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

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.

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:

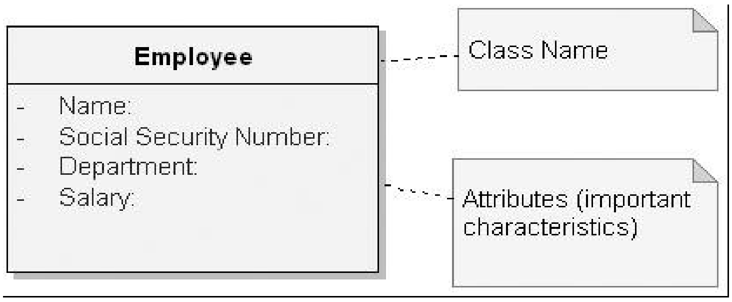
```
struct   personnelRecord{
char     name[100];
int      socialsecurityNumber;
char     department[10];
float    salary;
};
```

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.

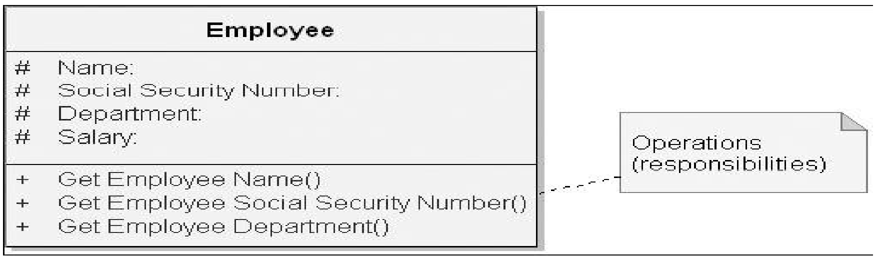
```
Class personnelrecord{
public: char*employeename()const;
 int      SSN() const;
```

```
 char*  empdept      const;
 protected:
char    name[100];
int     SSN;
 char   department[10];
 float  salary;
};
```



Employee Class with Attributes

Employee Objects Tom and Kaitlyn



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.

<u>Behavior:</u>

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{ p
ublic:
Queue(constQuue); 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.

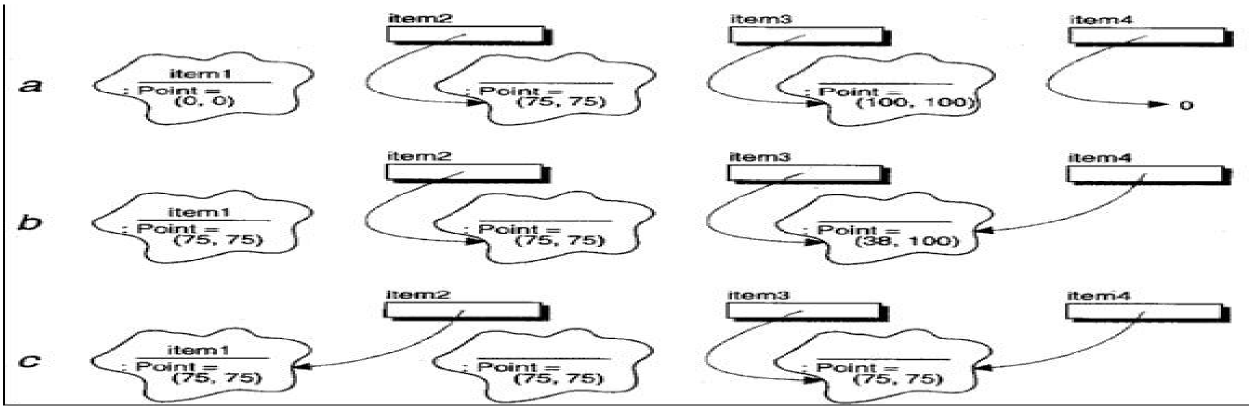
Identity: Identity is that property of an object which distinguishes it from all other objects.

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



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.

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.

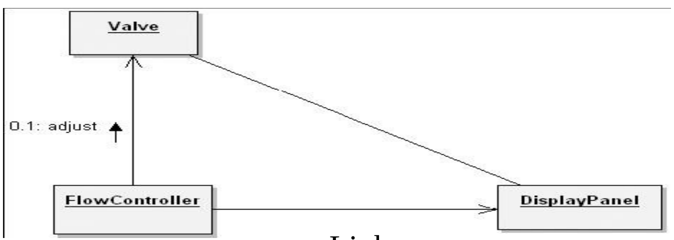
Two kinds of objects relationships are links and aggregation.

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 applicati



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.

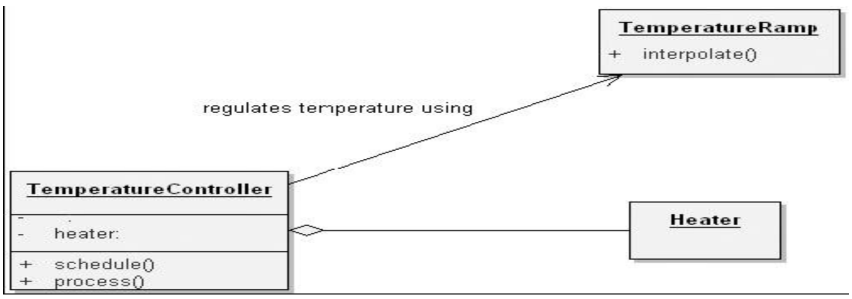
1. **Sequential:** The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
2. **Guarded:** The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual 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.



Aggregation

The Nature of the class:

A class is a set of objects that share a common structure, common behavior and common semantics.

A single object is simply an instance of a class. Object is a concrete entity that exists in time and space but class represents only an abstraction. A class may be an object is not a class.

Interface and Implementation: The interface of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and secrets of its behavior.

The interface primarily consists of the declarations of all the operators applicable to instance of this class, but it may also include the declaration of other classes, constants variables and exceptions as needed to complete the abstraction.

The implementation of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class consists of the class.

Interface of the class is divided into following four parts.

- **Public**: a declaration that is accessible to all clients
- **Protected**: a declaration that is accessible only to the class itself and its subclasses

- **Private:** a declaration that is accessible only to the class itself
- **Package:** a declaration that is accessible only by classes in the same package

Relationship among Classes:

We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some kind of sharing. Second, a class relationship might indicate some kind of semantic connection.
There are 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.
we may show a simple association between these two classes: the class Wheel and the class Vehicle.

Association

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

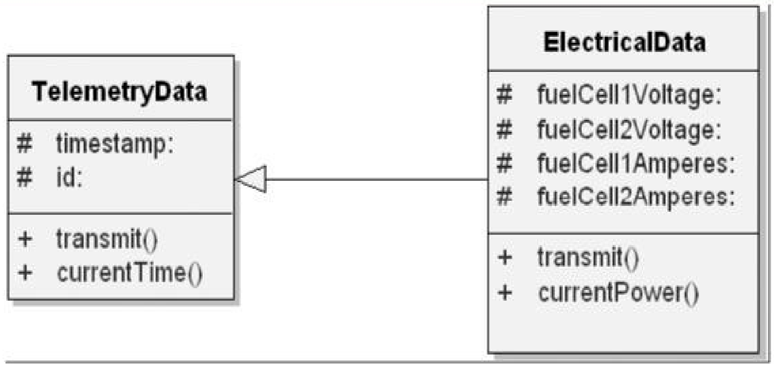
1. One-to-one
2. One-to-many
3. Many-to-many

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.

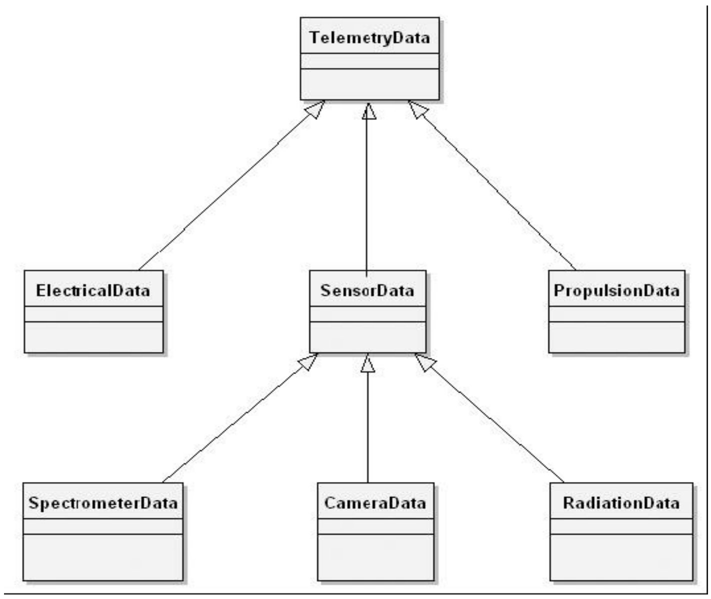


ElectricalData Inherits from the Superclass TelemetryData

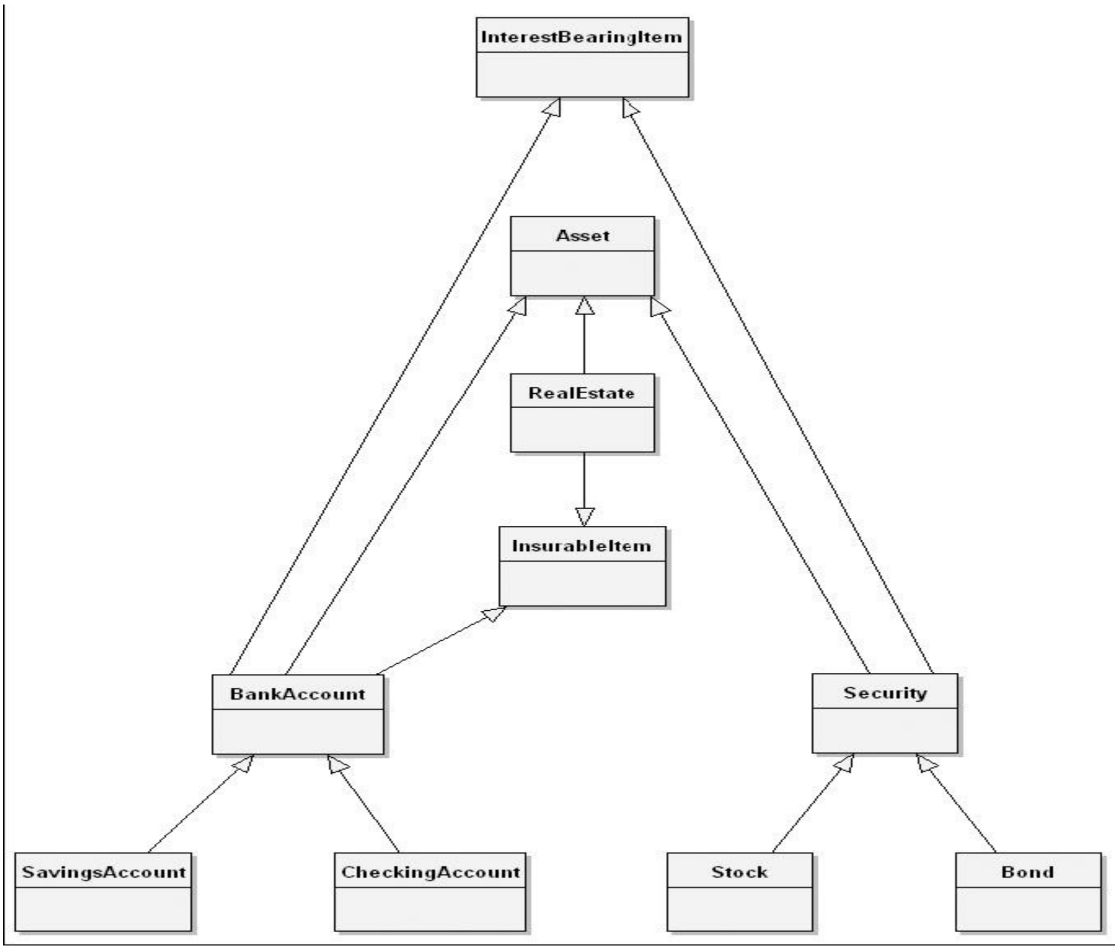
As for the class ElectricalData, this class inherits the structure and behavior of the class TelemetryData but adds to its structure (the additional voltage data), redefines its behavior (the function transmit) to transmit the additional data, and can even add to its behavior (the function currentPower,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

Single Inheritence:

the 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:



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.

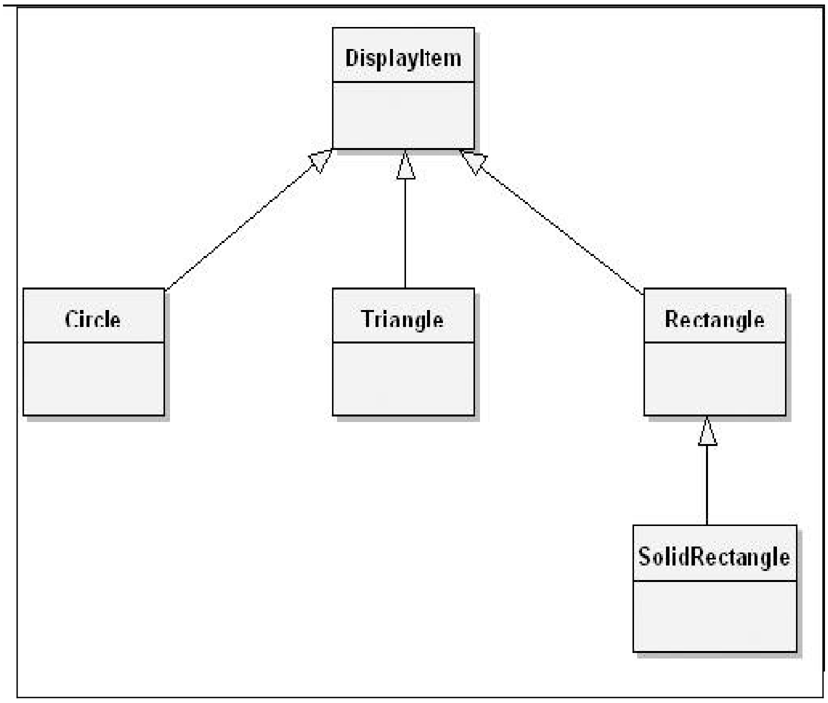
Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance.

The diagram 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.



Polymorphism

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

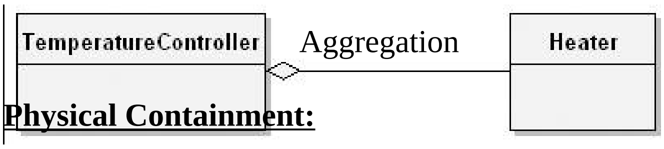
- draw: Draw the item.
- move: Move the item.
- location: Return the location of the item.

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 the class Temperature Controller denotes the whole, and the class Heater is one of its parts.



Physical Containment:

In the case of the class TemperatureController, we have aggregation as containment by value, a kind of physical containment meaning that the Heater object does not exist independently of its enclosing TemperatureController instance.

Rather, the lifetimes of these two objects are intimately connected: 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:

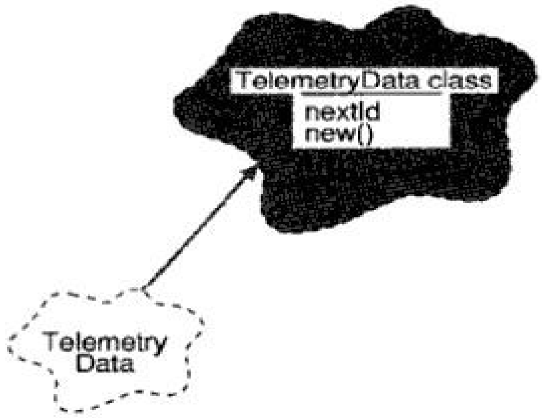
The possibility for a language to provided parameterized modules or types. E.g. List (of: Integer) or List (of: People). There are four basic ways of genericity

- 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.



Metaclass

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)
(vii)

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 sub views, the window next tells each if its sub views to draw them. Each sub view in turn tells the model to draw itself ultimately resulting in an image shown to the user.

The Interplay of Classes and Objects:

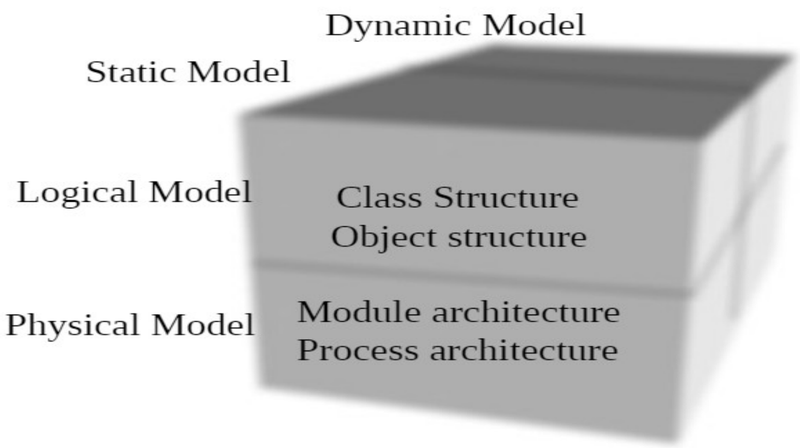
Relationships Between Classes and Objects:

•Classes and objects are separate yet intimately related concepts.
•Classes are static- Their existence, semantics and relationships are fixed prior to execution of a program.
•Class of most objects is static.
•Objects are typically created and destroyed at a rapid rate during lifetime of an application.
Eg. Air Traffic Control system Planes, flight plans, runways and airspaces.

Role of Classes and Objects in Analysis and DesignTwo primary tasks:

1. Identify the classes and objects that form the vocabulary of the problem domain.
2. Invent the structures whereby sets of objects work together to provide behaviors that satisfy the requirements of the problem.

•Focus must be outside view of both(Class structure and object structure)
•In later stages focus is on inside view of both. (process and module architecture)(Key abstractions)(mechanisms)



On Building Quality Classes and Objects:

Measuring quality of an abstraction
•A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible and all parts should be held in a uniform framework.
•Parts-Class n objects; framework-mechanism

•Design of classes and objects is incremental and iterative process.
•Cost of refining

How can one know if a given class or object is well designed?
•Coupling
•Cohesion
•Sufficiency
•Completeness
•Primitiveness

Coupling:

• The measure of the strength of association established by a connection from one module to another
• Strong coupling complicates a system.
• Weakest possible coupling
• Eg. power supply in speaker cabinet
• Coupling with regards to modules
• Coupling and inheritance.

Cohesion:

• The measure of degree of connectivity among elements of a single module (class/object).
• Least desirable- coincidental cohesion
• (Eg. dogs and spacecraft)
• Most desirable- functional cohesion
• (Eg. dog, whole dog and nothing but dog)

Sufficiency:

•The class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction.

Completeness:

Does the class capture all of the useful behavior of the thing being modeled to be re-usable? What about future users (reusers) of the class? How much more time does it take to provide completeness? Is it worth it?

Primitiveness:

Do all the behaviors of the class satisfy the condition that they can only be implemented by accessing the state of the class directly? If a method can be written strictly in terms of other methods in the class, it isn't primitive.

Primitive classes are smaller, easier to understand, with less coupling between methods, and are more likely to be reused. If you try to do too much in the class, then you're likely to make it difficult to use for other designers.