**Complexity**

- **Complexity** takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems,

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

- Intra-component *linkages* are generally stronger than inter-component linkages.

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

- **A complex system** that works is invariably found to have evolved from a simple system that worked.
Canonical Form of a Complex System

Combining the concept of the class and object structures together with the five attributes of a complex system (hierarchy, relative primitives [i.e., multiple levels of abstraction], separation of concerns, patterns, and stable intermediate forms), we find that virtually all complex systems take on the same (canonical) form of Architecture.
Industrial-Strength Software

- A successful project
  - one whose deliverables meet the customer’s experience
  - develop in a timely and economical fashion
  - resilient to change

- Features common to many successful projects
  - Intelligent use of OO decomposition
  - Existence of a strong architecture vision
  - Application of a well-managed iterative and incremental development lifecycle

Object-Oriented Decomposition

- Employs the principles of Abstraction, Encapsulation, Modularity, and Hierarchy
- Yields architectures that consist of collections of cooperative objects, whose classes are organized into hierarchies

- Recognize that the class is a necessary but insufficient vehicle for decomposition
**Architecture Vision**

- Seek to craft systems that
  - Are constructed in well-defined layers of abstraction
  - Have a clear separation of concerns among each layer, and between the interface and implementation of each individual layer
- Build simple architectures, in which common behavior is achieved through common patterns
- At all times, maintain control of the system’s strategic and tactical architecture
Iterative and Incremental Lifecycle

- The focus of development process should be the successive refinement of architectural prototypes, each representing intermediate stable forms that can be continuously integrated and measured.
- Distinguish between architectural prototypes and behavioral prototypes.
- Each release must be undertaken for noble reasons.
Developing Complex Systems

- **Engineering as a Science and an Art**

- **Meaning of Design**
  - Design involves balancing a set of competing requirements.
  - **Importance of Model Building** (*model building* appeals to the principles of decomposition, abstraction, and hierarchy)

- **Elements of SW Design methodology**
  - **Notation:** the language for expressing each model
  - **Process:** the activities leading to the orderly construction of the system's models
  - **Tools:** the artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed
Summary-1

- **Software is inherently complex;** the complexity of software systems often exceeds the human intellectual capacity.

- **The task of the software development team is to engineer the illusion of simplicity.**

- Complexity often takes the form of a hierarchy; it is useful to model both the "is a" and the "part of" hierarchies of a complex system.

- Complex systems generally evolve from **stable intermediate forms.**

- There are fundamental limiting factors of human cognition; we can address these constraints through the use of **decomposition, abstraction, and hierarchy.**
Complex systems can be viewed by focusing on either things or processes; there are compelling reasons for applying object-oriented decomposition, in which we view the world as a meaningful collection of objects that collaborate to achieve some higher-level behavior.

Object-oriented analysis (OOA) and design (OOD) is the method that leads us to an object-oriented decomposition. OOD uses a notation and process for constructing complex software systems and offers a rich set of models with which we may reason about different aspects of the system under consideration.
Evolution of the Object Model

1. The shift in focus from programming-in-the-small to programming-in-the-large
2. The evolution of high-order programming languages
Generations of Programming Languages-1

First-generation languages (1954–1958):
FORTRAN I, ALGOL 58, Flowmatic, IPL V

FORTRAN II, ALGOL 60, COBOL, Lisp
Generations of Programming Languages-2

Third-generation languages
PL/1 (FORTRAN + ALGOL + COBOL),
ALGOL 68, Pascal, Simula

Many different languages were invented, but few endured. However, the following are worth noting: C (Efficient; small executables), FORTRAN 77.
Generations of Programming Languages-3

Object-orientation boom (1980–1990, but few languages survive)
Smalltalk 80, C++, Ada83, Eiffel
Generations of Programming Languages-4

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:  Visual Basic for the Microsoft .NET Framework
Object-Oriented Programming

- **Object-oriented programming** (OOP) focuses on tactical issues

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

- **OOP Language** must support: Data abstraction, Encapsulation, and Inheritance with polymorphism
Object-Oriented Design

- **Object-oriented design** (OOD) focuses on the invention of the architecture that yields the desired behavior of the system.

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

- **Object-oriented analysis** (OOA) focuses on the discovery of the behavior and the corresponding abstraction that are central to the vocabulary of the problem domain.

- **OOA** is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.
Programming Paradigms

1. Procedure-oriented | Algorithms
2. Object-oriented | Classes and objects
3. Logic-oriented | Goals, often expressed in a predicate calculus
4. Rule-oriented | If–then rules
5. Constraint-oriented | Invariant relationships
Elements of Object Model-1

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

2. **Encapsulation**: Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

3. **Modularity**: Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
Elements of Object Model-2

4. **Hierarchy**: Hierarchy is a ranking or ordering of abstractions. The two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

5. **Typing**: Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

*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.
Elements of Object Model-3

6. **Concurrency:** Concurrency is the property that distinguishes an active object from one that is not active.

7. **Persistence:** Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created).
Benefits of Object Model

- helps us to exploit the expressive power of object-based and object-oriented programming languages
- encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks
- produces systems that are built on stable intermediate forms, which are more resilient to change
The maturation of software engineering has led to the development of object-oriented analysis, design, and programming methods, all of which address the issues of programming-in-the-large.

There are several different programming paradigms: procedure-oriented, object-oriented, logic-oriented, rule-oriented, and constraint-oriented.

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

Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.
Summary-2

- **Modularity** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- **Hierarchy** is a ranking or ordering of abstractions.
- **Typing** is the enforcement of the class of an object, such that objects of different types may not be interchanged or, at the most, may be interchanged only in very restricted ways.
- **Concurrency** is the property that distinguishes an active object from one that is not active.
- **Persistence** is the property of an object through which its existence transcends time and/or space.
What Is an Object

- An object is
  - A tangible and/or visible thing
  - Something that may be comprehended intellectually
  - Something toward which thought or action is directed
- An object is an entity that has state, behavior, and identity.
- The structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable.
State

- The state of an object encompasses
  - all of the (usually static) properties of the object
  - the current (usually dynamic) values of each of these properties.
- A property is an inherent or distinctive characteristic, trait, quality, or feature that contributes to making an object uniquely that object.
- The state of an object represents the cumulative results of the object’s behavior.
Behavior

- **Behavior** is how an object acts and reacts, in terms of its state changes and message passing.
- The behavior of an object represents its outwardly visible activity.

![Employee schema diagram](image)
Operations

- **Operation** denotes a service that a class offers to its clients.
  - **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
  - **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
Operations

- All the operations (methods) associated with a particular object denote its protocol.
- Responsibilities are meant to convey a sense of the purpose of an object and its place in the system.
- Objects can play many different roles during their lifetime.
- Objects may be passive or active.
Identity

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

(a) DplItem item1;
    DplItem* item2=new DplItem(point(75,75));
    DplItem* item3=new DplItem(point(100,100));
    DplItem* item4=0;

(b) item1.move(item2->location());
    item4=item3;
    item4->move(point(38,100));

(c) item2=&item1;
    item4->move(item2->location());
Relationships among Objects

- Objects contribute to the behavior of a system by collaborating with one another.

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

- Object relationships include
  1. Links
  2. Aggregation
Links

- A link is defined as a "physical or conceptual connection between objects"
- A link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another.
Across a link, an object may play one of three roles.

1. **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.

2. **Server:** This object doesn't operate on other objects; it is only operated on by other objects.

3. **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.
Aggregation

- **Aggregation** denotes a whole/part hierarchy, with the ability to navigate from the whole to its parts (TemperatureController/Heater)

- **Aggregation** may or may not denote physical containment
What Is a Class

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

- Sometimes **abstractions** are so complex that they cannot be conveniently expressed in terms of a single class declaration.

- A **class serves** as a sort of binding contract between an abstraction and all of its clients.

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

- The **implementation** of a class is its inside view, which encompasses the secrets of its behavior.
The interface of a class can be divided into four parts:

1. Public: a declaration that is accessible to all clients
2. Protected: a declaration that is accessible only to the class itself and its subclasses
3. Private: a declaration that is accessible only to the class itself
4. Package: a declaration that is accessible only by classes in the same package
Relationships among Classes

- A class relationship indicates some sort of sharing and some kind of semantic connection.

- Class relationships include:
  - Association
  - Inheritance
  - Aggregation
  - Using, Instantiation, Metaclass, etc.
**Association**

- An **association** only denotes a semantic dependency and does not state the direction of this dependency.
- In general, an association implies **bidirectional** navigation.
- In practice, there are three common kinds of **multiplicity** across an association:
  1. One-to-one, 2. One-to-many, and 3. Many-to-many.
Inheritance

- Inheritance expresses generalization/specialization (is-a) 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

- 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.
**Aggregation**

- **Aggregation** denotes a **whole/part** hierarchy, with the ability to navigate from the whole to its parts (TemperatureController/Heater)
- **Aggregation** may or may not denote **physical** containment
Measuring the Quality of an Abstraction

1. **Coupling**: the strength of association established by a connection from one module to another.
2. **Cohesion**: the degree of connectivity among the elements of a single module
3. **Sufficiency**: By sufficient, we mean that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction.
4. **Completeness**: By complete, we mean that the interface of the class or module captures all of the meaningful characteristics of the abstraction.
5. **Primitiveness**: Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction.
Functional Semantics

- In OOD, we design the methods of a class as a whole because all these methods cooperate to form the entire protocol of the abstraction.

1. Reusability: Would this behavior be more useful in more than one context?
2. Complexity: How difficult is it to implement the behavior?
3. Applicability: How relevant is the behavior to the type in which it might be placed?
4. Implementation knowledge: Does the behavior's implementation depend on the internal details of a type?

Time and Space tradeoffs: we must specify our decisions about the amount of time it takes to complete an operation and the amount of storage it needs.
Key Abstractions and Mechanisms

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

1. **Identify the classes and objects** (key abstractions) that form the vocabulary of the problem domain

2. **Invent the structures** (mechanisms) whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem
Summary-2

- An **object** has **state**, **behavior**, and **identity**.
- The **structure** and **behavior** of similar objects are **defined in their common class**.
- 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.
- **Behavior** is how an object acts and reacts in terms of its state changes and message passing.
- **Identity** is the property of an object that distinguishes it from all other objects.
A class is a set of objects that share a common structure and a common behavior.

The three kinds of relationships include association, inheritance, and aggregation.

Key abstractions are the classes and objects that form the vocabulary of the problem domain.

A mechanism is a structure whereby a set of objects work together to provide a behavior that satisfies some requirement of the problem.

The quality of an abstraction may be measured by its coupling, cohesion, sufficiency, completeness, and primitiveness.
Importance of Proper Classification

- Classification is the means whereby we order knowledge.
- There is no such thing as the "perfect" class structure, nor the "right" set of objects.
- The identification of classes and objects is a challenging part of object-oriented analysis and design. Our experience shows that identification involves both discovery and invention.
- Through discovery, we come to recognize the key abstractions and mechanisms that form the vocabulary of our problem domain.
- Through invention, we devise generalized abstractions as well as new mechanisms that specify how objects collaborate.
- Classification helps us to identify generalization, specialization, and aggregation hierarchies among classes.
- Classification is both incremental and iterative.
**Identifying Classes and Objects-1**

1. **Classical categorization**
   
   - All the entities, which have a given property or collection of properties in common, form a **category**
   
   - The classical approach uses related properties as the criteria for **sameness** among objects.

2. **Conceptual clustering**
   
   - **Conceptual clustering** attempts to explain how knowledge is represented
   
   - **Classes** (clusters of entities) are generated by first formulating **conceptual descriptions** of these classes and then classifying the entities according to the descriptions
3. Prototype theory

- In **prototype theory**, we group things according to the degree of their relationship to concrete prototypes.
- 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.
**Classical OO Approaches-1**

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

  | **Tangible things** | Cars, telemetry data, pressure sensors |
  | **Roles**           | Mother, teacher, politician            |
  | **Events**          | Landing, interrupt, request            |
  | **Interactions**    | Loan, meeting, intersection             |
Coad and Yourdon suggest yet another set of sources of potential objects:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>&quot;Is a&quot; and &quot;part of&quot; relationships</td>
</tr>
<tr>
<td>Other systems</td>
<td>External systems with which the application interacts</td>
</tr>
<tr>
<td>Devices</td>
<td>Devices with which the application interacts</td>
</tr>
<tr>
<td>Events remembered</td>
<td>A historical event that must be recorded</td>
</tr>
<tr>
<td>Roles played</td>
<td>The different roles users play in interacting with the application</td>
</tr>
<tr>
<td>Locations</td>
<td>Physical locations, offices, and sites important to the application</td>
</tr>
<tr>
<td>Organizational units</td>
<td>Groups to which users belong</td>
</tr>
</tbody>
</table>
Behavior Analysis

- **Behavior Analysis** focuses on dynamic behavior as the primary source of classes and objects.

- **Wirfs-Brock, Wilkerson, and Wiener**, for example, emphasize responsibilities, which denote "the knowledge an object maintains and the actions an object can perform. 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".

- **Rubin and Goldberg** offer an approach to identifying classes and objects derived from system functions.
Domain and Use Case Analyses

- Neighbors uses domain analysis, an attempt to identify the objects, operations, and relationships [that] domain experts perceive to be important about the domain

- Jacobson et al. define a use case as "A behaviourally related sequence of transactions performed by an actor in a dialogue with the system to provide some measurable value to the actor"
CRC Analysis

- CRC stands for Class/Responsibilities/Collaborators and was proposed by Beck and Cunningham as a tool for teaching object-oriented programming.

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

- A **key abstraction** is a class or object that forms part of the vocabulary of the problem domain.
- The identification of **key abstractions** (classes and objects) involves two processes: **discovery** and **invention**.
- A **mechanism** describes any structure whereby sets of objects collaborate to provide the behaviors that satisfy the requirements of the problem.
- Mechanisms are the soul of the design. They represent **strategic design decisions**, as does the design of a class structure.
**Mechanisms as patterns**

- **Idiom**: an expression peculiar to a certain programming language or application culture, representing a generally accepted convention for use of the language (low-level pattern)
- **Mechanism** is a medium-level pattern
- **Framework** is a collection of classes that provides a set of services for a particular domain, such as Microsoft’s .NET Framework,
Strategic and Tactical Decisions

- During analysis, we must consider
  - What is the desired behavior of the system?
  - What are the roles and responsibilities of the objects that carry out this behavior?
- During design, we must consider
  - What classes exist, and how are these classes related?
  - What mechanisms are used to regulate how objects collaborate?
  - Where should each class and object be declared?
  - To what processor should a process be allocated, and for a given processor, how should its multiple processes be scheduled?

- Strategic decisions have sweeping architectural implications; tactical decisions have localized implications.
The identification of classes and objects is a fundamental issue in object-oriented analysis and design; identification involves both discovery and invention.

Classification is fundamentally a problem of clustering.

Classification is an incremental and iterative process, made difficult because a given set of objects may be classified in many equally proper ways.

The three approaches to classification include classical categorization (classification by properties), conceptual clustering (classification by concepts), and prototype theory (classification by association with a prototype).
Scenarios are a powerful tool of object-oriented analysis and can be used in approaches such as classical analysis, behavior analysis, domain analysis, and use case analysis.

Key abstractions reflect the vocabulary of the problem domain and may either be discovered from the problem domain or invented as part of the design.

Mechanisms denote strategic design decisions regarding the collaborative activity of many different kinds of objects.