UNIT - II

CLASSES AND OBJECTS

Both the engineer and the artist must be intimately familiar with the materials of their trade. oil versus water colors, steel versus Aluminium, bolts versus nails, object versus classes.

Each of these materials serves similar functions. each has its own specific properties and uses.

The architect may not know the most efficient way to drive a nail, but the architect must understand when it is appropriate to use nails or bolts or glue or welds.

when we use object oriented Methods to analyze or design a complex s/w system, our basic building blocks are classes and objects.

1. The Nature of an object:

The ability to recognize physical objects is a skill that human learns at a very early age.

A brightly colored Ball will attract an infant's attention, but typically, if you hide the ball, the child will not try to look for it; when the object leaves her field of vision; as far as she can determine, it ceases to exist.

It is not until near the age of one that a child normally develops what is called the object concept; A skill ie of critical importance to future cognitive development.

* What is and what isn't an object:

An object as a tangible entity that exhibits some well-defined behavior. An object is any of the following:

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

In s/w the term object was first formally applied in Simula language; objects typically existed in Simula Programs to Simulate some aspect of reality.

Real world objects are not the only kinds of objects that are of interest to us during s/w development.

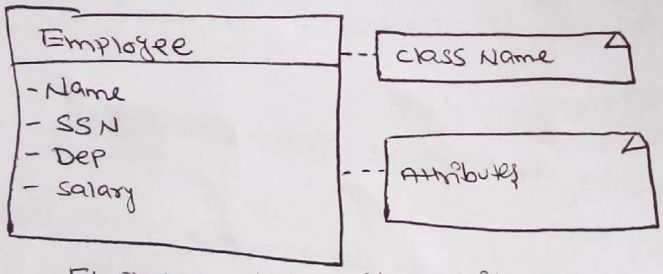
Some objects may have crisp conceptual boundaries yet represent intangible events or processes.

* An object has state, exhibits some well-defined behavior & has a unique identity.

" An object is an entity that has state, behavior, and identity, the structure & behavior of similar objects are defined in their common class. The terms instance and objects are interchangable".

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 those properties.



Employee class with attributes.

Behavior: No objects exists in isolation. Rather, objects are acted on and themselves act on other objects.

Behavior is how an object acts and reacts, in term of its state changes & message passing.

The Behavior of an object represents its outwardly visible activity.

Operations:

An operation denotes a service that a class offers to its clients. we have found that a client typically performs five kinds of operations on an object. The 3 most common kinds of operations are the following:

- Modifier: Alter the state of an object.
- selector: Access the state but doesnot change object.
- Iterator: Permits All parts of an object to access.

Two kinds of operations are common:

- constructor: - create object.
- Destructor: - destroy object.

Roles & Responsibilities:

Collectively, all of the methods associated with a particular object comprise its protocol.

"A Role is a mask that an object wears and so defines a contract blw 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 the contracts it supports".

Objects as Machines:

The existence of state within an object means that the order in which operations are invoked is important.

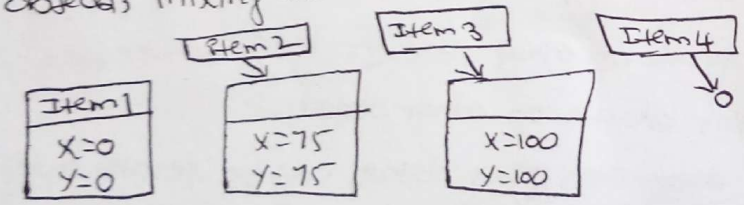
This gives rise to the idea that each object is like a tiny, independent machine.

Identity:

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

"most Programming & Database languages use variable names to distinguish temporary objects, mixing addressability and identity.

(A)



object identity

What is not object: some more modern languages like Haskell are functional rather than object oriented.

Most modern languages are multi paradigm and support object oriented programming in addition to one or more alternative paradigms

functional
procedural
declarative etc.

7. Relationships among Objects

Objects contribute to the behavior of a system by collaborating with one another. "Instead of a bit-grinding processor raping and plundering data structures we have a universe of well-behaved objects that courteously ask each other to carryout their various desires".

The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results.

Two kinds of objects relationships are 1. Links.
2. Aggregations.

Links:

The term link derives from Rumbaugh et al, who define it as a "physical or conceptual connection b/w objects".

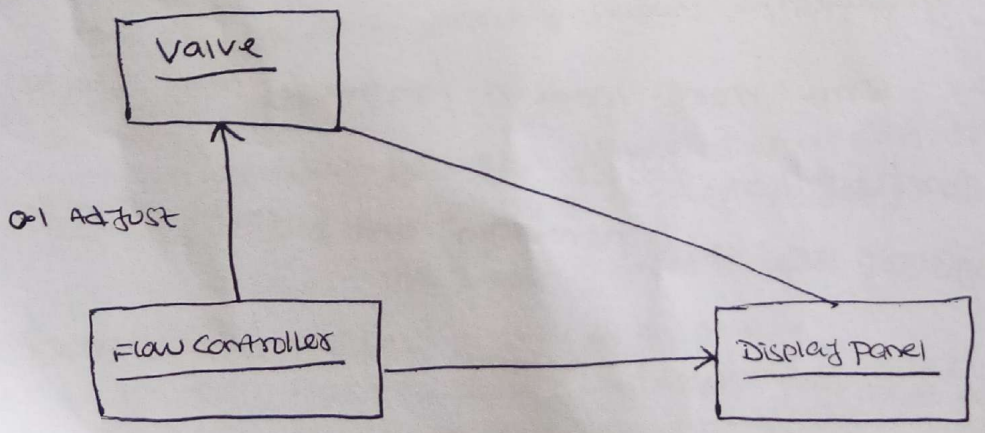
An object collaborates with other objects through its links to these objects.

A Link denotes the specific association through which one object applies the service of another object.

A line b/w two object icons represents the existence of a link b/w the two and means that messages may pass along this path.

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

1. Controller: this object can operate on other objects.
2. Server: this object doesn't operate on other objects.
3. Proxy: this object can both operate on other objects and be operated on by other objects



* Links

sibility:

Consider two objects, A and B, with a link between the two.
In order for A to send a message to B, B must be visible to A in some manner.
We must consider the visibility across links because as decisions here dictate the scope and access of the objects on each side of a link.

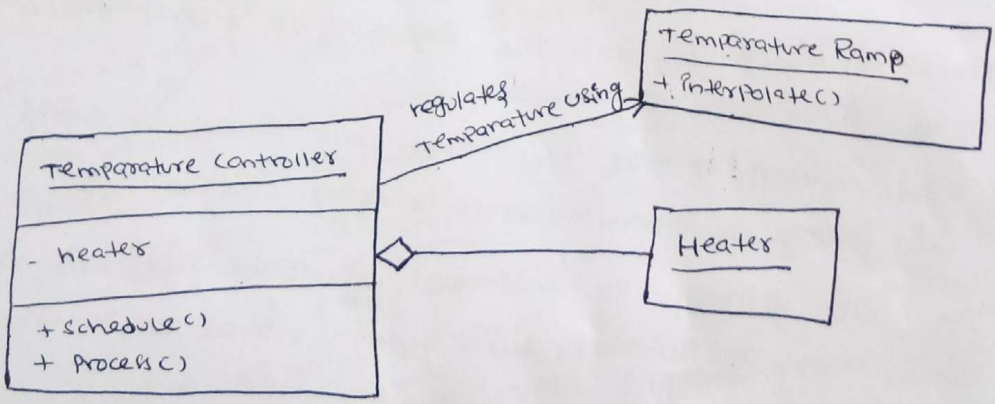
Synchronization:

Whenever one object passes a message to another across a link, the two objects are said to be synchronized.
When one active object has a link to a passive one, we must choose one of three approaches to Synchronization.

1. Sequential
2. Guarded
3. Concurrent.

Aggregation:

Whereas links denote peer to peer or client/suppiier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole to its parts.
Aggregation is a specialized kind of association.



Aggregation

* Aggregation is sometimes better because it encapsulates parts of secrets of the whole.

The Nature of a class:

The concept of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class.

There are important differences Blw these two terms.

what is and what isn't a class

Whereas an object is a concrete entity that exists in time and space, A class represents only an abstraction. the "essence" of an object, as it ~~were~~

We may speak class Mammal, which represents the characteristics common to all mammals.

To identify a particular mammal in this class, we must speak of "this mammal or that mammal"

"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.

what isn't a class: An object is not a class - objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects.

Interface and Implementation

A class serves as a sort of binding contract between an abstraction and all of its clients. By capturing these decisions in the interface of a class, a strongly typed programming language can detect violations of this contract during compilation.

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

we can further divide the interface of a class into four parts

1. Public: A declaration i.e accessible to all clients.
2. Protected: A declaration i.e accessible only to the class itself and its sub classes.
3. Private: A declaration i.e accessible only to the class itself.
4. Package: A declaration i.e accessible only by classes in the same package.

Class Lifecycle:

we may come to understand the behavior of a simple class just by understanding the semantics of its distinct public operations in isolation.

The behavior of more interesting classes involves the interaction of their various operations over the lifetime of each of their instances.

An instance of classes act as little machines, and since all such instances embody the same behavior, we can use the class to capture these common event- and time-ordered semantics.

4. Relationships among classes

Consider for a moment the similarity and difference among the following classes of objects:

Flowers, daisies, red roses, yellow roses, petals, and ladybugs. We can make the following observations.

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

We establish relationships b/w two classes for one of two reasons.

1. Class relationship might indicate some sort of sharing.
2. class relationship might indicate some kind of semantic connection.

In all, there are three basic kinds of class relationships.

1. Generalization | Specialization, denoting an "is a" relationship.

Ex: A Rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flowers.

2. Whole | Part, which denotes a "part of" relationship.

Ex: A petal is not a kind of flower; it is part of flowers.

3. Association, which denotes some semantic dependency among otherwise unrelated classes, such as b/w ladybugs & flowers.

Associations:

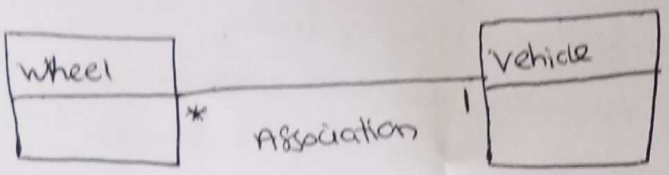
associations are the most general but also the most semantically weak.

The identification of associations among classes is often an activity of analysis and early design, at which time we begin to discover the general dependencies among our abstractions.

Semantic Dependencies:

An Association only denotes a semantic dependency ... does not state the direction of this dependency, nor does it state the exact way in which one class relates to another.

Through the creation of associations, we come to capture the participa... in a semantic relationship, their roles & their cardinality.



* one to many relationship

Multiplicity:

→ Introduced a one to many association, meaning that for each instance of the class Vehicle, there are zero or more instances of the class Wheel, and for each wheel, there is exactly one vehicle.

This denotes the multiplicity 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.

A one-to-one relationship denotes a very narrow association. Each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale; many-to-many relationships are also common.

Inheritance:

Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships.

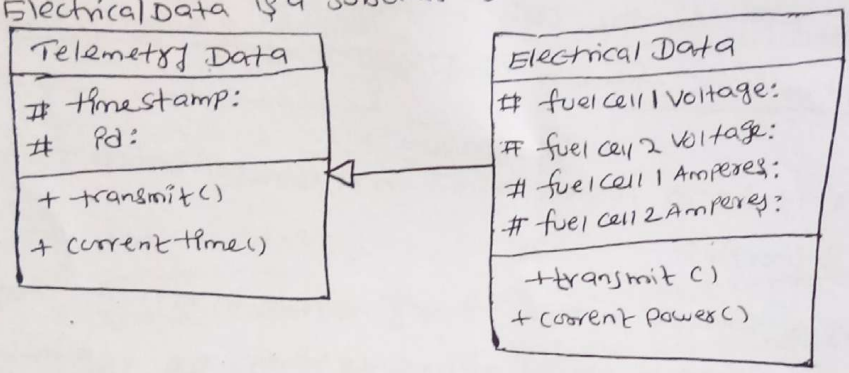
Inheritance is an insufficient means of expressing all of the rich relationships that may exist among the key abstractions in a given problem domain.

gle Inheritance:

Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one or more other classes. we call the class from which another class inherits its superclass.

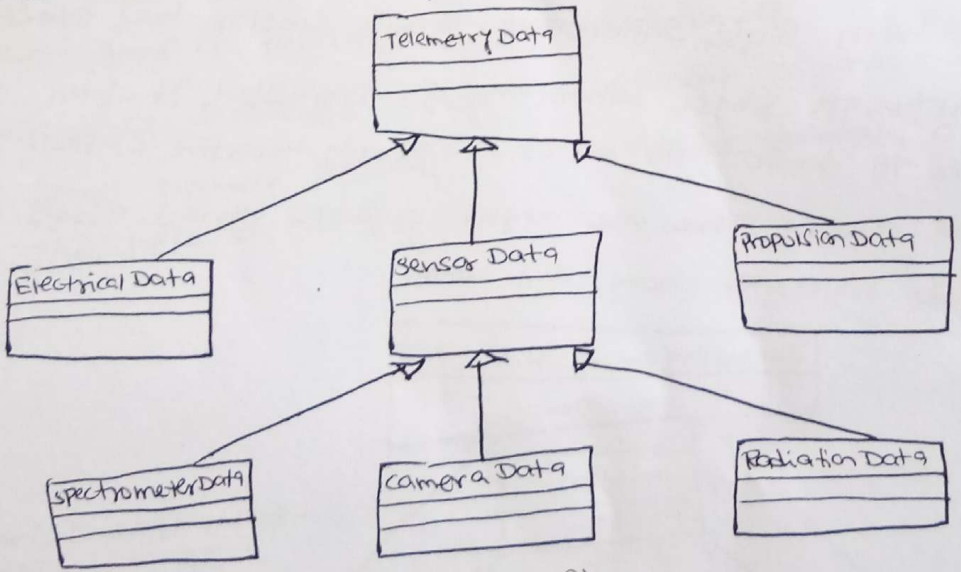
Ex: TelemetryData is a superclass of ElectricalData, similarly, we call a class that inherits from one or more classes a subclass;

ElectricalData is a subclass of TelemetryData.



* ElectricalData Inherits from the superclass TelemetryData

There is a very real tension between inheritence and encapsulation. To a large degree the use of inheritence exposes some of the secrets of an inherited class.



Single inheritance

* Inheritance means that subclasses inherit the structure of their superclass.

Polymorphism:

For the classes also inherit the behavior of their superclass. Thus instances of the class Electrical Data may be acted on with the operations current time, current power and transmit.

For the class Telemetry Data, the fuction transmit may transmit the identifier of the telemetry stream & its timestamp. But the same fuction for the class Electrical Data may invoke the Telemetry Data transmit fuction and also transmit its voltage and current values.

This behavior is due to polymorphism. in a generalization, such operations are called Polymorphic.

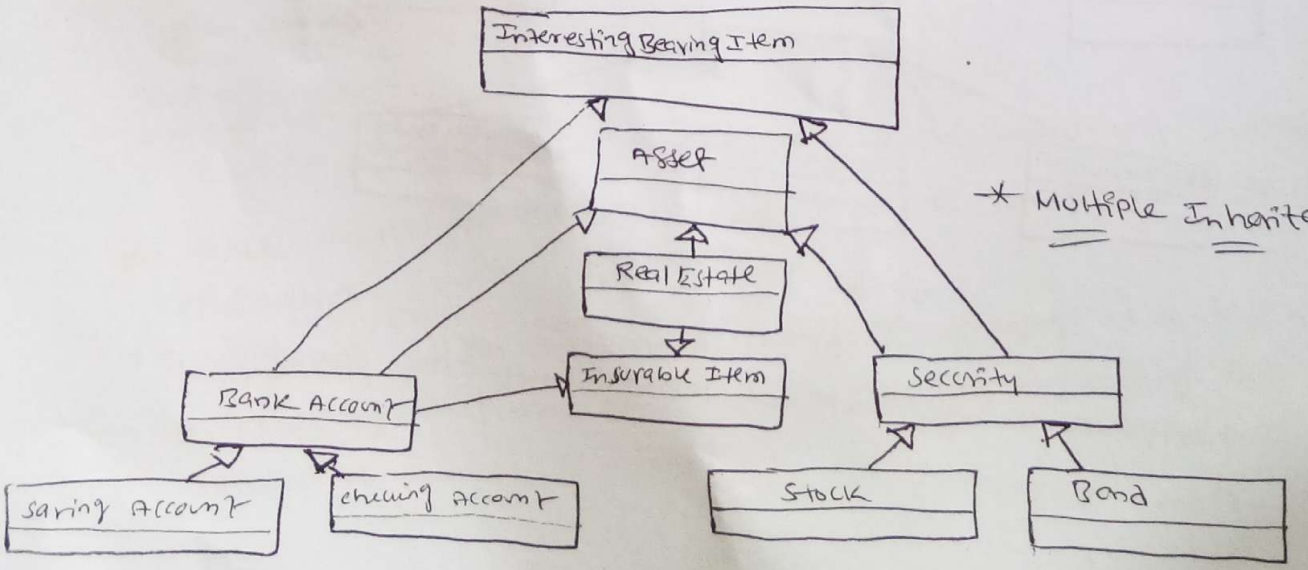
Polymorphism is a concept in type theory where in a name may denote instances of many different classes as long as they are related by some common superclass.

Inheritence without polymorphism is possible, but it is certainly not very useful.

Multiple Inheritance:

With single inheritance, each subclass has exactly one superclass. although single inheritance is very useful, it often forces the programmer to derive from one of two equally attractive classes.

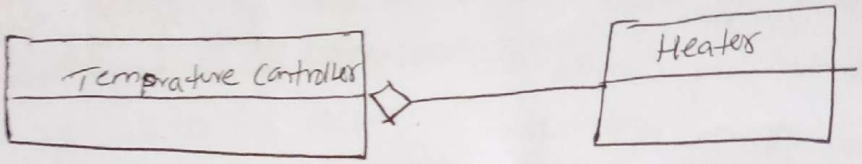
Consider for a moment how one might organize various assets such as savings accounts, realestate, stocks, and bonds.



* Multiple Inheritance

Aggregation:

We also need aggregation relationships which provides 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.



* Aggregation

Physical Containment:

In the case of the class Temperature Controllers, we have aggregation as contained by value, a kind of physical containment meaning that the Heater objects does not exist independently of its enclosing Temperature controller instance.

Dependencies:

A side from Inheritance, Aggregation, and Association, there is another group of relationships called dependencies.
A dependency indicates that an element on one end of the relationship in some manner depends on the elements ~~changes~~ on the other end of the relationship.

5. The Interplay of Classes & Objects

Classes and objects are separate yet intimately related concepts, specifically, every object is the instance of some class, and every class has zero or more instances.

Practically all applications classes are static; ∴ their existence, semantics, and relationships are fixed prior to the execution of a program. Objects are typically created and destroyed at a furious rate during the lifetime of an application.

Relationships B/w classes and objects:

Consider the classes and objects in the implementation of an air traffic control system. Some of the more important abstraction include planes, flight plans, runways, and airspace.

The meaning of these classes & objects are relatively static. They must be static, for otherwise one could not build an application that embodied knowledge of such commonsense facts as that planes can take off, fly, and then land, and that two planes should not occupy the same space at the same time.

Role of classes & objects in analysis & Design:

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

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

Collectively, we call such classes and objects the key abstractions of the problem, and we call these cooperative structure the mechanism of the implementation.

3.6 On Building Quality Classes and Objects

A system should be built with a minimum set of unchangeable parts, those parts should be general as possible; and all parts of the system should be held in a uniform framework." With object-oriented development these parts are the classes & objects that make up the key abstractions of the system, and the framework is provided by its mechanisms.

Measuring the Quality of an Abstraction

We suggest five meaningful metrics:

* coupling
* cohesion (action or fact)
* Sufficiency
* completeness
* Primitiveness.

Coupling: It a notation borrowed from structured design, but with a liberal interpretation it also applies to object oriented design.

Cohesion: The idea of cohesion also comes from structured design. cohesion measures the degree of connectivity among the elements of a single module.

Sufficiency: closely related to the ideas of coupling & cohesion are the criteria that a class or module should be sufficient, complete, and Primitive.

By sufficient, we mean that the class or module captures enough characteristics of the abstraction to permit meaningful & efficient interaction.

Completeness: By complete, we mean that the interface of the class or module captures all of the meaningful characteristics of the abstraction.
Completeness is a subjective matter, and it can be overdone.

Primitiveness: Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction.

An operation is indisputably primitive if we can implement it only through access to the underlying representation.

Choosing operations: Create
Destroy,
Modify,
Reorganize.

Functional Semantics: Reusability,
Complexity,
Applicability,
Implementation knowledge.

Time & Space Semantics: This means that we must specify our decisions about the amount of time it takes to complete an operation and the amount of storage it needs.

Choosing Relationship: Choosing the relationships among classes and among objects is linked to the selection of operations.

The Law of Demeter: One useful guide line in choosing the relationships among objects is called the "Law of Demeter."
Which states that "the method of class should not depend in any way on the structure of any class, except the immediate structure of their own class.

PART-II

Classification

Classification is the means whereby we order knowledge. In object oriented design, recognized the sameness among things allows us to expose the commonality within key abstractions and mechanisms and eventually leads us to smaller applications and simpler architectures.

1. The Importance of Proper classification:

The identification of classes & objects is a challenging part of object-oriented analysis and design.

Identification involves both discovery & invention.

Through discovery, we come to reorganize the key abstractions and mechanisms that form the vocabulary of our problem domain.

Through invention, we devise (plan) generalized abstractions as well as new mechanisms that specify how objects collaborate.

Ultimately, discovery and invention are both problems of classification, and classification is fundamentally a problem of finding sameness.

When we classify, we seek to group things that have a common structure or exibit a common behavior.

Intelligent classification is actually a part of all good science.

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

classification also plays a role in allocating processes to processors.

we place certain processes together in the same processor or different processors, depending on packaging, performance, or reliability concerns.

The Difficulty of classification:

We defined an object as something that has a crisply defined boundary. However, the boundaries that distinguish one object from another are often quite fuzzy.

- The fact that intelligent classification is difficult is hardly new information. Since there are parallels to the same problems in object oriented design, consider for a moment the problems of classification in two other scientific disciplines: Biology and Chemistry.

- The Incremental and Iterative Nature of classification:

Intelligent classification is intellectually hard work and that it best comes about through an incremental & iterative process.

In s/w Engineering, "the development of individual abstractions often follows a common pattern, First problems are solved ad hoc. As experience accumulates, some solutions turn out to work better than others, and a sort of folklore is passed informally from person to person. Eventually, the useful solutions are understood more systematically, and they are codified and analyzed.

- The incremental & iterative nature of classification directly impacts the construction of class and object hierarchies in the design of a complex s/w system.

Why then, is classification so hard? There are two important reasons:

1. There is no such things as a "perfect" classification, although certainly some classifications are better than others.
2. Intelligent classification requires a tremendous amount of creative insight.

Identifying classes and objects:

The problem of classification has been the concern of countless philosophers, linguists, cognitive scientists, and mathematicians, since before the time of plato, it is reasonable to study their experiences and apply what we learn to object oriented design.

classical and Modern Approches:

Historically, there have been any three general approches to classification:

1. classical categorization
2. conceptual clustering
3. prototype theory.

classical categorization:

In the classical approch to categorization, "All the entities that have a given property or collection of properties in common form a category. such properties are neccessary & sufficient to define the category"

Ex: married people constitute a category: one is either married or not,

classical categorization come to us first from plato, and then from Aristotle through his classification of plants & animals,

the classical approch to categorization is also reflected in modern theories of child development.

The child aquires skills in classifying these objects, first using basic categories such as dogs, cats, and toys.

conceptual clustering:

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

In this approch, classes are generated by first formulating conceptual descriptions of these classes and then classifying the entities according to the descriptions".

conceptual clustering is closely related to fuzzy set theory, in which objects may belong to one or more groups in varying degree of fitness.

Prototype Theory:

Classical categorization and conceptual clustering sufficiently expressive to account for most of the classifications we need the design of complex S/w systems.

there are still some situations in which these approaches to classification are inadequate.

There are some abstractions that have neither clearly bounded properties nor concepts.

This is why the approch is called Prototype theory:

A class of objects is represented by a prototypical object, and an object is considered to be a member of this class if and only if it resembles this prototype in significant ways.

Lakoff and Johnson apply Prototype theory to the earlier problem of classifying chairs.

Interactional properties are prominent among the kinds of properties that count in determining sufficient family resemblance.

This notation of Interactional properties is central to the idea of Prototype theory.

Applying classical and Modern theories:

Actually, these three approches to classification have direct application to object-oriented design.

we identify classes and objects first according to the properties relevant to our particular domain. Here we focus on identifying the structures and behavior that are part of the vocabulary of our problem space.

More directly, these three approches to classification provide the theoritical foundation of object-oriented analysis, which offers a no. of pragmatic parametrics & rules of thumb that we may apply to identify classes & objects in the design of a complex s/w system.

Classical Approaches:

A no of methodologists have proposed various sources of classes and objects, derived from the requirements of the problem domain. we call these approaches classical because they derive primarily from the principles of classical categorization.

Ex: Shlaer and Mellor suggests that candidate classes & objects usually come from one of the following sources:

* Tangible Things — Cars, telemetry data, pressure sensors
* Roles — Mother, Teacher, Politician
* Events — Landing, interrupt, request.
* Interactions — Loan, Meeting, intersection.

From the perspective of DB modeling, Rols offers a similar list:

* People — Humans who carry out some function
* Places — Areas set aside for people or things
* Things — Physical objects, or groups of objects, that are tangible.
* organizations — organized collection of people, resources, facilities & capabilities
* Concepts — Principles or ideas not tangible per se;
* Events — Things that happen usually to something else at a given date & time

Coad and Yourdon suggest yet another set of sources of potential objects:

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

Behavior Analysis:

Object oriented Analysis focuses on dynamic behavior of the primary source of classes & objects. these approaches are more akin to conceptual clustering: we form classes based on groups of objects that exhibit similar behavior.

Domain Analysis:

Domain Analysis, on the otherhand, seeks to identify the classes & objects that are common to all applications within a given domain, such as patient record tracking, bond trading, compilers.

The idea of domain analysis was first suggest by neighbors.

Domain analysis as "an attempt to identify the objects, operations, and relationships domain experts perceive to be important about the domain.

Domain analysis are: Generic model of the domain
Existing system within the domain
Identify similarities & differences,
refine the generic model.

Use case Analysis: Use case As a "A behaviorally related sequence of transactions performed by an actor in a dialogue with the system to provide some measurable value to the actor.

CRC Cards:

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

CRC cards have proven to be a useful development tool

A CRC card is nothing more than a 3x5 index card.

Informal English Description:

Abbot suggest writing an English description of the problem and then underlining the nouns & verbs.

noun represent candidate objects, verb represent candidate operations.

Structured Analysis: Some organizations have tried to use the products of structured Analysis as a front end to object-oriented design.

This approach starts with an essential model of the system.

Key Abstractions and Mechanisms:

A Key Abstraction is a class or object that forms part of the vocabulary of the problem domain.

The primary value of identifying such Abstraction is that they give boundaries to our problem:

they highlight the things that are in the system and therefore relevant to our design, and they suppress the things that are outside the system and therefore superfluous.

The term mechanism to describe any structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem.

whereas the design of a class embodies the knowledge of how individual objects behave, a mechanism is a design decision about how collections of objects cooperate. Mechanisms that represent patterns of behavior.

Identifying Key Abstractions:

The identification of key Abstractions is highly-domain specific. As Goldberg states, "The appropriate choice of objects depends, of course, on the purposes to which the application will be put and the granularity of information to be manipulated."

The Identification of key Abstractions involves two processes:

Discovery and

Inventions.

* Discovery is we come to recognize the Abstractions used by domain experts:
* Invention is we create new classes & objects that are necessarily part of the Problem domain but are useful artifacts in the Design or Implementation.

Refining key Abstractions:

once we Identify a certain key abstraction as a candidate, we must evaluate it according to the metrics.

Given a new abstraction, we must place it in the context of the existing class and object hierarchies we have designed.

Placing classes and objects at the right levels of Abstraction is difficu

Naming key abstractions:

Naming things properly so that they reflect their semantics - is often treated lightly by most developers yet is important in capturing the essence of the abstractions we are describing.

Slw should be written as carefully as English Prose, with consideration given to the reader as well as to the computer.

All the names we may need just to identify a single object: we have the name of the object itself, the name of its class, and the name of the module in which that class is declared.

We offer the following suggestions:

- object should be named with Proper noun phrases; sensor-shape
- classes should be named with common noun phrases, sensor-shape.
- Reflects the names used and recognized by the domain expert.
- Modifier operations should be named with active verb phrases.
- Selector operations should imply a query
- The use of underscores and styles of capitalization are largely matters of personal taste.

Identifying Mechanisms:

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, any of the following designs might be considered:

- a mechanical linkage connects the accelerator directly to the fuel injectors.
- an electronic mechanism connects a pressure sensor
- No linkage exists.

which mechanism a developer chooses from a set of alternatives is most often a result of other factors, such as cost, reliability, manufacturability and safety.

Mechanisms as Patterns:

Mechanisms are actually one in a spectrum of patterns we find in well structured SW systems.

At the low end of the food chain, we have idioms.

An idiom is an expression peculiar to a certain programming language or application culture, representing a generally accepted convention for use of the language.

Idioms play an important role in codifying low-level patterns.

"many common programming tasks are idiomatic".

Whereas idioms are part of a programming culture, at the high end of the food chain, we have frameworks.

A Framework is a collection of classes that provides a set of services for a particular domain;