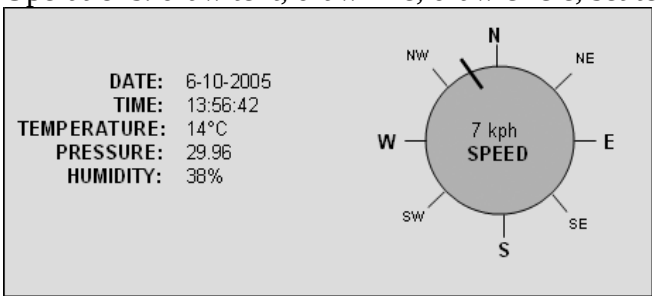


Fig: The Display for the Weather Monitoring System

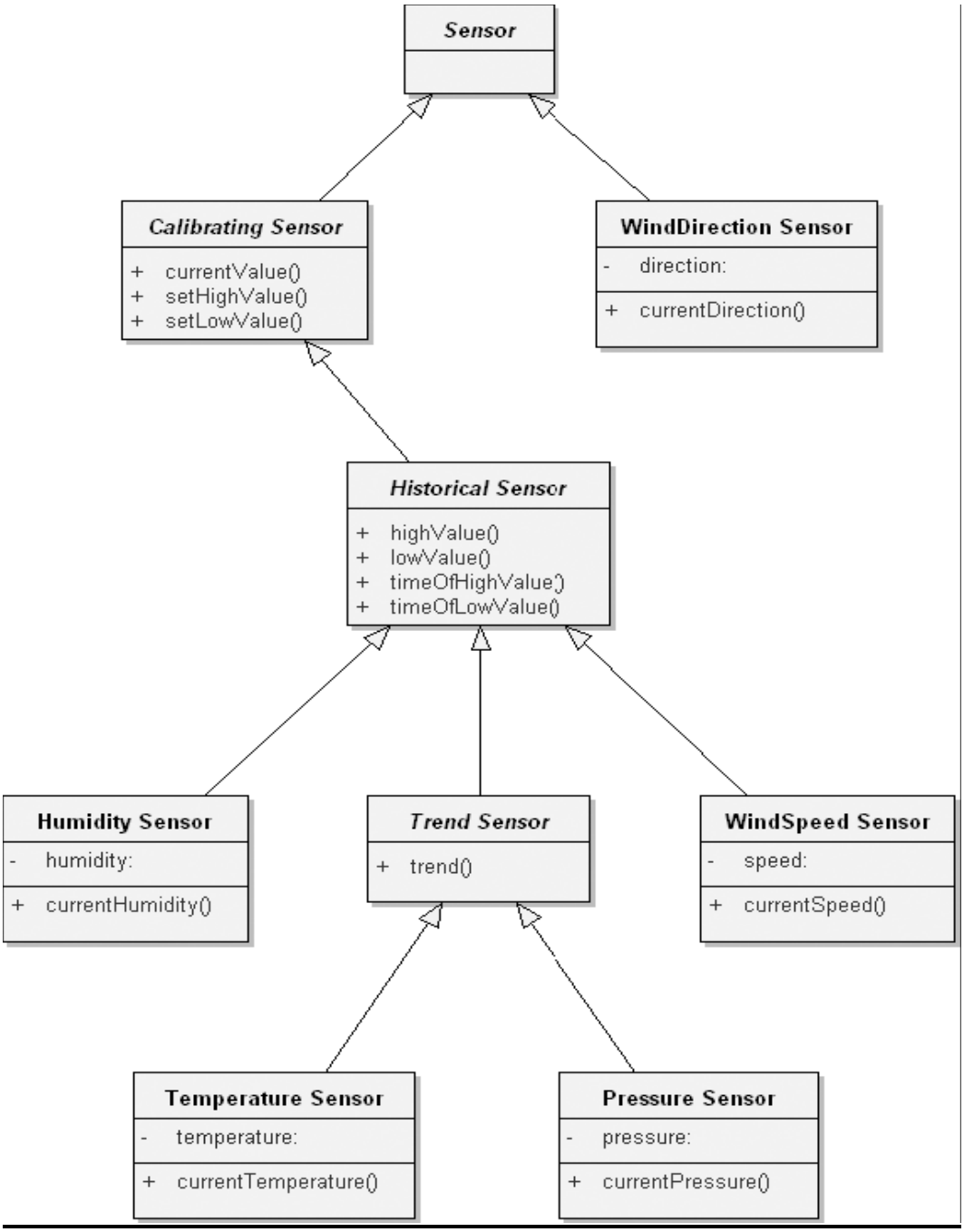


Fig: The Hierarchy of the Sensor Class

SCENARIOS

We continue our analysis by studying several scenarios of its use. A no. of use cases, as viewed from the clients of the system are:

- Monitoring basic weather measurements including wind speed and direction, temperature, barometric pressure and humidity.

- Monitoring derived measurements including wind chill, dew point, temperature trend and barometric pressure trend.
- Displaying the highest and lowest value of a selected measurement.
- Setting the time and date.
- Calibration of a selected sensor
- Powering up the system

We add to this list two secondary use cases:

- Power failure
- Sensor Failure

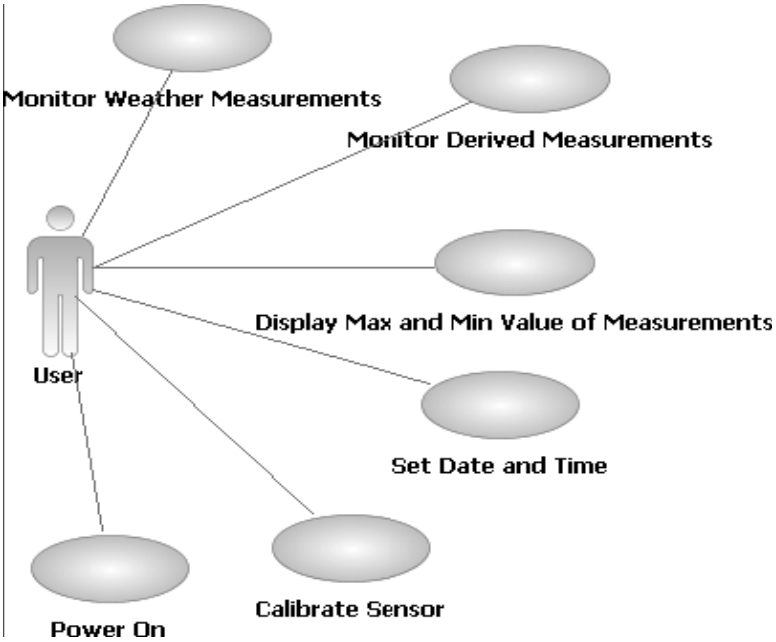


Fig: Primary Use Cases for the Weather Monitoring System

Sampling rates are sufficient to capture changing conditions as follows:
Every 0.1 second – wind direction
Every 0.5 second – wind speed
Every 5 minutes – temperature, barometric pressure and humidity

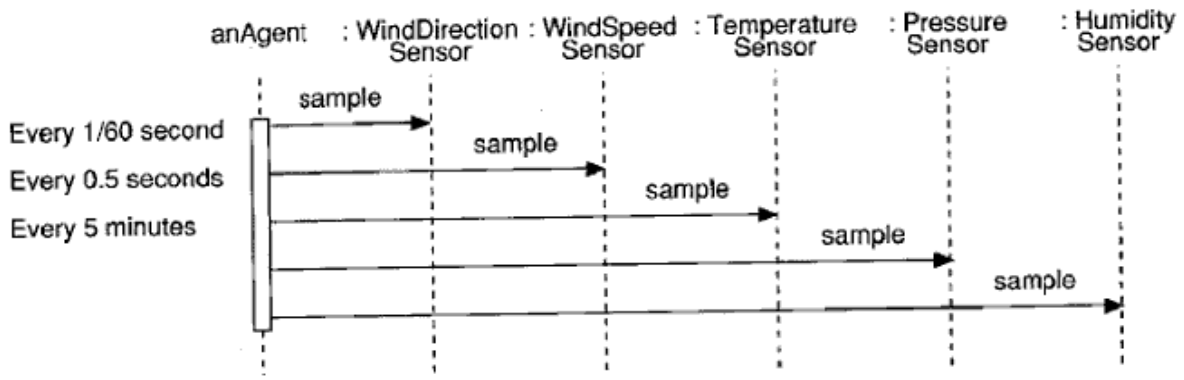


Figure: Scenario for monitoring basic measurements

We can add following class specification to our products of analysis:
Name: Display Manager
Responsibilities: Manage layout or items or the LCD device
Operation: Draw static items, display time, display date, display temperature, display humidity, display pressure, display wind chill, display dew point, display wind speed, display wind director, display high, display low.

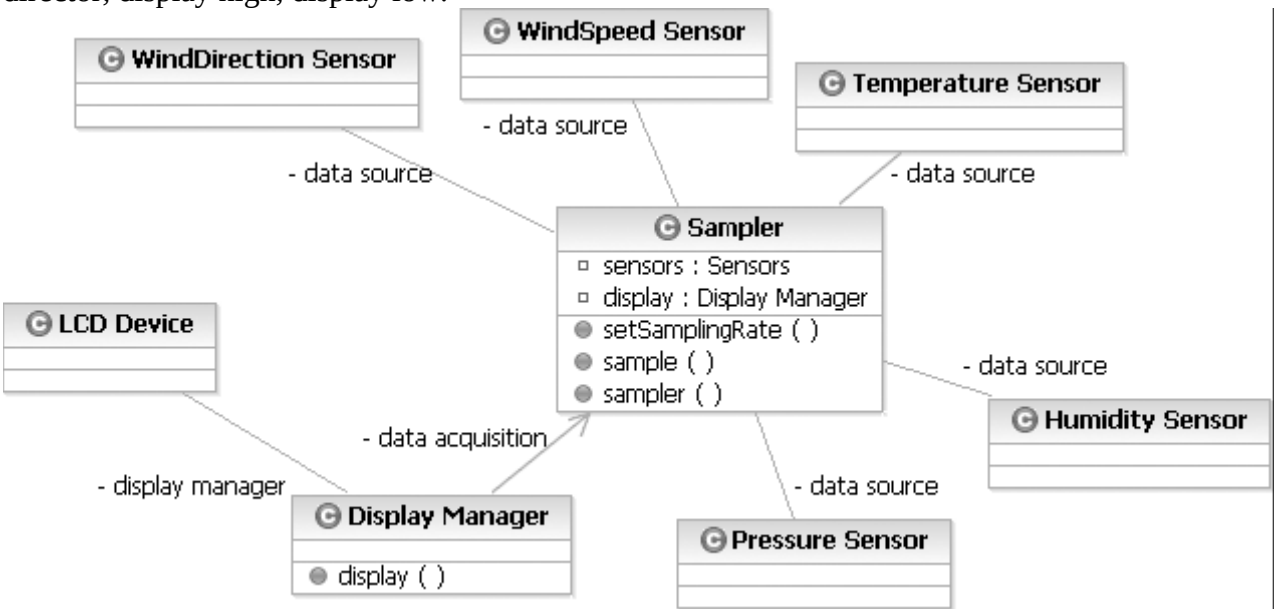


Figure: Sample and Display classes

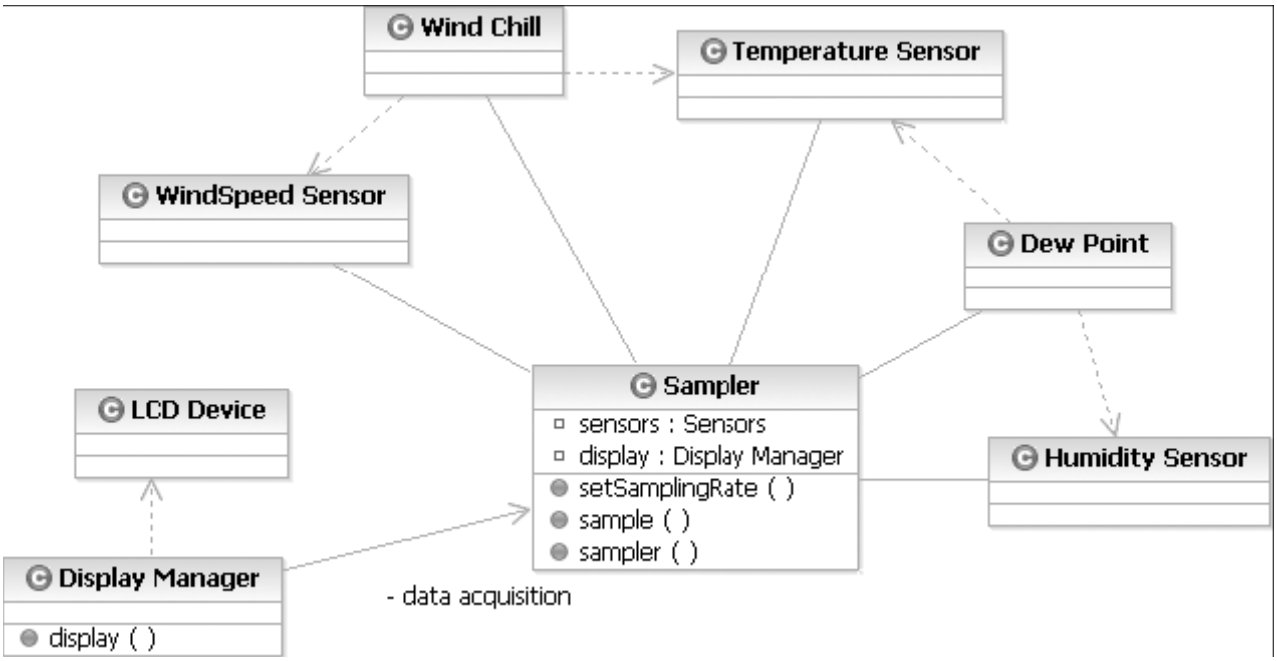


Figure: derived measurements classes

Fig: Weather monitoring system user keypad:

Calibrating a particular sensor follows a related pattern of user gestures:
Calibrating a sensor:

- User presses the CALIBRATE key
- System displays CALIBRATING
- User processes any one of the keys WIND SPEED, TEMPERATURE, PRESSURE or HUMIDITY and other keypress (excepted RUN) is ignored.
- System flashes the corresponding label.
- User presses the UP or DOWN keys to select high or low calibration point.
- Display flashes the corresponding value
- User presses the UP or DOWN keys to adjust the selected value
- Control passes back to step 3 or 4.

Input manager is responsible for carrying out the managing and dispatching user input. Abstraction of input manger is as follows:
Name: Input manager
Responsibilities: Manage & dispatch user input

Operations: Process key process

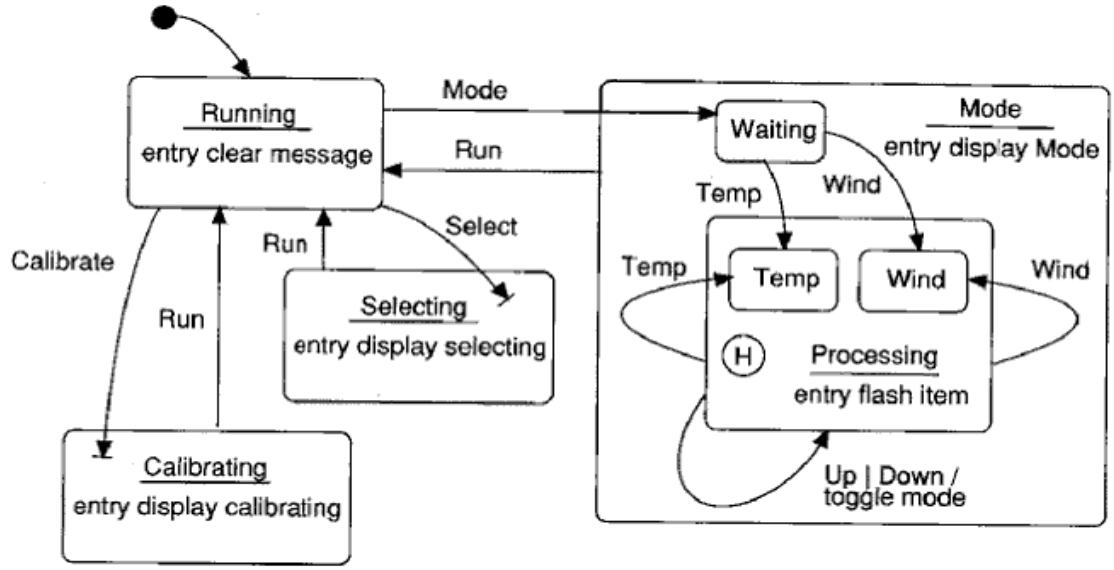


Fig: InputManager State Transition Diagram

Final scenario involves powering up the system – we may write a script for our analysis of scenario as follows:

- Power is applied
- Each sensor is constructed; historical sensors clear their history and trend sensors prime their slope calculating algorithms.
- The user input buffer is initialized, causing garbage key presses (due to noise upon power up) to be described.
- The static elements of the display are drawn.
- The sampling process is initiated.

DESIGN

Architectural Framework: System architecture is designed this model takes time and divides into several frames. E.g. we might sample wind direction every 10 frames, but sample wind speed only every 30 frames.

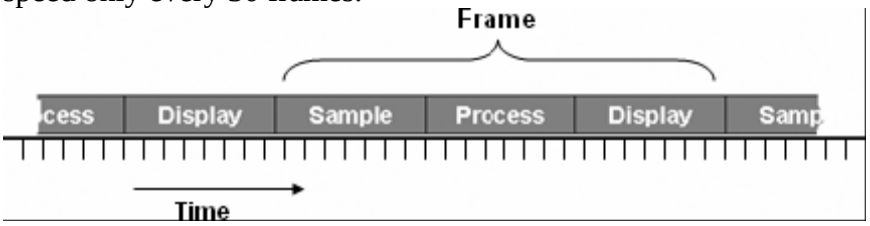


Fig: Time-Frame Processing

In the class diagram, one new class sensors is invented whose responsibility is to serve as the collection of all the physical sensors in the system.

In the class diagram, one new class sensors is invented whose responsibility is to serve as the collection of all the physical sensors in the system.

Frame mechanism: Central behavior of this architecture is carried out by a collaboration of the sample and timer classes. 11 clock ticks, measures in 1/60 second

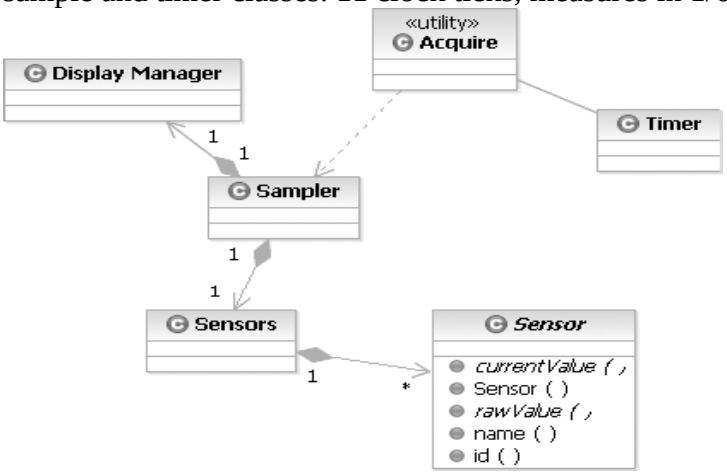


Fig: Frame Mechanism

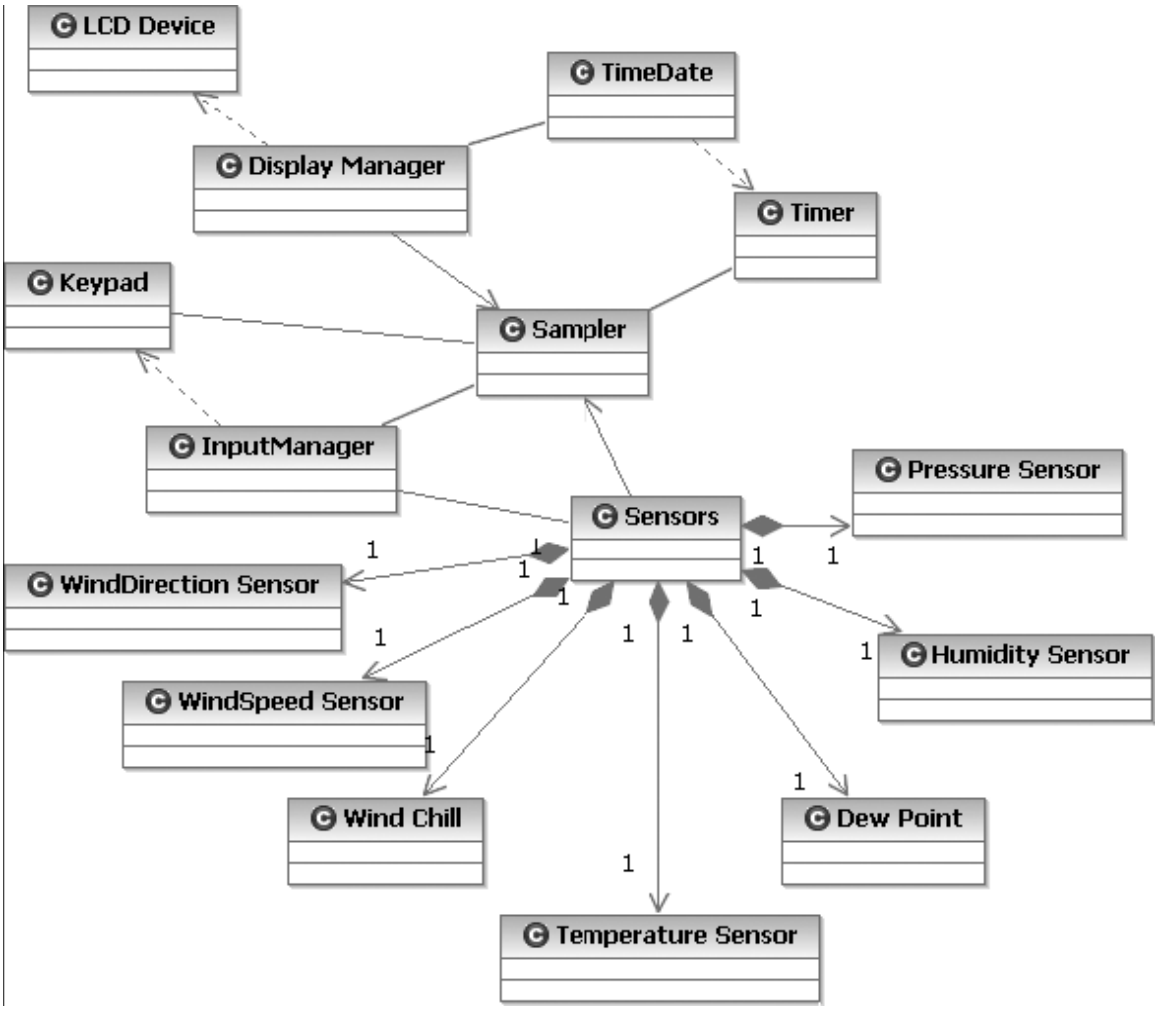


Fig: The Architecture of the Weather Monitoring System

EVOLUTION

Release Planning: Start the process using a sequence of release as follows:

(i) Develop the minimal functionality release which monitors just one sensor.
(ii) Complete the sensor hierarchy
(iii) Complete the classes responsible for managing the display
(iv) Complete the classes responsible for managing the user interface.

Sensor Mechanism: We can complete design of sensor class as follows:

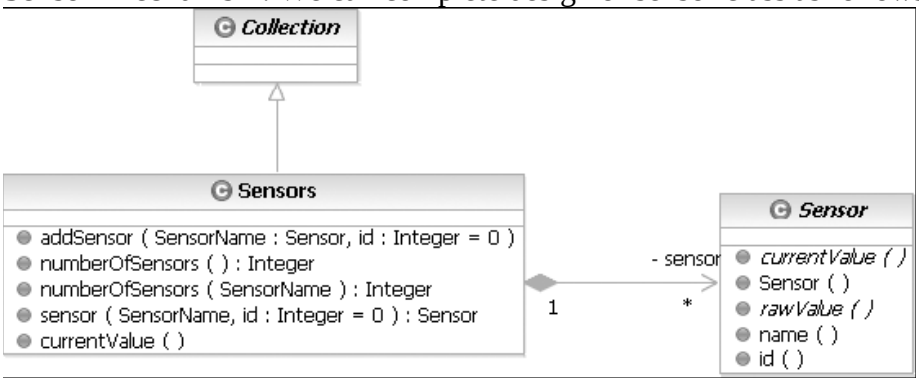


Fig: The Interface of the Sensors class

Now, immediate subclass calibrating sensor can be inherited from sensor class. Similarly, subclass historical sensor can be inherited from calibrating sensor. Trend sensor inherits from historical sensor and temperature sensor inherits from trend sensor.

DISPLAY MECHANISM

Implementing the next release completes the display manager and LCD device.

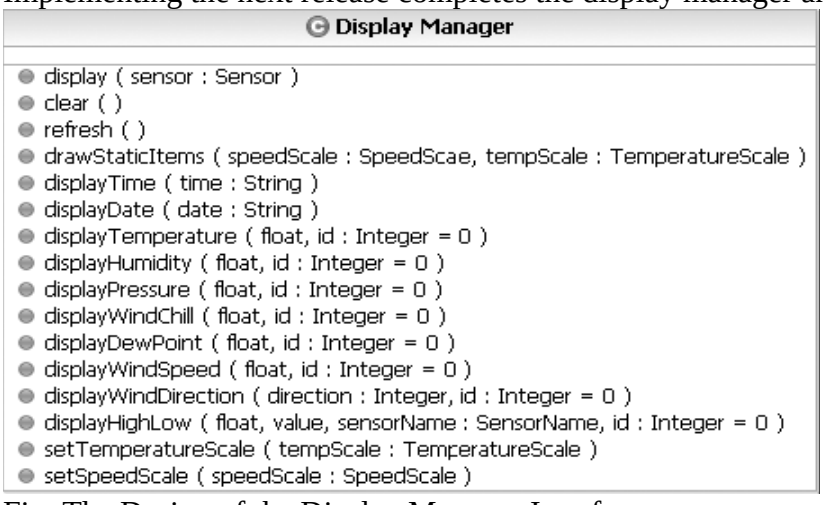


Fig: The Design of the Display Manager Interface

User interface Mechanism

Define and construct classes related to:
Keypad, Input Manager, Sampler

Maintenance

User may want to measure rainfall as well. So, implement this new feature, we must

- Create a new class rainfall sensor and insert it in proper place in the sensor class hierarchy.
- Update the enumeration sensor name.
- Update the display manager so that it knows how to display values of this sensor.
- Update the input manager so that it knows how to evaluate the newly defined key rainfall.
- Properly add instances of this class to the system's sensor collection.

Suppose we desire the ability to download a day's record of weather conditions to a remote computer to implement this:

- Create a new class serial port, responsible for managing on RS-232 port used for serial communication.
- Invert a new class report manager responsible for collecting the information required for the download.
- Modify the implementation of sampler: sample to periodically service the serial port.

Chapter – 7
Applications
FRAMEWORKS: FOUNDATION CLASS LIBRARY

A major benefit of object-oriented programming languages such as C++ and Smalltalk is the degree of reuse that can be achieved in well-engineered systems. A high degree of reuse means that far less code must be written for each new application; consequently, that is far less code to maintain. We can reuse individual lines of code (simplest form of reuse), specific classes or logically related societies of classes. A framework is a collection of classes that provide a set of classes and mechanisms that clients can use or adapt. Frameworks apply to a wide variety of applications. General foundation class libraries, math libraries and libraries for graphical user interfaces fall into this category. Whenever there exists a family of programs that all solve substantially similar problems, there is an opportunity for an application framework. Object oriented technology is applied to the creation of a foundation class library.

ANALYSIS

Defining the boundaries of the problem:

It is necessary to focus upon providing library abstractions and services that are of the most general use, rather than trying to make this a framework that is everything for everybody. One discovery we make in this analysis is the clear separation of structural abstractions (such as queues, stacks and graphs) versus algorithmic abstractions (such as storing, pattern matching and searching). We can devise classes whose instances are agents responsible for carrying out these actions. Thus, we choose to bound our problem by organizing our abstractions into one of two major categories.

i) **Structures:** contains all structural abstractions.
ii) **Tools:** contains all algorithmic abstractions

It is clear there is "USING RELANSHIPS" between two categories. Further analysis consists of study of the foundation classes used by production system in a variety of application areas. On the basis of this analysis, we may settle upon the following kinds of structured.

(i) Bags: Collection of (Possible duplicate)
(ii) Collections: Indexable collection of items
(iii) Deques: Sequence of items in which items may be added and removed from either end.
(iv) Graphs: Unrooted collection of nodes and arcs, which may contain cycles and cross references: structural sharing is permitted
(v) Lists: Rooted sequence of items: structure sharing is permitted
(vi) Maps: Dictionary of item/value pairs
(vii) Queues: Sequence of items in which items may be added from one end and removed from the opposite end.
(viii) Rings: Sequence of items in which items may be added and removed from the top of a circuit structure.
(ix) Sets: Collection of (unduplicated) items
(x) Stacks: Sequence of items in which items may be added and removed from the same end.

(xi) Strings: Indexable sequence of items, with behaviors involving the manipulation of sub strings.
(xii) Trees: Rooted collection of nodes and arcs which may not contain cycles or cross references, structural sharing is permitted

We choose this particular organizing of the abstraction because it offers a clear separation of behavior among each category of abstractions. We may also settle upon the following kinds of tools based upon our domain analysis.
(i) Date/Time – Operations for manipulating date and time.
(ii) Filters – Input, process and output transformations
(iii) Pattern matching – Operations for searching for items within structures
(iv) Storing – Operations for ordering structures.
(v) Utilities – Common composite operations that build upon more primitive structural operation
There are obvious functional variations for many of these abstractions.

Patterns: A framework consists of a collection of classes together with a number of patterns of collaboration among instance instances of these classes.
Analysis reveals that there are a no of important patterns essential to this foundation class library, encompassing the following issues.
(i) Time and space semantics
(ii) Storage management policies
(iii) Response to exceptional conditions
(iv) Idioms for iteration
(v) Synchronization in the presence of multiple threads of control

There may be two kinds of clients for each abstraction in this library. The clients that use an abstraction by declaring instances of it and then manipulating those instances and clients that subclass an abstraction to specializes or augment its behavior. Designing in favor of first client leads us to hide implementation details and focus upon the responsibilities of the abstraction in the real word. Designing in favor of secured client requires us to expose certain implementation details. In practice, we observe that developers generally start by using the most obvious classes in a library. Developers may discover a pattern in this own tailoring of a predefined class and so add it to the library as a primitive abstraction. Similarly, a team of developers may realize that certain domain specific classes keep showing up across system, these too get introduced into the class library. This is precisely how class libraries grow over time: not overnight, but from similar, stable, intermediate forms.

DESIGN

The architecture of library addresses each of the patterns and then populate the library its implementation.

Tactical Issues: Design can never be language independent. The particular features and semantics of a given language influence architectural decisions. OOP offer three basic facilities for organizing a rich collection of classes' inheritance, aggregation, and parameterization. Inheritance is certainly the most visible (and most popular) aspect of object oriented technology; however, it is not the only structuring principle that we should consider. Parameterization combined with inheritance and aggregation can lead us to a very powerful yet small architecture.

The below figure shows relationships among parameterized class (Queue), its subclass (Priority Queue), one of its instantiations (Priority Event Queue), and one of its instances (mail Queue). This example leads us to assert our first architectural principle for this library: Except for a few cases, the classes we provide should be parameterized. This decision supports the library's requirement for safety.

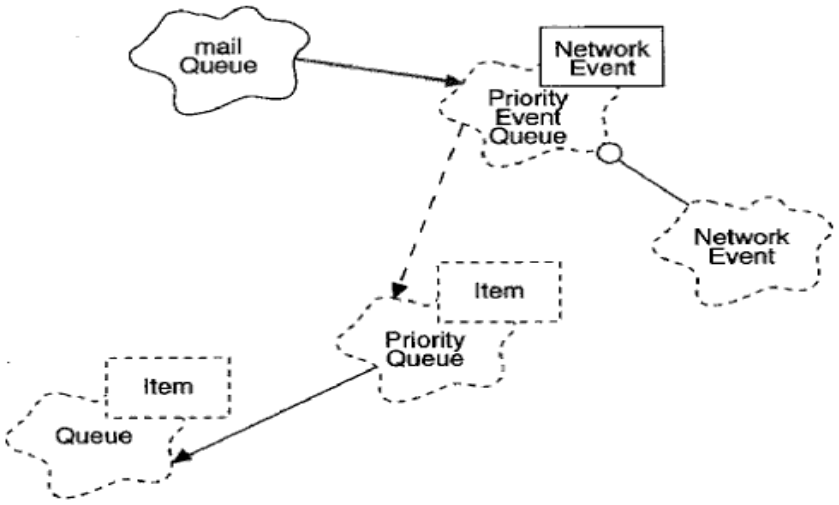


Fig: Inheritance and Parameterization

Macro Organization:
One of worse organization could be to form a flat collection of classes, through which developers would have to navigate to find the classes needed. It can be done for better by placing each cluster of classes into its own category as shown in figure.

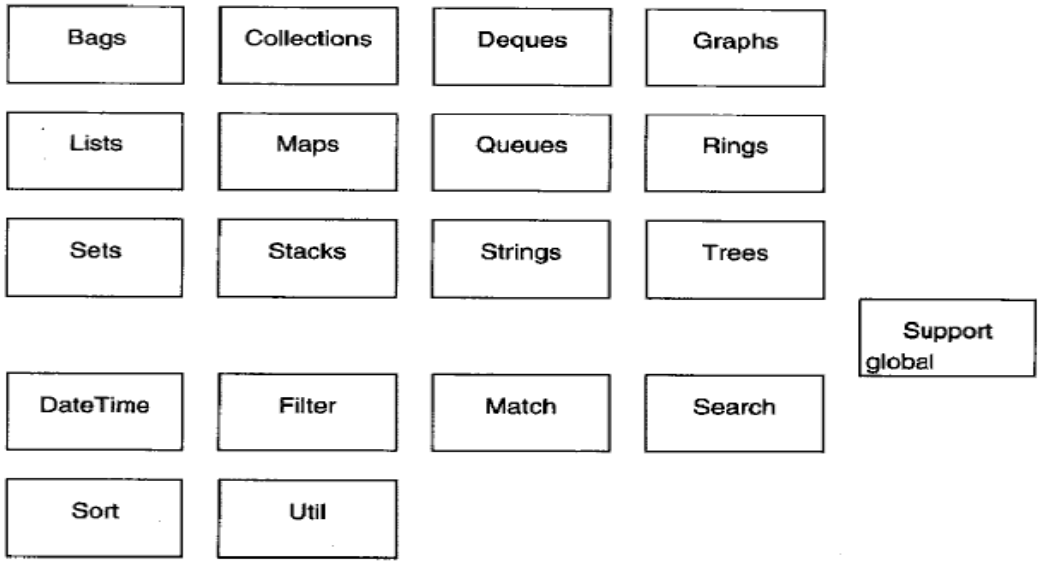


Fig: Foundation Class Library Class Categories

A quick analysis suggests that are is an opportunity for explaining the representations common among the classes in this library. So, there is shown the existence of the globally accessible category named support, whose purpose is to organize such lower level abstractions. This category is also used to collect the classes needed is support of the library's common mechanisms.
Principle for this library may also be possible using policy and implementation. In a sense, abstractions such as queues, sets and rings represent particular policies for using lower level structures such as linked lists or arrays. E.g. a queue defines the policies whereby items can only be added to one end of a structure and removed from the other,. A set, on the other hand, enforces no such policy requiring an ordering of items. A ring does enforce an ordering, but sets the policy that the front and back of its items are connected.

Class Families: A third principle central to the design of this library is the concept of building families of classes, related by lines of inheritance. For each kind of structure, several different kinds of classes are provided, united by a shared interface (such as the abstract base class queue), but with several concrete subclasses, each having a slightly different representation with different time and space semantics developer can select the one concrete class that best fit the needs of a given application. The intentional and clear separation of concerns between an abstract class and its concrete class allows a developer to initially select one concrete class and later, as the application is being turned, replace with a sibling concrete class with minimal effort. The developers can be confident that the application will still work because all sibling concrete classes share the same interface and the same central behavior.

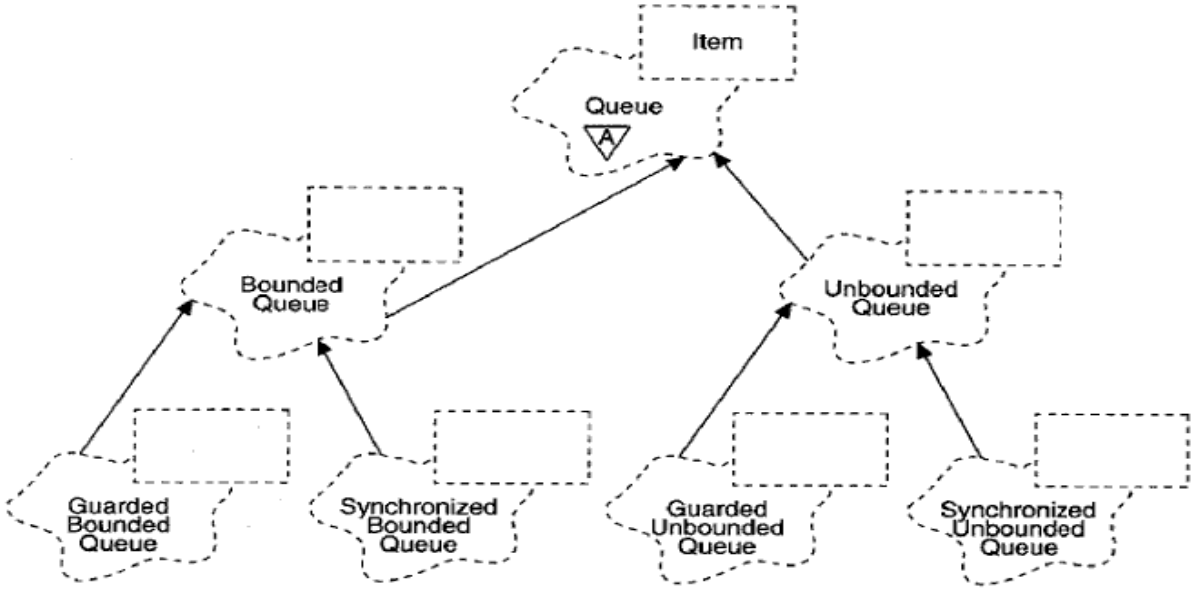


Fig: Class Families

The various concrete members of a family of classes represent the forms of an abstraction. There are two fundamental forms of most abstractions. First is the form of representation which establishes the concrete implementation of an abstract base class. The two forms of an abstraction are:

(i) **Bounded:** The structure is stored on the stack and thus has a static size at the time the object is constructed.

(ii) **Unbounded:** The structure is stored on the leap and thus may grow to the limits of available memory.

The second important variation concerns synchronization many useful applications involve only a single process, called sequential systems as they involve only a single thread of control. Real time applications may require the synchronization of several simulations threads of control within the same system; called concurrent system.

The three designs which require different degrees of cooperation among the gents that interact with a shared object are sequential, guarded and synchronous

Micro Organization:

In support of the library requirement for simplicity, we choose to follow a consistent style for every structure and tool in the library.

Time and space Semantics

The most important thing in the architecture of framework is the mechanism that provides the client with alternative time and space semantics within each family of classes. Unbounded forms are applicable in those cases where the ultimate size of the structure cannot be predicted and where allocating and de-allocating storage from the help is neither too costly nor unsafe (as it may be in certain time critical applications). Bounded forms are better suited to similar structures whose average and maximum sizes are predictable and where help usage is deemed insecure.

The responsibility of the class unbounded is to provide a very efficient linked list structure that uses items allocated from the heap. This representation is time efficient but less space efficient

because for each item, we must also save storage for a pointer to the next item. The responsibility of the class bounded is to provide a very efficient optimally packed array-array based class. This representation is space efficient, but less time efficient because when adding new items in the middle of the container, items at one end must be moved down by copying.

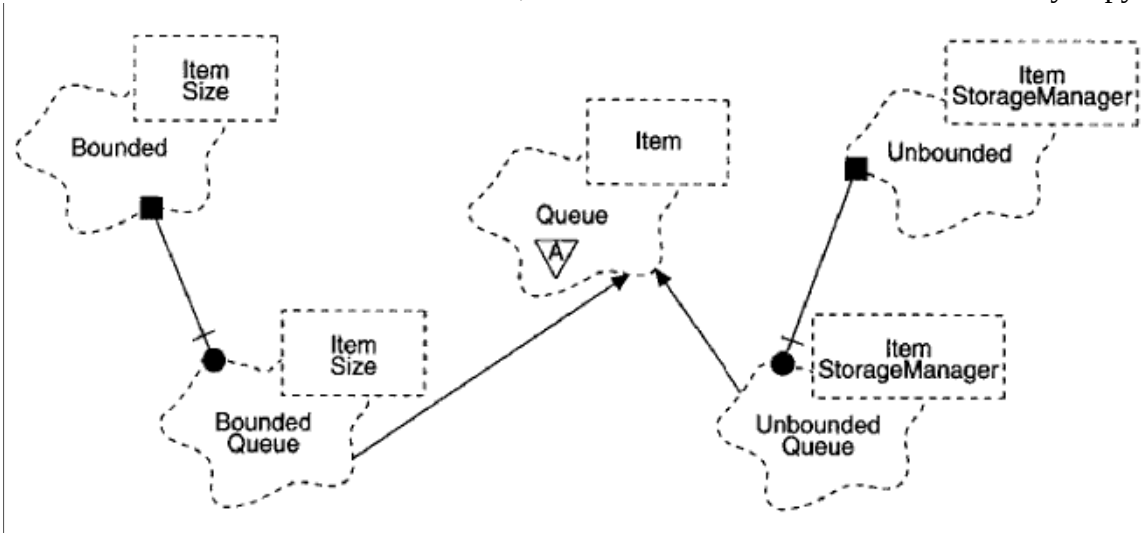


Fig: Bounded and Unbounded Forms

As shown in figure, we use aggregation to place these lower-level classes in our family of classes. Specifically, the diagram shows that we use physical containment by value with protected access, meaning that this lower-level representation is accessible only to subclasses and friends.

Storage Management: Storage management is an issue for all unbounded forms, because the library designer must consider specific policies for allocating and de-allocating nodes from the heap. A naïve approach will simply use the global **new** and **delete** functions, but this strategy will often exhibit very poor runtime performance.

The members functions allocate and de-allocate are also used for storage management which dispense and release memory respectively.

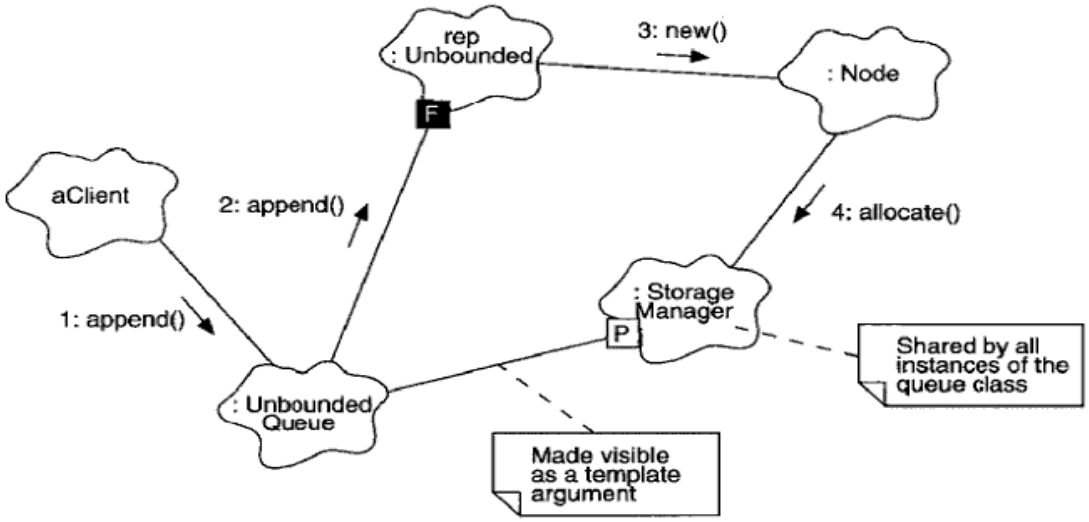


Fig: Storage Management Mechanism

Figure illustrates the mechanism we have chosen to provide storage management. Let's walk through the scenario described in this object diagram:

- A client invokes the operation append upon an instance of UnboundedQueue (more precisely, upon an instance of an instantiation of UnboundedQueue).
- The UnboundedQueue object in turn delegates responsibility for this operation to its member object rep, an instance of the class Unbounded.
- Unbounded allocates a new instance of Node by invoking its static member function new.
- The Node instance in turn delegates the responsibility of allocation to its storage manager, which is made visible to the class UnboundedQueue (and in turn to the classes Unbounded and Node) as a template argument. This storage manager is shared by all instances of the class, and so serves to provide a consistent class-wide storage management policy.

We implement the protocol for the class Unmanaged by inline calls to the global operators new and delete. We call this policy unmanaged, because it effectively does nothing beyond the default policy provided by the language. A far better policy is called managed. Under this policy, nodes are allocated and de-allocated from a common pool of memory. Unused nodes of any kind are returned to a free list, and allocation takes nodes from this free list unless it is empty, in which case another chunk of free memory is allocated from the heap.

Figure below provides a class diagram illustrating the various classes that collaborate to provide a managed storage policy. We show only an association between the classes Managed and its clients Unbounded and UnboundedQueue, because this association will only be manifested in a specific instantiation of the classes.

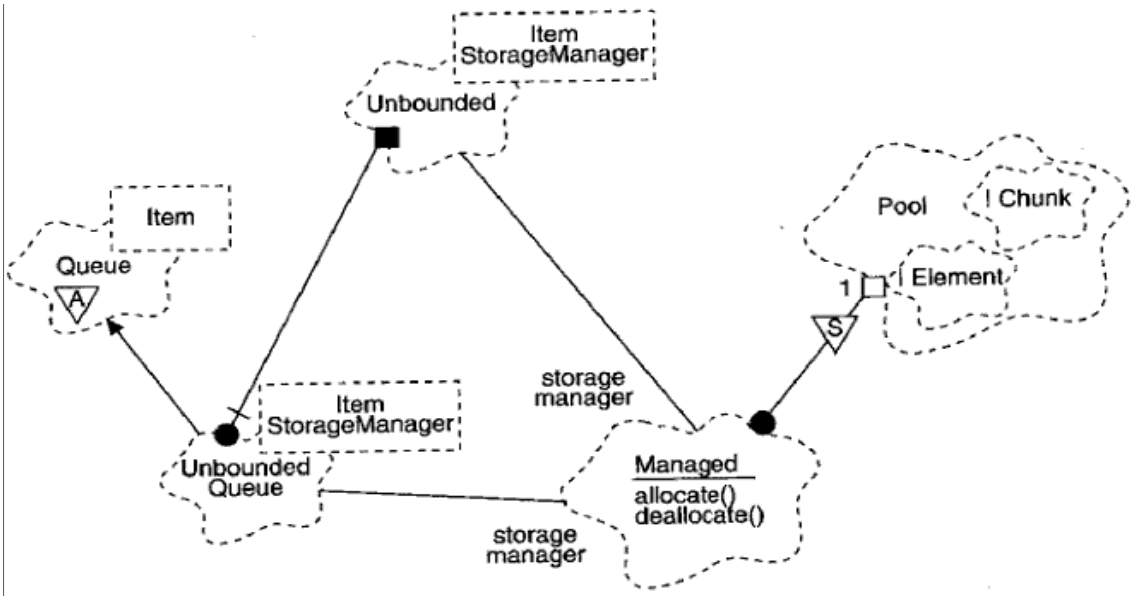


Fig: Storage Management Classes

Exceptional Conditions: Some mechanisms, for reporting any dynamic violations such as trying to add an item to an already full bounded queue or removing item from an empty queue, must be available for exceptional conditions. The architecture uses a hierarchy of exception classes and separates them from the mechanisms involved in reporting them. a base exception classes can be declared as follows:
More exceptional conditions can be declared as sub classes to the base classes as Error, Duplicate, Illegal pattern, Isnull, Lexical Error, Math error, Not found, Not Null, Overflow, Range Error, Storage Error, Under flow.

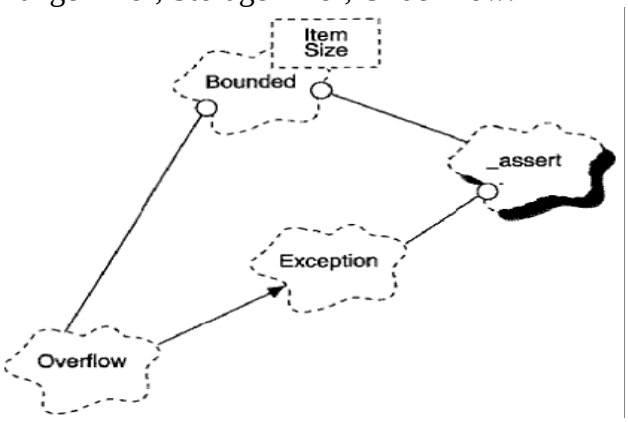


Fig: Exception Classes
One very important aspect of our use of exceptions is that they are guaranteed not to corrupt the state of any object that throws an exception, except in the case of out-of-memory conditions (in which case all bets are usually off, anyway).

Iteration: An iterator is an operation that permits all parts of an object to be accessed in some well defined order. We can define iteration as part of an object's protocol or we can invent separate objects that act as agents responsible for iterating across a structure.
When introducing iterators, we have two design choices: we can define iteration as part of an object's protocol, or we can invent separate objects that act as agents responsible for iterating across a structure. We choose the second alternative for two compelling reasons:

- By providing separate iterator classes, we make it possible to have several iterator objects working upon the same object.
- Iteration slightly breaks the encapsulation of an object's state; by separating the behavior of iteration from the rest of an abstraction's protocol, we provide a much clearer separation of concerns.

For each structure, we provide two forms of iteration. Specifically, an active iterator requires that clients explicitly advance the iterator; in one logical expression, a passive iterator applies a client-supplied function, and so requires less collaboration on the part of the client.

Synchronization: Classes like semaphore and monitors can be declared for concurrency in the design of library. Subclasses like a writer (an agent that alters the state of an object) or a reader (an agent that operates upon an object) can be derived from semaphore and monitor base classes.

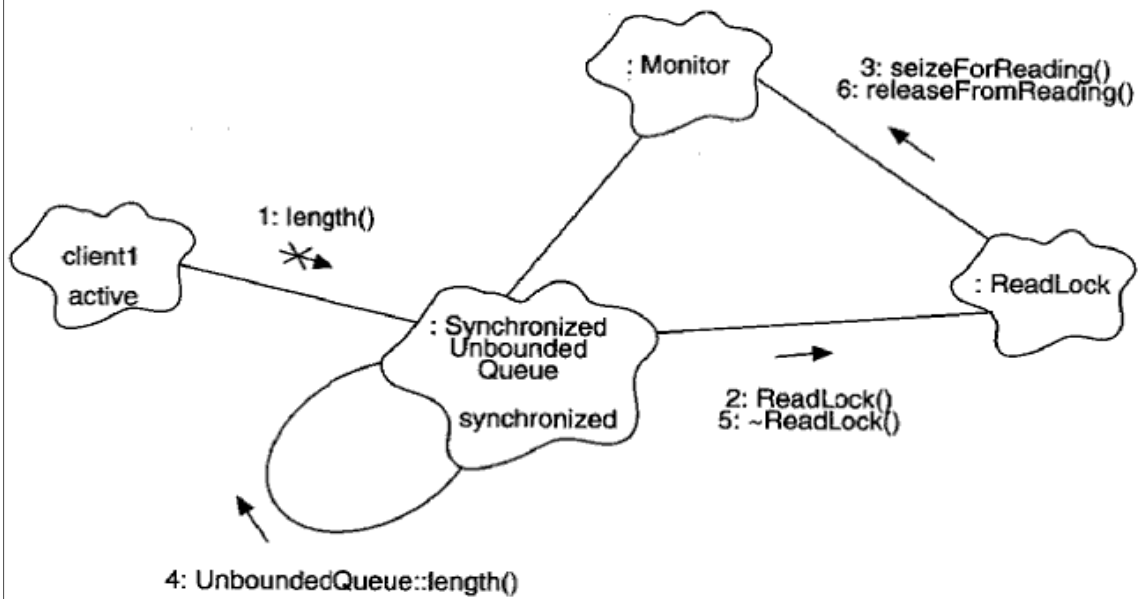


Fig: Synchronized Process Mechanism

There exist two basic kinds of process synchronization via monitors of this sort:

- Single: Guarantees the semantics of a structure in the presence of multiple threads of control, with a single reader or writer.
- Multiple: Guarantees the semantics of a structure in the presence of multiple threads of control, with multiple simultaneous readers or a single writer.

A writer is an agent that alters the state of an object; writers are those agents that invoke modifier member functions. A reader is an agent that operates upon an object, yet preserves its state; readers are those objects that only invoke selector functions. The multiple form therefore provides the greatest amount of real parallelism. We can implement these two policies as subclasses of the abstract base class Monitor. Both the single and multiple forms may be built upon the Semaphore class.

EVOLUTION

Class Interface Design: Once we have selected the patterns that make up its architecture, three or four family of classes (such as queue, set, tree) are taken, implemented term against this architecture and then tested them against real client applications. Activity is decided upon a suitable interface for each base class. This involves isolated class design.

Support classes: Bounded and unbounded is insufficient for range of time and space semantics. So, third form of representation, dynamic class can be used as support classes. This structure is stored on the leap as an array where length may shrink or grow.

Tools: In this library, the primary use of templates is to parameterize each structure with the kind of item it contains; this is why such structures are often called container classes. For example,

consider the algorithms that search for patterns within a sequence. A number of such algorithms exist, with varying time semantics:

- Simple: The structure is searched sequentially for the given pattern; in the worst case, this algorithm has a time complexity on the order of *O(pn)*, where *p* is the length of the pattern, and *n* is the length of the sequence.
- Knuth-Morris-Pratt: The structure is searched for the given pattern, with a time complexity of *O(p + n)*; searching requires no backup, which makes this algorithm suitable for streams.
- Boyer-Moore: The structure is searched for the given pattern, with a sublinear time complexity of *O(c * (p + n))*, where $c < 1$ and is inversely proportional to *p*.
- Regular expression: The structure is searched for the given regular expression pattern.

MAINTENANCE

If the framework is well-engineered, they tend to reach a sort of a critical mass of functionality and adaptability. As we discover patterns in the ways that clients use our framework, then it makes sense to codify these patterns by formally making then a part of the library. A sign of a well designed framework is that we can introduce. A sign of a well-designed framework is that we can introduce these new patterns during maintenance by reusing existing mechanisms and thus preserving its design integrity. We may design classes in library that supplied persistence semantics.

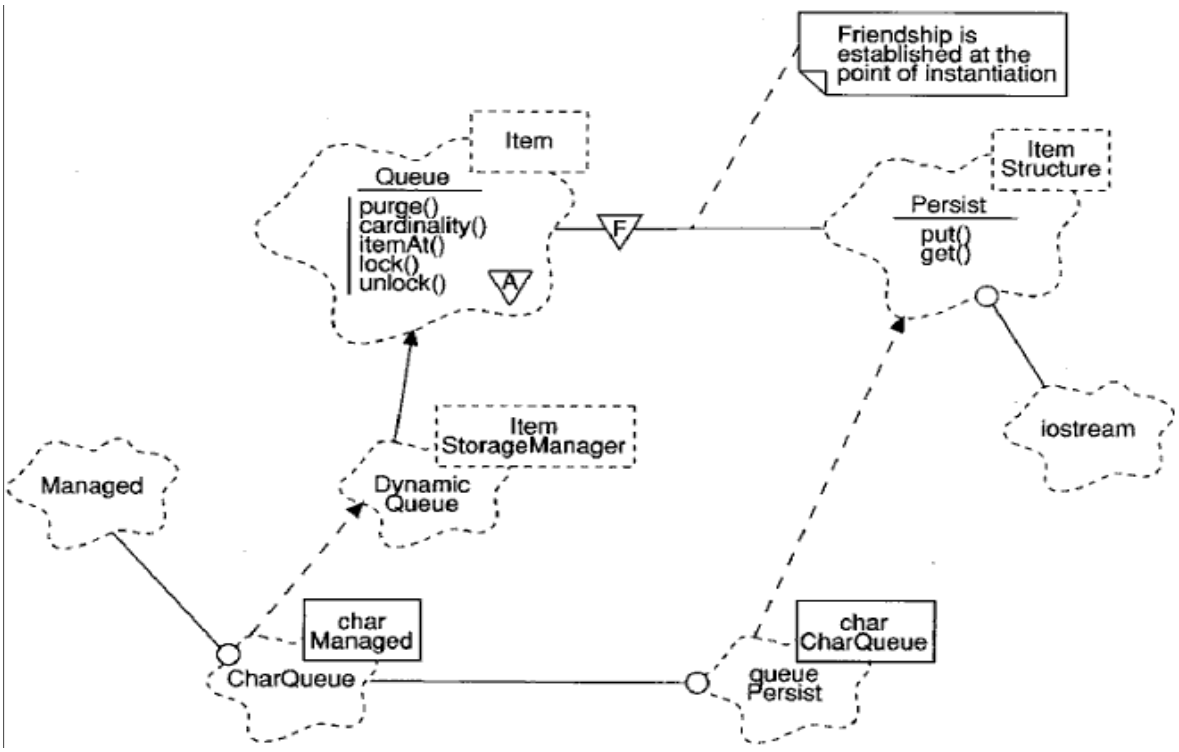


Fig: Persistence Classes

Chapter – 7
Applications
CLIENT / SERVER COMPUTING
INVENTORY TRACKING

In the past, traditional mainframe computing raised some very real walls around a company's database assets. However, given the advent of low-cost computing, which places personal productivity tools in the hands of a multitude of workers, together with networks that serve to link the ubiquitous personal computer across offices as well as across nations, the face of information management systems has been irreversibly changed. Clearly a major part of this fundamental change is the application of client/server architectures. As points out, "The rapid movement toward downsizing and client-server computing is being driven by business imperatives.

ANALYSIS

Defining the boundaries of the problems: The key function of inventory tracking system includes:

- Order entry: Responsible of taking customer orders and for responding to customer queries about the status of an order.
- Accounting: Responsible for sending invoices and tracking customer payments (accounts receivable) as well as for paying suppliers for orders from purchasing (account payable)
- Snipping: Responsible for assembling package for shipment in support of filling customer orders.
- Stocking: Responsible for placing new inventory in stock as well as for retrieving inventory in support of filling customer orders.
- Purchasing: Responsible for ordering stock from suppliers and tracking supplier shipments.
- Receiving: Responsible for accepting stock from suppliers.
- Planning: Responsible for generating reports to management and studying trends in inventory levels and customer activity.

We must design the inventory system expecting it will change over time. The two elements are most likely to change over the lifetime of this system namely the kinds of data to be stored and the hardware upon which the application executes.

Inventory-Tracking System Requirements

As part of its expansion into several new and specialized markets, a mail-order catalog company has decided to establish a number of relatively autonomous regional warehouses. Each such warehouse retains local responsibility for inventory management and order processing. To target niche markets efficiently, each warehouse is tasked with maintaining inventory that is best suited to the local market. The specific product line that each warehouse manages may differ from region to region; furthermore, the product line managed by any one region tends to be updated almost yearly to keep up with changing consumer tastes. For reasons of economies of scale, the parent company desires to have a common inventory- and order-tracking system across all its warehouses. The key functions of this system include:

- Tracking inventory as it enters the warehouse, shipped from a variety of suppliers.
- Tracking orders as they are received from a central but remote telemarketing organization; orders may also be received by mail, and are processed locally.
- Generating packing slips, used to direct warehouse personnel in assembling and then shipping an order.
- Generating invoices and tracking accounts receivable.
- Generating supply requests and tracking accounts payable.

In addition to automating much of the warehouse's daily workflow, the system must provide a general and open-ended reporting facility, so that the management team can track sales trends, identify valued and problem customers and suppliers, and carry out special promotional programs.

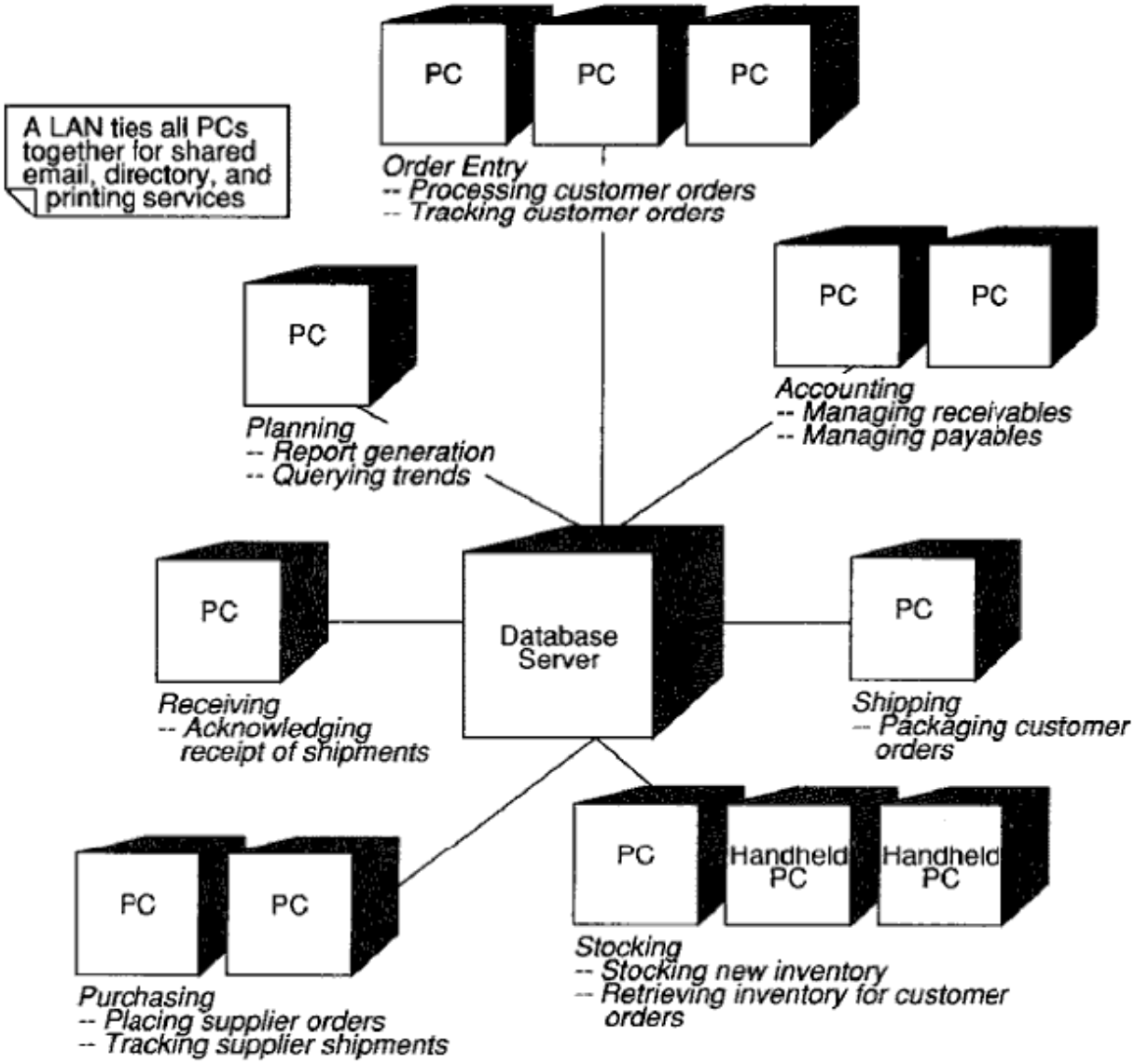


Fig: Inventory-Tracking System Network

Client Server Computing: It is a decentralized architecture that enables end users to gain access to information transparently within a multi-vendor environment. Client server applications couple a GUI to a server based RDBMS. Client server application can typically be divided into one of four components.

- **Presentation logic:** The part of an application that interacts with an end user device such as a terminal, a bar code reader or a handheld computer
- **Business logic.** Uses information from the user and from the database to carry out transactions as constrained by the rules of the business.
- **Database logic:** Manipulates data within the application using SQL
- **Database Processing:** Actual processing of database data by DBMS

Not only do standards impact the architect's decisions, but issues such as security, performance, and capacity must be weighed as well. Berson goes en to suggest some rules of thumb for the client/server architect:

- In general, a presentation logic component with its screen input-output facilities is placed on a client system.
- Given the available power of the client workstations, and the fact that the presentation logic resides en the client system, it makes sense to also place some part of the business logic en a client system.
- If the database processing logic is embedded into the business logic, and if clients maintain some low-interaction, quasi-static data, then the database processing logic can be placed on the client system.
- Given the fact that a typical LAN connects clients within a common purpose workgroup, and assuming that the workgroup shares a database, all common, shared fragments of the business and database processing logic and DBMS itself should be placed on the server.

If we make the right architectural decisions and succeed in carrying out the tactical details of its implementation, the client/server model offers a number of benefits:

- It allows corporations to leverage emerging desktop computing technology better.
- It allows the processing to reside close to the source of data being processed. Therefore, network traffic (and response time) can be greatly reduced.
- It facilitates the use of graphical user interfaces available on powerful workstations.
- It allows for and encourages the acceptance of open systems.

Of course, there are risks:

- If a significant portion of application logic is moved to a server, the server may become a bottleneck in the same fashion as a mainframe in master-slave architecture.
- Distributed applications are more complex than non-distributed applications.

Scenarios:

We begin by enumerating a number of primary use cases, as viewed from the various functional elements of the system:

- A customer phones the remote telemarketing organization to place an order.
- A customer mails in an order.
- A customer calls to find out about the status of an order.
- A customer calls to add items to or remove items from an existing order.

- A stockperson receives a packing order to retrieve stock for a customer order.
- Shipping receives an assembled order and prepares it for mailing.
- Accounting prepares a customer invoice.
- Purchasing places an order for new inventory.
- Purchasing adds or removes a new supplier.
- Purchasing queries the status of an existing supplier order.
- Receiving accepts a shipment from a supplier, placed against a standing purchase order.
- A stockperson places new stock into inventory.
- Accounting cuts a check against a purchase order for new inventory.
- The planning department generates a trend report, showing the sales activity for various products.
- For tax-reporting purposes, the planning department generates a summary showing current inventory levels.

For each of these primary scenarios, we can envision a number of secondary ones:

- An item a customer requested is out of stock or on backorder.
- A customer's order is incomplete, or mentions incorrect or obsolete product numbers.
- A customer calls to query about or change an order, but can't remember what exactly was ordered, by whom, or when.
- A stockperson receives a packing order to retrieve stock, but the item cannot be found. Shipping receives an incompletely assembled order.
- A customer fails to pay an invoice.
- Purchasing places an order for new inventory, but the supplier has gone out of business or no longer carries the item.
- Receiving accepts an incomplete shipment from a supplier.
- Receiving accepts a shipment from a supplier for which no purchase order can be found.
- A stockperson places new stock into inventory, only to discover that there is no space for the item.
- Business tax code changes, requiring the planning department to generate a number of new inventory reports.

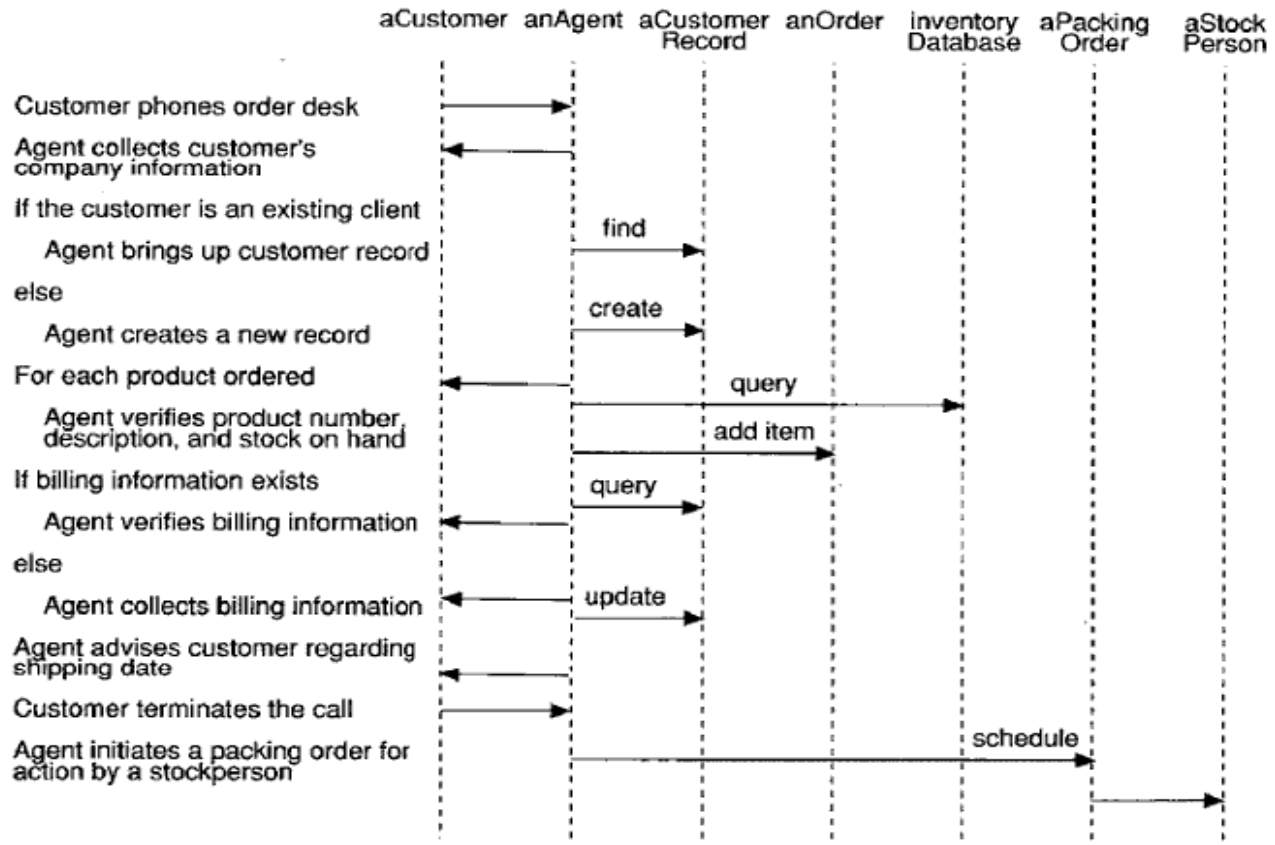


Fig: Order Scenario

Figure above provides a primary use case for a customer placing an order with the remote telemarketing organization. Here we see that a number of different objects collaborate to carry out this system function. Although control centers around the customer/agent interaction, three other key objects (namely, aCustomerRecord, the inventory Database, and aPackingorder, all of which are artifacts of the inventory tracking system) play a pivotal role.

Figure below continues this scenario with an elaboration upon the packing order/stockperson interaction, another critical system behavior. Here we see that the stockperson is at the center of this scenario's activity, and collaborates with other objects, namely, shipping, which did not play a role in the previous scenario.

First, we list the various people that interact with the system such as Customer, Supplier, OrderAgent, Accountant, ShippingAgent, Stoffierson, PurchasingAgent, ReceivingAgent, Planner.

Our analysis also reveals the following key abstractions, each of which represents some information manipulated by the system such as CustomerRecord, ProductRecord, SupplierRecord, Order, PurchaseOrder, Invoice, PackingOrder, StockingOrder, ShippingLabel.

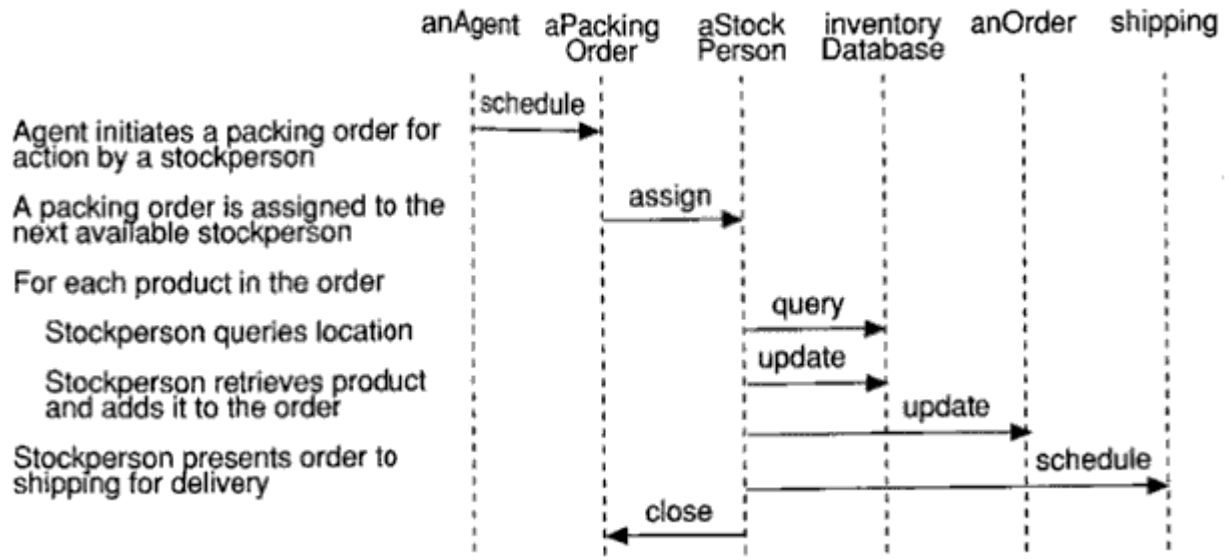


Fig: Packing Order Scenario

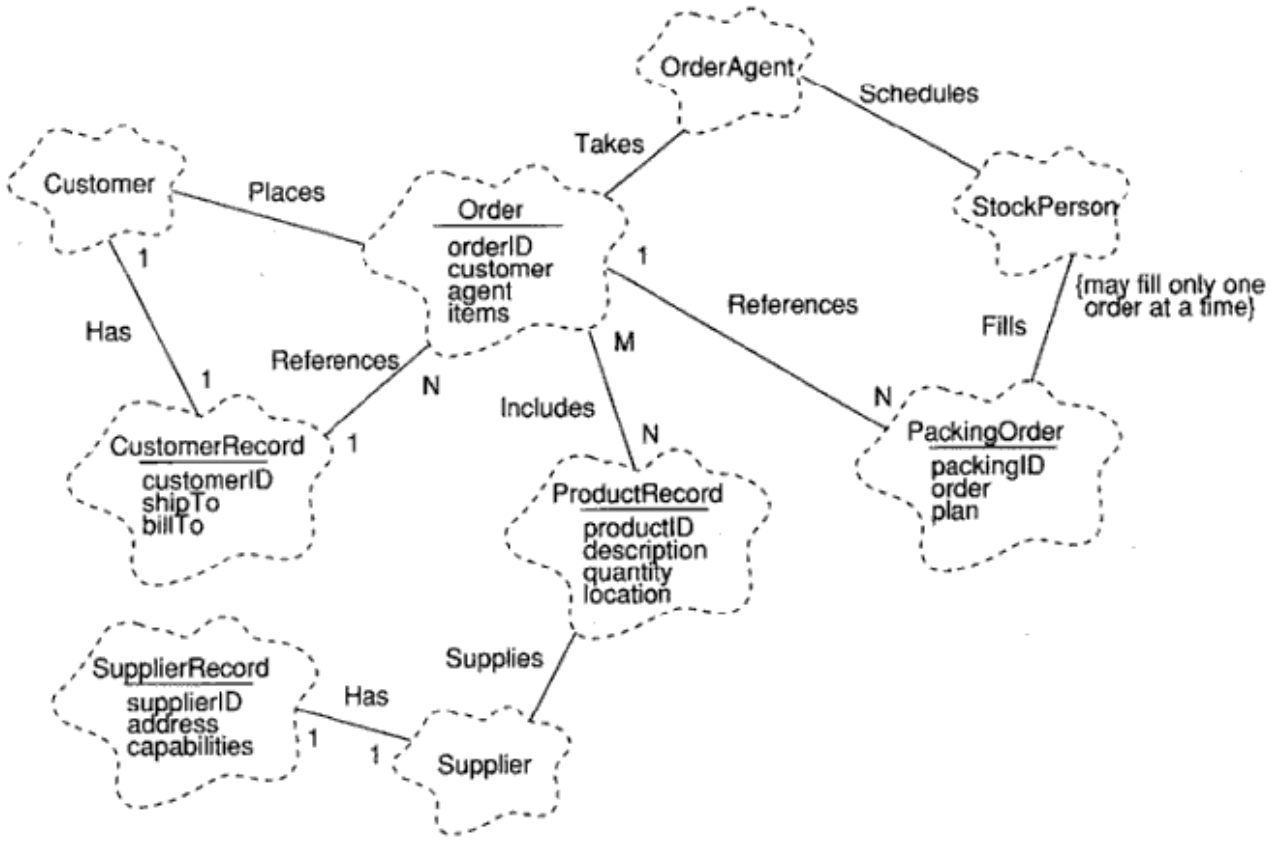


Fig: Key Classes for Taking and Filling Orders

To complete this phase of our analysis, we introduce two final key- classes:

- **Report**
- **Transaction**

Database Models: Different database models are hierarchical, Network and relational database models. Recently, a fourth kind of database model has emerged, namely, object-oriented databases (OODBMS). An OODBMS represents a merging of traditional database technology and the object model.

Normalization is a property of a table; we say that a particular table is in normal form if it satisfies certain properties. There are several levels of normal forms, each of which builds upon the other:

- First normal form (1NF): Each attribute represents an atomic value (non decomposable attributes).
- Second normal form (2NF) Table is in 1NF, and each attribute depends entirely upon the key (functionally independent attributes).
- Third normal form (3NF) Table is in 2NF, and no attribute represents a fact about another attribute (mutually independent attributes).

SQL: Especially given our object-oriented view of the world, wherein we unite the data and behavioral aspects of our abstractions, a user might wish to perform a variety of common transactions upon the tables. For example, we might want to add new suppliers, delete products, or update quantities in the inventory. We also might want to query these tables in a variety of ways. For instance, we might want a report that lists all the products that we can order from a particular supplier. We might also want a report listing the products whose inventory is either too low or too high, according to some criteria we give it. Finally, we might want a comprehensive report giving us the cost to restock the inventory to certain levels, using the most inexpensive sources of products. These kinds of transactions are common to almost every application of an RDBMS, and so a standard language called SQL (Structured Query Language) has emerged for interacting with relational databases. SQL may be used either interactively or programmatically.

Schema Analysis: In data intensive applications such as the inventory-tracking system, we start with an object oriented analysis and use its process to drive our design of the database, rather then the reverse of focusing upon the database schema first and deriving an object-oriented architecture from that.

DESIGN

In formulating the architecture of the inventory-tracking system, we must address three organizational elements: the split in client/server functionality, a mechanism for controlling transactions, and a strategy for building client applications.

Client Server Architecture: In formulating the architecture of the inventory-tracking system, we must address three organizational elements: the split in client/server functionality, a mechanism for controlling transactions, and a strategy for building client applications. First principle gives us the answer as to how: focus upon the behavior of each abstraction first, as derived from a use-case analysis of each entity, and only then decide where to allocate each abstraction's behavior. Once we have done this for a few interesting objects, some patterns of behavior will emerge, and we can, then codify these patterns to guide us in allocating functionality for all the remaining abstractions.

For example, let's consider the behavior of the two classes Order and ProductRecord. Further analysis of the first class, together with some isolated class design, suggests that the following operations are applicable:

Construct, setCustomer, setorderAgent, addItem, removeItefn, ordered, customer, orderAgent, numberOfItems, itemAt, quantityOf, totalvalue.

The below object diagram illustrates this SQL mechanism, with a scenario showing a client setting the customer for an order. Our scenario follows this order of events:

- aClient invokes setCustomer upon some instance of Order; an instance of the class Customer is provided as a parameter to this method function.
- The order object invokes the selector customerID upon the customer parameter, to retrieve its corresponding primary key.
- The order object dispatches an SQL UPDATE statement to set the customer id in the order database.

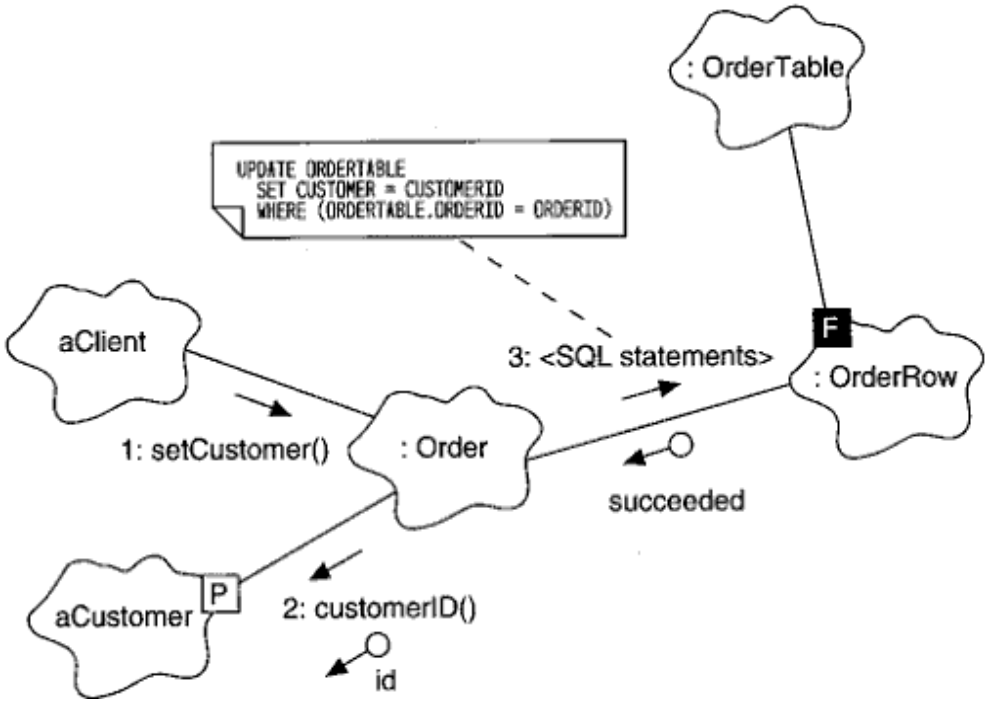


Fig: SQL Mechanism

Continuing our example, let's turn to our abstraction of a different class, namely Product. Further analysis, together with some isolated class design, suggests that the following operations are applicable: Construct, setDescription, setQuantity, setLocation, setSupplier, productID, description, quantity, location, supplier.

These operations are common to every kind of product. However, use-case analysis reveals that these semantics are not sufficient for certain kinds of products. For example, given the open-ended nature of our inventory-tracking system together with the fact that product lines may change, our application may have to deal with the following kinds of products, which have their own unique behavior:

- Plants and food products that are perishable and therefore require special handling and shipping.
- Chemical products that also require special handling because they are caustic or toxic.
- Distinct products, such as radio transmitters and receivers, that should be shipped in matched sets and are therefore dependent upon one another.
- High-tech components, whose shipping is constrained by the local country's export laws.

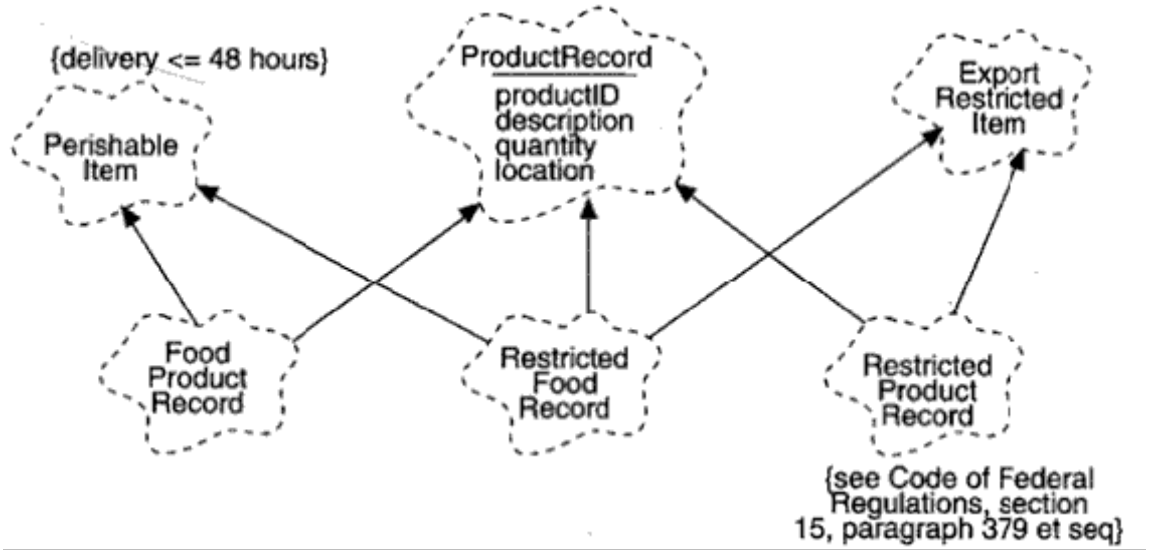


Fig: Product Classes

Transaction Mechanism: Client/server computing implies collaboration between client and server, and so we must have some common mechanism whereby disparate parts of the system communicate with one another. There are three basic types of cooperative processing communications techniques that client/server architecture can use.

- Pipes
- Remote procedure calls
- Client/server SQL interactions

From the outside, we observe that the following operations capture our core abstraction of a transaction's behavior: attachOperation, dispatch, Commit, rollback, status.

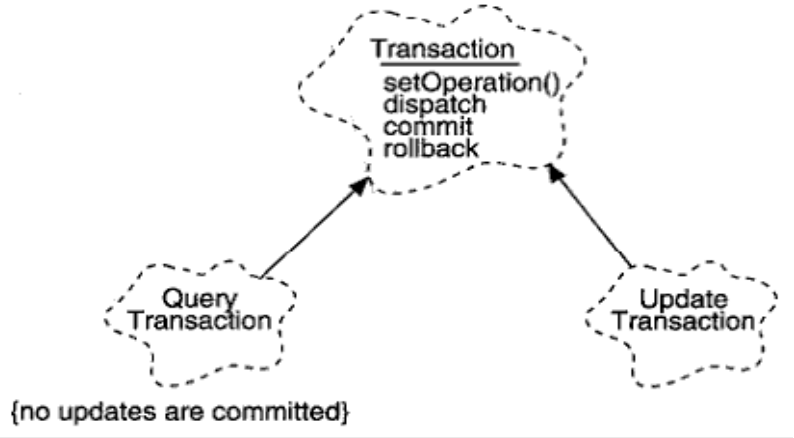


Fig: Transaction Classes

Above figure shows a class diagram that illustrates our abstraction of transactions. Here we find a hierarchy of transactions. The base class Transaction above captures the structure and behavior common to all transactions, whereas the subclasses carry out the semantics of certain specialized transactions. We distinguish between UpdateTransaction and Query Action, for example, because each provides very disjoint semantics: the former class modifies the state of the server, whereas the second class does not.

Building Client Applications: To a large degree, building a client application is a problem of building a GUI intensive program. Building an intuitive and friendly user interface is, however, as much an art as it is a science. In client/server applications such as this one, it is often the look and feel of the user interface that makes the difference between a wildly popular system and one that is quickly discarded.
The truly hard part of living with any large, integrated class library especially those for user interfaces is learning what mechanisms it embodies. Perhaps the most important mechanisms we must understand, at least in the context of cooperative client/server computing, are how a GUI application responds to events. GUI clients have to contend with the following kinds of events:

- Mouse events
- Keyboard events
- Menu events
- Window update events
- Resizing events
- Activation/deactivation events
- Initialize/terminate events

There are several architectural models for dealing with events:

- Event-loop model: The event loop checks for pending events and dispatches an appropriate event-handling routine.
- Event-callback model: The application registers a callback function for each GUI widget that knows how to respond to a certain event; the callback is invoked when the widget detects an event.
- Hybrid model: A combination of the event loop and event callback models.

EVOLUTION

Release Management: Now that we have established an architectural frameworks for the inventory tracking system, we can proceed with the system's incremental development. We start this process by first selecting a small number of interesting transactions, taking a vertical slice through our architecture, and then implementing enough of the system to produce an executable product that at least simulates the execution of these transactions.

We might select just three simple transactions: adding a customer, adding a product, and taking an order. Together, the implementation of these three transactions requires us to touch almost every- critical architectural interface, thereby forcing us to validate our strategic assumptions. Once we successfully pass this milestone, we might then generate a stream of new releases, according to the following sequence:

- Modify or delete a customer; modify or delete a product; modify an order; query a customer, order, and product.
- Integrate all similar supplier transactions, create a stocking order, and create an invoice.
- Integrate remaining stocking transactions, create a report, and create a shipment.
- Integrate remaining accounting transactions, create a receiving order.
- Integrate remaining shipping transactions.
- Complete remaining planning transactions.

Application Generators:

Domains such as the inventory-tracking system often include many different kinds of screen templates and hard copy reports that must be generated. For large systems, these parts of the application are not technically difficult to write, just horribly tedious. This is precisely why application generators (or 4GLs, for fourth-generation languages) are so popular for business enterprises. The use of 4GLs is not inconsistent with an object-oriented architecture. Indeed, the controlled use of 4GLs can eliminate writing a considerable amount of code. We use 4GLS to automatically create screens and reports. Given the specification of a screen or report layout, a 4GI, can generate the code necessary to produce the actual screen or report.

MAINTENANCE

For the inventory-tracking system, we can envision several enhancements that changing business conditions may require us to address:

- Allow customers to electronically post their own orders and query the state of pending orders.
- Automatically generate personalized catalogs from our inventory database, tailored to target specific customer groups, or even individual customers.
- Completely automate all warehouse functions, thereby eliminating the human stockperson, as well as most receiving and shipping personnel.