CLASSES AND OBJECTS

- What is an Object?
- What is Not an Object?
- Kinds of Software Objects
- Dissecting the Object
- Relationships Among Objects
- What is a Class?
- Relationships Among Classes
- Relationships Between Classes and Objects
- Roles of Classes and Objects in OOD
- Building Quality Abstractions

WHAT IS AN OBJECT?

Object Concept **- objects have a permanence and identity apart from any operation upon them**

Informal definition of an object from the perspective of human cognition:

Object **- any of the following:**

- **a tangible and/or visible thing**
- **something that may be apprehended intellectually**
- **something toward which thought or action is directed**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 76

Formal definition of an object from the perspective of OOD:

Object **- an entity which 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**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 77

WHAT IS NOT AN OBJECT?

- Attributes, such as time, beauty, or color
- Emotions, such as love or anger
- Entities which are normally objects but are, instead, thought of as attributes of objects when a particular problem space is considered

Temperature

```
Oven_Temp : TEMPERATURE
   := 350.0;  -- degrees F
```

Temperature as an object

Temperature Sensor

```
type SENSOR is record
   Temp : TEMPERATURE;
   Redundancy : MULTIPLEX;
   Location: MEMORY_ADDRESS;
end record;
Oven_Temp : SENSOR := (
   Temp => 350.0,  -- degrees F
   Redundancy => TRIPLEX,
   Location => 16#1a0#  );
```

Temperature as an attribute of an object

KINDS OF SOFTWARE OBJECTS

- **Real-world, tangible objects with boundaries that may or may not be clearly defined**
- **Inventions of the design process which collaborate with other objects to provide some higher-level behavior**
- **Intangible events or processes with well-defined conceptual boundaries**

Tangible	**Clearly-defined boundaries** **No clearly-defined boundaries**
Intangible	**Clearly-defined boundaries** **No clearly-defined boundaries** **Events or processes** **Inventions of the design process**

DISSECTING THE OBJECT

Formal definition of an object from the perspective of OOD:

> ***Object*** **- an entity which 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**
>
> -- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 77

This definition of an object refers to three key features:

- **State**
- **Behavior**
- **Identity**

These key features will be discussed in detail.

State

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**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 78

Property or attribute of an object **- a part of the state of the object which is an inherent or distinctive characteristic, trait, quality, or feature that contributes to making an object uniquely that object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 78

All properties have some value:

- **a scalar quantity**
- **a vector quantity or an object**

Because every object has state, every object takes up some amount of space, be it physical space or computer memory.

State of an Object - Example

Temperature Sensor

```
type TEMPERATURE_SENSOR is record
   Temp : TEMPERATURE;  -- degrees F
   Redundancy : MULTIPLEX;
   Location: MEMORY_ADDRESS;
end record;
Oven_Temp : TEMPERATURE_SENSOR := (
   Temp => 350.0,  -- degrees F
   Redundancy => TRIPLEX,
   Location => 16#1a0#  );
```

Objects of class TEMPERATURE_SENSOR, such as Oven_Temp, have three attributes:

- ***Temp*, a dynamic attribute which changes with time**
- ***Redundancy*, a static attribute (the number of sensed points) which is fixed when the object is created**
- ***Location*, a static attribute which is fixed when the object is created**

Behavior

Behavior of an object **- how an object acts and reacts, in terms of its state changes and message passing**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

Operation **-- some action that one object performs upon another in order to elicit a reaction**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

The terms *operation* and *message* are interchangeable.

Method **-- operation that a client may perform upon an object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

The terms *method* and *member function* are interchangeable.

Behavior of an Object - Example

```
package Temperature_Sensor is

  type STATUS is (NOT_OK, OK);
  type TEMPERATURE is FLOAT range -400.0 .. 3_000.0; -- deg F
  type MULTIPLEX is (SIMPLEX, DUPLEX, TRIPLEX);
  type MEMORY_ADDRESS is INTEGER range 0 .. 1_024;

  type OBJECT is record
    Temp       : TEMPERATURE;
    Redundancy : MULTIPLEX;
    Location   : MEMORY_ADDRESS;
  end record;

  function Current_Temperature (Item : in OBJECT)
    return TEMPERATURE;

  function Reliability (Item : in OBJECT)
    return STATUS;

end Temperature_Sensor;
```

Behavior - Kinds of Operations

- ***Modifier*** **- an operation that alters the state of an object, such as a get_with_update or put operation**
- ***Selector*** **- an operation that accesses the state of an object, but does not alter the state, such as a get operation**
- ***Iterator*** **- an operation that permits all parts of an object to be accessed in some well-defined order, such as movement through a linked list**
- ***Constructor*** **- an operation that creates and object and/or initializes its state**
- ***Destructor*** **- an operation that frees the state of an object and/or destorys the object itself**

Behavior - The Protocol of an Object

Protocol **- all of the methods and free subprograms [procedures or functions that serve as nonprimitive operations upon an object or objects of the same or different classes] associated with a particular object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Pp 82-83

The protocol of an object defines the envelope of that object's allowable behavior, comprising the entire external view of the object (both static and dynamic).

Behavior - Objects as Machines

Since an object has state, the order in which operations are invoked is important. This gives rise to the view of an object as an *independent machine*. For some objects, time ordering of their operations is so important that the object's behavior can be formally characterized in terms of a *finite state machine*.

Objects may be either active or passive:

- ***Active Object*** **- an object that encompasses its own thread of control**
- ***Passive Object*** **- an object that does not encompass its thread of control**

Active objects are autonomous, exhibiting a behavior without being operated upon by another object.

Passive objects can only undergo a state change when explicitly acted upon.

Identity

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

-- Khoshafian and Copeland, "Object Identity," *SIGPLAN Notices*, Volume 21, Issues 11, November 1986, Page 406

The failure to distinguish between the name of an object and the object itself is the source of many errors in object-oriented programming.

Lifetime of an Object **- the time span extending from the time an object is first created (and consumes space) until that space is reclaimed**

Note that an object can continue to exist even if all references to it are lost.

Identity - Object Assignment

Object Assignment **differs from copying in that in object assignment, the identity of an object is duplicated by assignment to a second name. Two names then refer to the same object.**

Conventional Assignment **refers to the act of copying the state information of one object into another object. The state of two objects is now the same, but the state of one object may be changed without affecting the other.**

Identity - Equality

Like assignment, *Equality* can have two meanings:

- **two names are equal if they designate the same object**
- **two names are equal if they designate different objects but their state is identical**

RELATIONSHIPS AMONG OBJECTS

An object of and by itself is usually uninteresting. However, a system of objects, wherein the objects collaborate with one another to define the behavior of the system, is intensely interesting.

Two kinds of object hierarchies are extensively employed in OOD:

- **Using relationships, where one object employs the resources of another**
- **Containing relationships, where one object contains one or more other objects**

Using Relationships

Given a collection of objects involved in using relationships, each object may play one of three roles:

- *Actor* - an object that can operate upon other objects but that is never operated upon by other objects; an *active object*
- *Server* - an object that never operates upon other objects but is only operated upon by other objects; a *passive object*
- *Agent* - an object that can both operate upon other objects and be operated upon by other objects; an agent is usually created to do some work on behalf of an actor or another agent

Whenever one object passes a message to another with which it has a using relationship, the two objects must be *synchronized*. In a single thread of control, a subprogram call is adequate for synchronization. With multiple threads of control, a more complex method of synchronization must be devised in order to deal with the problems of mutual exclusion.

Using Relationships, Continued

The need for synchronization in an environment involving multiple threads of control leads to another way to classify kinds of objects:

- *Sequential object* - a passive object whose semantics are guaranteed only in the presence of a single thread of control
- *Blocking object* - a passive object whose semantics are guaranteed in the presence of multiple threads of control
- *Concurrent object* - an active object whose semantics are guaranteed in the presence of multiple threads of control

Containing Relationships

In a *containing relationship*, an object may encapsulate one or more other objects. Some real-world object relationships are clearly containing relationships, such as the automobile engine which contains pistons, spark plugs, etc.

Containing an object rather than using an object is sometimes better because containing reduces the number of objects that must be visible at the level of the enclosing object.

Using an object is sometimes better than containing an object because containing an object leads to undesirable tighter coupling among objects in some cases.

Intelligent engineering decisions require careful weighing of these two factors.

WHAT IS A CLASS?

Class **- a set of objects that share a common structure and a common behavior**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 93

A class represents only an *abstraction*, whereas an object, *an instance of a class*, is a concrete entity that exists in time and space.

What is NOT a Class?

An object is not a class**, but a class may be an object (to be discussed later).**

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.

The Class as a Contractual Binding

The class captures the structure and behavior common to all related objects, serving as a binding contract between an abstraction and all of its clients.

Strongly typed programming languages can detect violations of the contract that is a class during compilation.

Two views of a class:

- ***Interface*** **- the outside view of a class, emphasizing the abstraction while hiding the structure and details of how its behavior works**
- ***Implementation*** **- the inside view of a class, which details the internal structure of a class and the details of how its behavior works**

The Interface to a Class

The interface to a class consists of:

- **primarily, the declarations of all operations applicable to instances of the class; these operations may be invoked by clients of the class objects**
- **the declaration of other classes**
- **constants**
- **variables**
- **exceptions**

The last four are included if they are needed to complete the abstraction.

The Interface to a Class, Continued

The interface to a class can be divided into three parts:

- ***Public*** **- a declaration that is visible to all clients of the objects of a class**
- ***Protected*** **- a declaration that is not visible to any other classes except the subclasses of the class**
- ***Private*** **- a declaration that is not visible to any other classes**

C++ does the best job in allowing a developer to make explicit distinctions among these different parts of a class interface. Ada permits declarations to be public or private, but not protected.

The State of an Object

The state of an object is usually represented as constant and variable declarations placed in the private part of a class interface. This encapsulates the representation common to the objects of a class, and changes to this representation do not have a functional affect on the clients.

Why is the State of an Object NOT in the Implementation?

Placing state information in the implementation of a class would completely hide it from the clients, but, with today's technology, placing state information in the implementation rather than the private interface of a class would require either object-oriented hardware or very sophisticated compiler technology. Compiler technology can solve this problem, but the compiler must be able to discern information about the size of the object of the class.

RELATIONSHIPS AMONG CLASSES

Three basic kinds of class relationships:

- ***generalization*** **- a "kind of" relationship, as a sailboat is a kind of ship**
- ***aggregation*** **- a "part of "relationship, as a hull is a part of a ship**
- ***association*** **- a semantic connection among otherwise unrelated classes, as roses and candles representing different classes that share in common the fact that we might use them to decorate a dinner table**

Object-based and object-oriented programming languages support some combination of the following relationships to realize the basic kinds of class relationships:

- ***inheritance*** **relationships, which are perhaps the most powerful, supporting generalization and association**
- ***using*** **relationships, supporting aggregation**
- ***instantiation*** **relationships, supporting generalization and association in a different way from inheritance**
- ***metaclass*** **relationships, supporting the notion of a class of a class (classes as objects are made possible)**

Inheritance Relationships

Inheritance **- a "kind of" relationship among classes wherein one class shares the structure or behavior defined in:**

- **one other class (*single inheritance*)**
- **more than one other class (*multiple inheritance*)**

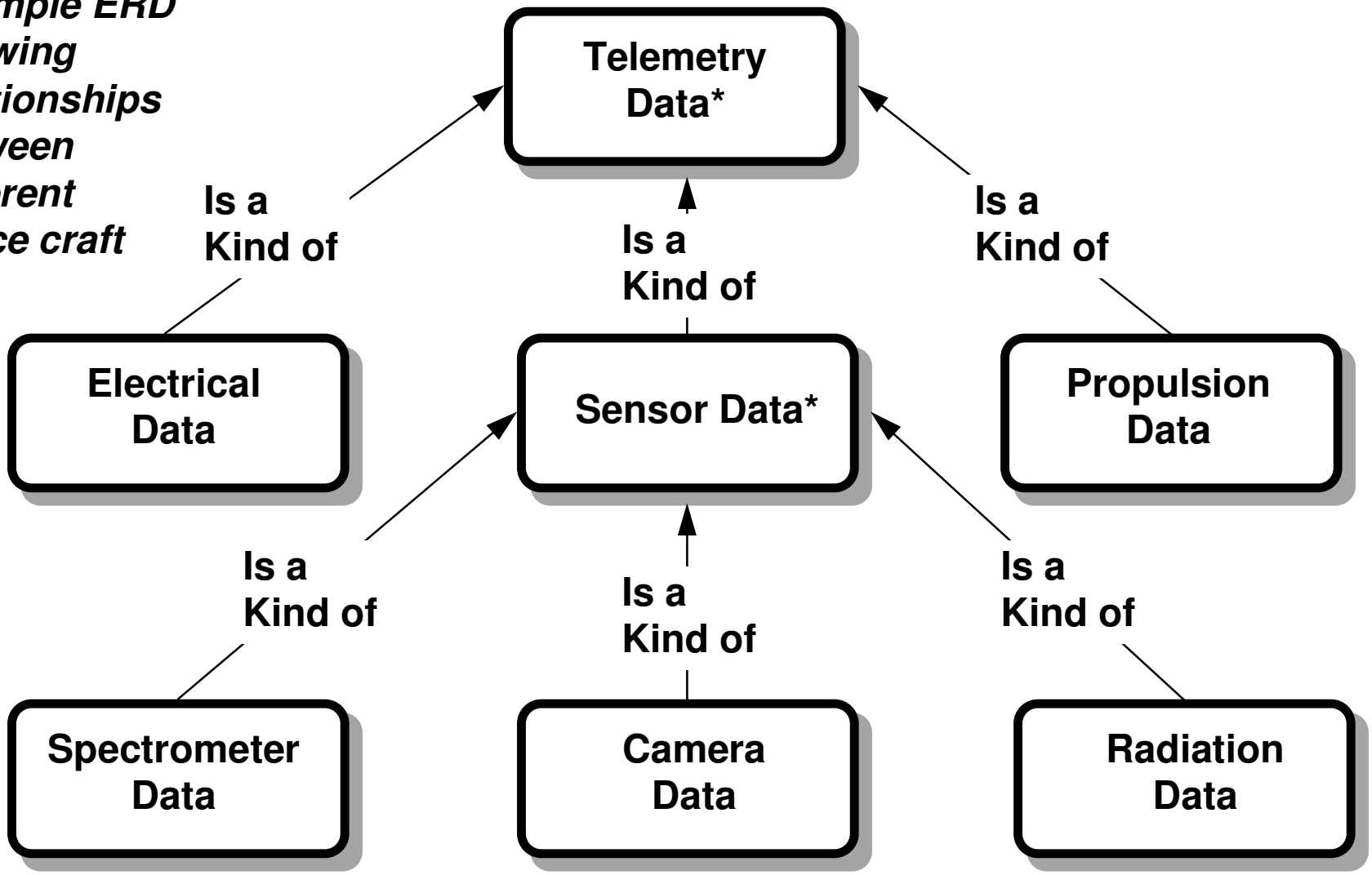
Superclass **- the class from which another class inherits structures and/or behaviors; a *Base Class* is the most generalized superclass**

Subclass **- a class that inherits from one or more other classes, typically augmenting or redefining the existing structures and behaviors of its superclasses in itself without affecting the superclasses**

The ability of a programming language to support this kind of inheritance distinguishes *object-oriented languages* (which support inheritance) from *object-based languages* (which do not support inheritance).

Single Inheritance Relationships

A simple ERD showing relationships between different space craft data



** Abstract classes, or classes with no instances*

Abstract Classes

Abstract Classes **- classes with no instances, written with the expectation that their subclasses will add to their structures and behaviors, usually by completing the implementations of their incomplete methods**

C++ allows a member function to be defined as a *pure virtual function*, and C++ prohibits the creation of instances of classes which contain pure virtual functions.

Kinds of Clients

A given class typically has two kinds of clients:

- ***Instances***
- ***Subclasses***

This is the motivation behind the three parts of a class definition:

- ***Public part*** **- members are visible to both kinds of clients**
- ***Protected part*** **- members are visible to subclasses only**
- ***Private part*** **- members are invisible to both kinds of clients**

Inheritance and Encapsulation

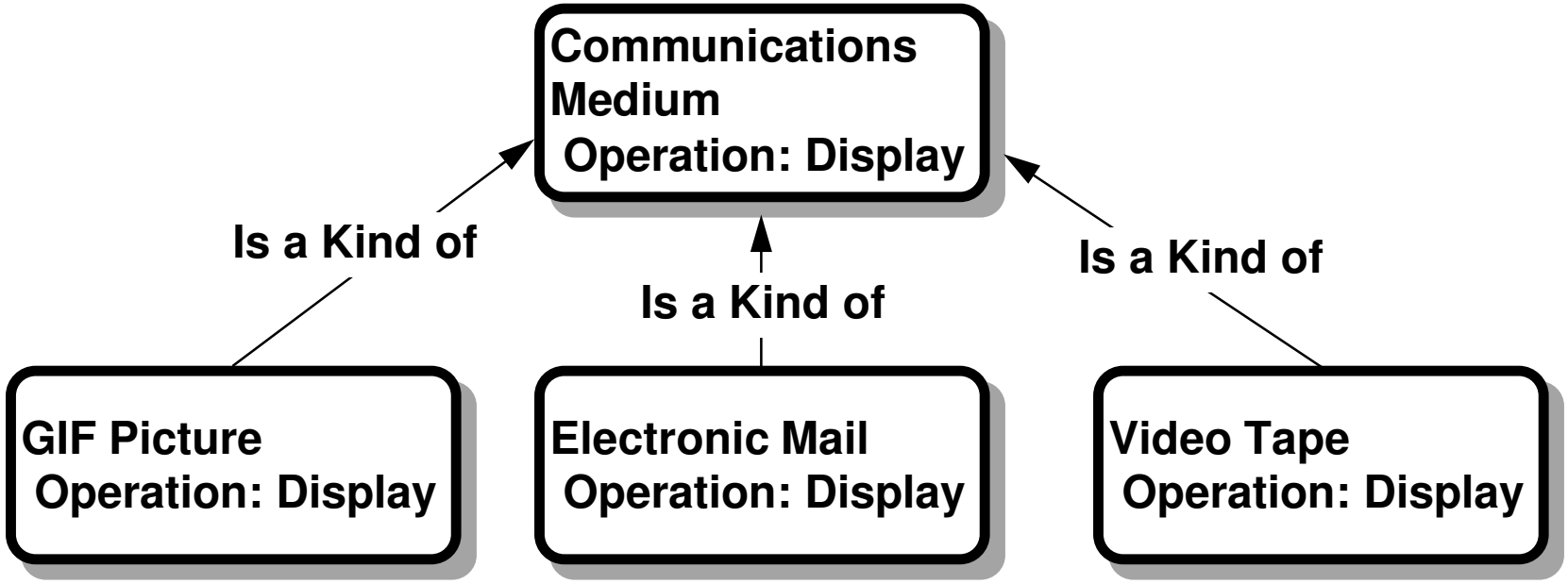
Some tension exists between inheritance and encapsulation in that the use of inheritance exposes some of the internal details of an inherited class.

This means that to understand the meaning of a particular class, you must often study all of its superclasses, sometimes including their inside views.

Polymorphism

Polymorphism **- a concept in type theory in which a name may denote objects of many different classes that are related by some common superclass**

Any polymorphic object may respond to some comon set of operations (defined by the superclass) in different ways.



Overloading

Many languages, such as Ada and C++, allow functions and procedures to have the same name so long as they can be distinguished by their parameters. There many be many functions named GET or "+", for instance, but there is only one GET function which gets an integer and only one "+" function which adds two integers. Such functions and procedures are said to be *overloaded*.

There are two kinds of polymorphism, then:

- ***ad hoc polymorphism*** **- otherwise known as operator or subprogram overloading**
- ***parametric polymorphism*** **- the "class" polymorphism discussed on the previous transparency**

Inheritance without polymorphism is possible, but this kind of inheritance is not useful.

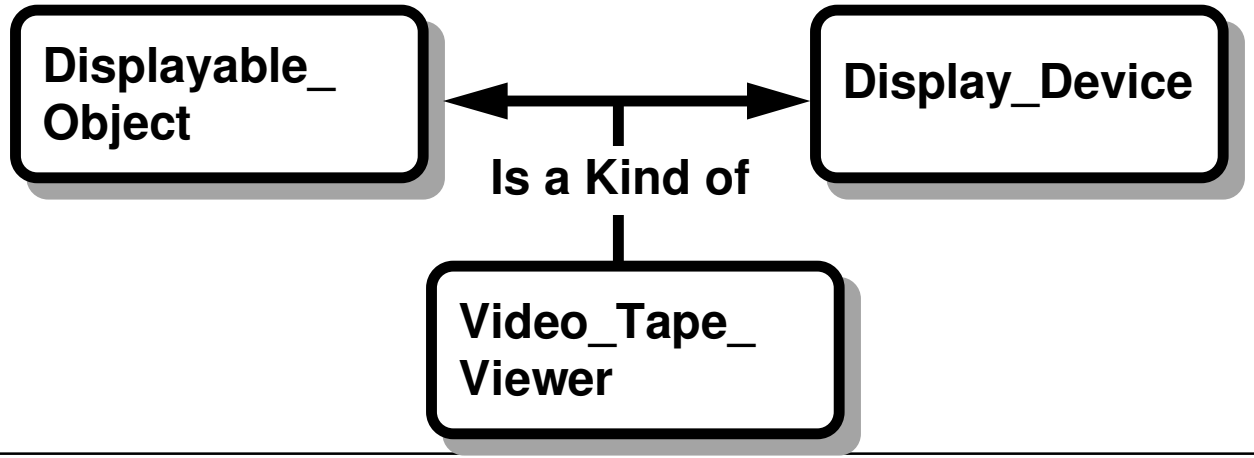
Polymorphism and late binding go hand in hand.

Multiple Inheritance

Multiple inheritance comes into play when a class inherits from more than one superclass. The need for multiple inheritance in object-oriented programming languages is still a topic of debate.

Multiple Polymorphism

Multiple polymorphism comes hand in hand with multiple inheritance. In *Multiple Polymorphism*, the polymorphic function depends on two or more parameters associated with two or more superclasses.



Using Relationships

Two kinds of using relationships for classes:

- **a class's interface may use another class, in which case the used class is visible to the clients of the using class**
- **a class's implementation may use another class, in which case the used class is not necessarily visible to the clients of the using class**

Using relationships imply a *cardinality*:

- **a 1:1 relationship**
- **a 1:n relationship, created by establishing *friends*, which are methods involving two or more objects of different classes**
- **a m:n relationship, also created by establishing friends**

Instantiation Relationships

Instantiations **entail the use of templates which are implemented in one class to operate on instances of other classes, such as a linked list class which can create linked lists of integers, floats, strings, files, databases, etc.**

Instantiations are usually realized in the creation of *container classes*, which are classes that contain instances of other classes.

Generic Classes or Parameterized Classes **- serve as templates for other classes, such as the class containing a generic sort serving as a template to sort integers, floats, files, etc.**

Metaclasses

Metaclass **- a class whose instances are themselves classes**

The three kinds of class relationships discussed so far, namely inheritance, using, and instantiation, cover all the important kinds of class relationships that most developers need. The fourth kind of class relationship, the metaclass, is more exotic and still of a theoretical nature.

The metaclass allows a programmer to manipulate a class as an object, but is this of real value?

CLOS supports metaclasses, but Ada and C++ do not directly support them, altho C++ offers the notion of static member data and functions to aid in the support of a metaclass.

RELATIONSHIPS BETWEEN CLASSES AND OBJECTS

- Every object is the instance of some class.
- Every class has zero or more instances.
- Classes are static, so their existence, semantics, and relationships are fixed at compile time.
- Objects are static or dynamic.
- The class of most objects is static, meaning that once and object is created, its class is fixed.
- Objects are created and destroyed often during the lifetime of an application program.

ROLES OF CLASSES AND OBJECTS IN OOD

During OOA and the early stages of OOD, the developer has two primary tasks:

- **Identify the classes and objects that form the vocabulary of the problem domain. These classes and objects are called the *key abstractions* of the problem.**
- **Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem. These structures are called the *mechanisms* of the implementation.**

BUILDING QUALITY ABSTRACTIONS

In order to build a quality object-oriented system of classes and objects, we must be able to do several things:

- **Measure an abstraction to determine its quality**
- **Apply heuristics for choosing the operations**
- **Apply heuristics for choosing the relationships**
- **Apply heuristics for choosing the implementations**

Classes and objects **make up the key abstractions of an object-oriented system, and the framework for such a system is provided by its *mechanisms*.**

Measuring the Quality of an Abstraction

The design of classes and objects is an incremental, iterative process, and quality is seldom achieved on a first attempt.

There are five meaningful metrics in assessing the quality of an abstraction:

- Coupling
- Cohesion
- Sufficiency
- Completeness
- Primitiveness

Measuring Quality, Continued

Coupling **is a measure of the strength of association established by a connection from one module to another in structured design, and it is a measure of the a similar strength between classes and objects in object-oriented design.**

In OOD, however, coupling and inheritance are at odds with each other. Strong coupling complicates a structured system, so weakly-coupled classes are desired. Inheritance, which strongly couples superclasses and subclasses, however, is also desired to exploit the commonality among classes.

Cohesion **is a measure of the degree of connectivity among the elements of a single module in structured design, and it is a measure of a similar strngth among the elements of classes and objects in object-oriented design.**

Coincidental cohesion**, in which unrelated abstractions are thrown into the same class, is undesirable.** ***Functional cohesion*****, in which elements of a class work together to provide some well-rounded behavior, is desirable.**

Measuring Quality, Continued

Classes should be sufficient, complete, and primitive:

- **By *sufficient*, the class captures enough characteristics of the abstraction to permit meaningful and efficient interaction. For example, a linked list class should allow for adding objects from the list, but it should also allow for removing objects from the list to be sufficient.**
- **By *complete*, the class captures all of the meaningful characteristics of the abstraction. Sufficiency implies a minimal interface, where completeness implies one that covers all aspects of the abstraction. Warning: *completeness is a subjective concept and can be overdone*, providing much more functionality than needed for an application.**
- **By *primitive*, the operations associated with a class are those that can be efficiently implemented only if given access to the underlying representation of the abstraction. An operation that could be implemented on top of existing primitive operations, but at the cost of significantly more computational resources, is also a candidate for inclusion as a primitive operation.**

Heuristics for Choosing Operations

- Create *fine-grained methods*, which are primitive operations that exhibit small, well-defined behaviors.
- Separate methods that do not communicate with each other.
- Design the methods of a class as a whole, because all these methods cooperate to form the entire protocol of the abstraction.
- Given a desired behavior, decide in which class to place it based on the following:
  - *Reusability* - Would the behavior be more useful in more than one context?
  - *Complexity* - How difficult is it to implement the behavior?
  - *Applicability* - How relevant is the behavior to the class in which it might be placed?
  - *Implementation Knowledge* - Does the behavior's implementation depend upon the internal details of a class?

Heuristics for Choosing Operations, Continued

Once an operation is established and defined in terms of its functional semantics, its time and space semantics must be determined:

- ***Synchronous*** **- An operation commences only when the sender has initiated the action and the receiver is ready to accept the message. The sender and receiver will wait indefinitely until both parties are ready to proceed.**
- ***Balking*** **- Like synchronous, except that the sender will abandon the operation if the receiver is not immediately ready.**
- ***Timeout*** **- Like synchronous, except that the sender will only wait for a specified amount of time for the receiver to be ready.**
- ***Asynchronous*** **- A sender may initiate an action regardless of whether the receiver is expecting the message.**

Heuristics for Choosing Relationships

- **Choosing the relationships among classes and among objects is linked to the selection of operations, since for one object or class to send a message to the other, the other object or class must be visible to the first.**

  Visibility **- the ability for one abstraction to access the interface of another**

- ***Law of Demeter*** **- The methods of a class should not depend in any way on the structure of another class, except for the immediate (top level) structure of its own class.**

- **Class structures that are wide and shallow usually represent forests of free-standing classes that can be mixed and matched, and such classes are more loosely coupled (which is good) but may not exploit all the commonality that exists (which is bad).**

- **Class structures that are narrow and deep represent trees of classes that are related by a common ancestor, and such classes exploit all the commonality that exists (which is good) while requiring the user to understand the meanings of all classes it inherits from or uses (which is bad).**

Heuristics for Choosing Implementations

- The implementation can only be designed after the interface is completed.
- The implementation of a class or object should almost always be encapsulated in the abstraction, making it possible to change the implementation without violating the interface to the clients.
- Implementations should be optimized for operation based on the most frequent expected use of the abstraction.
- Examine time versus space constraints to determine how best to implement the object's state information, particularly when it comes to the tradeoff of storing state information in the object or computing it when needed.
- Seek to build functionally cohesive, loosely coupled modules, so trade off the visibility of abstractions and the concept of information hiding against cohesion and coupling.
- Always consider the possiblities of reuse, security, and documentation.