Classes and Objects

Yi-ting Chiang

Department of Applied Mathematics
Chung Yuan Christian University

September 26, 2023

Classes and Objects

Classes and objects are the basic building blocks when using object-oriented methods to analyze or design a complex software system. This topic discusses details about classes and objects in object-oriented design and analysis.

Objects can represents not only tangible (有形的) things (a car or a plant) but also intangible (無形的) events, ideas, or processes (the procedure to apply to college).

A possible definition for object is:

Object(物件)

An object is an entity that has state, behaviour, and identity. The structure and behaviour of similar objects are defined in their common class.

The following slides discuss the state, behaviour, and identity of an object

State

An object can be in different states. For example, a vending machine can have the following state:

- waiting for money
- waiting for selection
- dispense the item
- return change

The user puts enough money into the machine and then selects the item he/she wants. If the money is correct, the vending machine will dispense the selected item and return the change. If the user selects an item before/without putting in enough money, he/she cannot get the item.

State

A vending machine has some important properties. For example, the vending machine can have change to return to the user. In addition, it has item to sell. The amount of change available and the quantity of the item remained represent the current state of the vending machine.

As the example above, the state of an object can be define as follows:

State(狀態)

The state of an object encompasses all of the (usually static) properties of the objects plus the current (usually dynamic) values of each of these properties.

State

So what is "property"? A property is an inherent or distinctive characteristic, quality, or feature that make an object unique.

About properties:

- Properties are usually statics[1]
- All properties have some values which can be
  - numerical: a quantity (the amount of the items available)
  - categorical: denote another object (a collection of drinks it can sell)

As the vending machine example, there are items (another object) and the number of the items available to sell (a quantity)

[1] For example, an elevator can only travel up and down, not horizontally.

State

Here is another example:

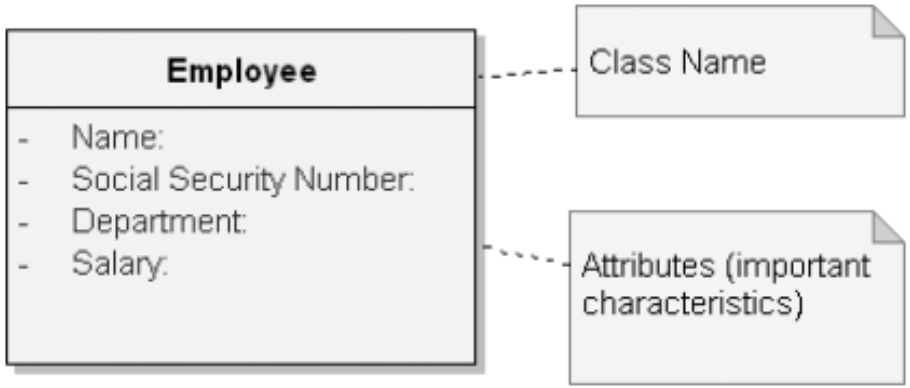


Figure: The Employee class with attributes (properties).[2]

This figure gives the abstraction of an employee record. The name, social security number, department, and the salary are important characteristics of an employee. Notice that the abstraction is not an object[3], but we can use the abstraction to define objects (refer to a specific employee)

[2] From: Object-Oriented Analysis and Design with Applications, 3rd, 2007

[3] because it does not represent a specific instance.

State

Here we give two distinct objects by the abstraction of employee:

Employee

- Name: Tom
- ID: 25xxx
- Department: Marketing
- Salary: 45k

Employee

- Name: Cathy
- ID: 17xxx
- Department: Finance
- Salary: 50k

The objects will keep some amount of space in the memory. These two objects have the same properties but with different values. They do not share their spaces with any other objects.

Behaviour

Objects in a complex system collaborate to perform tasks. That is, they act on other objects (as server) while being acted on (as client) other objects. The result of an object's action depends on the input (message) sent to the object and the internal state of the object. This defines the behaviour of an object:

Behaviour(行為)

Behaviour is how an object acts and reacts, in terms of its state changes and message passing.

That is, the behaviour of an object is a function of its state as well as the operation[4] performed on it.

[4]The action that an object performs on another to elicit a reaction is called "operation". In Java, operations are called "method". In C++, they are "member functions".

Behaviour

For example, consider the vending machine, the user can invoke some operation on the vending machine to make a selection:

1. If the change the user deposit is not sufficient for any item, the machine will do nothing (the state is not changed).
2. If the change is sufficient for buying any item, machine may be ready to wait for the user to select an item (the state is changed)

As the second case shown above, the operation causes the side effect of altering the object's state. Therefore, we can redefine the state of an object as:

State

The state of an object represents the cumulative results of its behaviour.

Behaviour: Operations

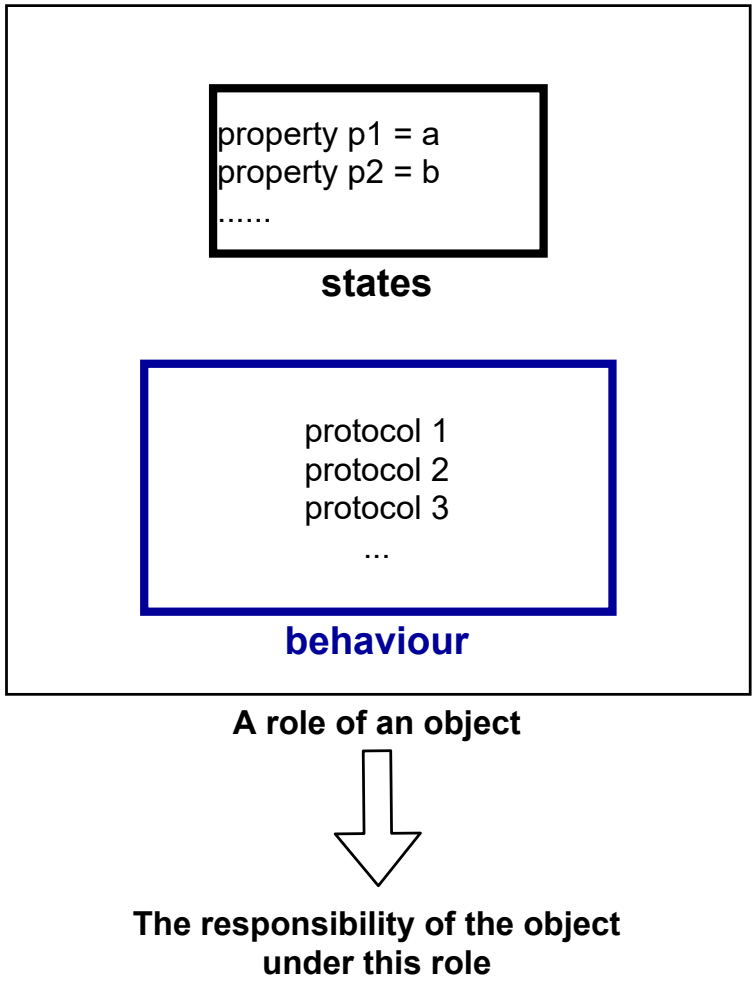
An operation denotes a service to the clients. Typically, there are five kinds of operations:

1. Modifier: an operation that alters the state of an object
2. Selector: an operation that accesses the state of an object but does not alter the state
3. Iterator: an operation that permits all parts of an object to be accessed in some well-known order
4. Constructor: an operation that creates an object and/or initializes its state
5. Destructor: an operator that frees the state of an object and/or destroys the object itself

Not all operations are available in the programming languages. For example, Java has constructor but does not have destructor.

Behaviour: Roles and Responsibilities

All the methods of an object comprise its protocol, which defines an object's behaviour. The states and behaviours of an object define the roles it can play, which represent its responsibilities.



Behaviour: Roles and Responsibilities

One object can have many roles. The roles of an object

- define the contracts between an abstraction and its clients
- are the object's logical groupings of its behaviour

The roles between objects can overlap or be mutually exclusive, and can be invariable or be changed. Different roles usually have different responsibilities.[5]

When analysing a problem, one should examining the various roles that an object plays. During design, refine these roles by defining the operations that carry out each role's responsibilities.

[5] For example, a person can plays different roles such as a passenger, a student, a customer, or a son/daughter.

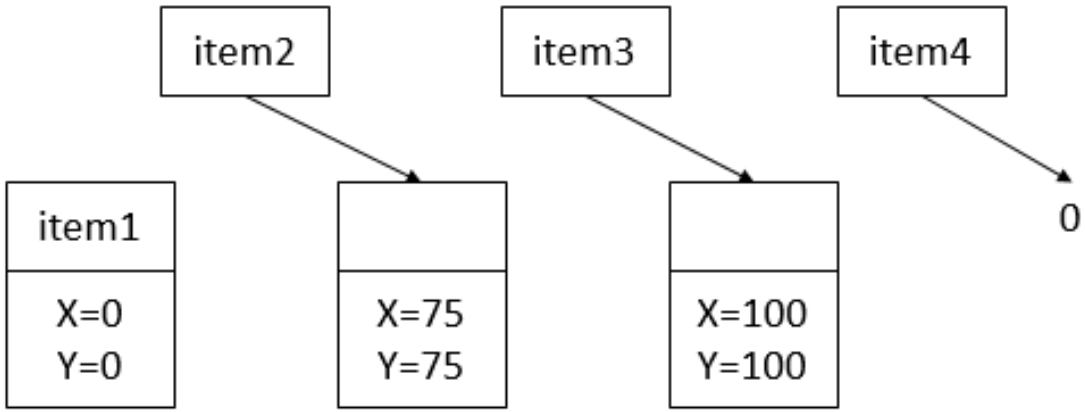
Identity

Identity(識別)

Identity is the property of an object that distinguishes it from all other objects. The unique identity of each object is preserved over the lifetime of the object, even when its state is changed.

Identity

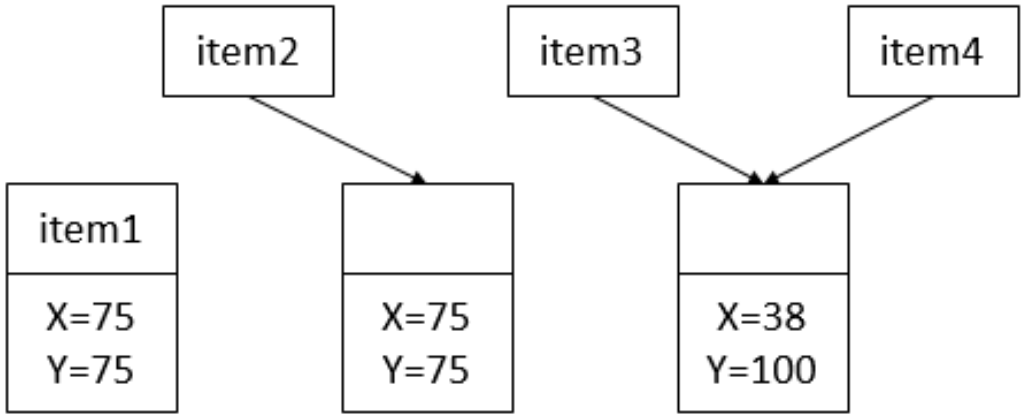
The identity of an object is not necessary its name, and can be irrecoverably lost. See the following example[6]:



In this figure, item1 is the name to distinguish it from other objects, but item2, item3, and item4 are pointers. In addition, item4 points to a null object.

[6] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Identity

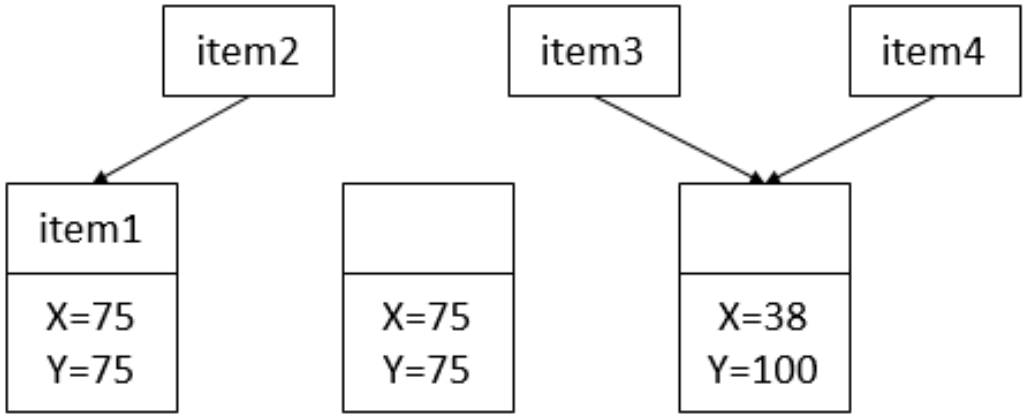


Changing the value of item1 and the objected pointed by item4 results in the figure above. Note that:

- item1 and the object that pointed by item2 have the same X and Y value, but they are different objects
- item3 and item4 point to the same object

The situation of item3 and item4 is called structural sharing. If you use item3 to destroy the object, item4 will point to a meaningless location. This is called dangling reference.

Identity



In this example[7], item2 and item1 point to the same object. Notice that there is an object with no identity. In some programming language, the memory will be reclaimed automatically. However, in C++ (using traditional C++ pointer), this causes memory leak, which can be fatal in some applications.

[7] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Class

Object is a concrete entity that exists in time and space; class is only an abstraction. An object is NOT a class. It's an instance of a class.
In object-oriented analysis and design, a class can be defined as follows:

Class(類別)

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

Sometimes abstractions are too complex to be expressed as a single class.

For example, a GUI(graphical user interface) framework and a database are conceptually individually objects but are composed of a cluster of classes whose instances collaborate to provide the desired structure and behaviour.

Class: Interface and Implementation

A class captures the structure(inside view) and behaviour(outside view) common to all related objects.

The interface(介面) of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and the secrets of its behaviour. It consists of the declaration of

- all operations applicable to instances of this class
- other classes, constants, variables, and exceptions necessary to complete the abstraction

By contrast, the implementation(實作) of a class consists of how all the operations defined in the interface of the class are actually implemented. It is the inside view of a class, which encompasses the secret of the behaviour of that class.

Class: Interface and Implementation

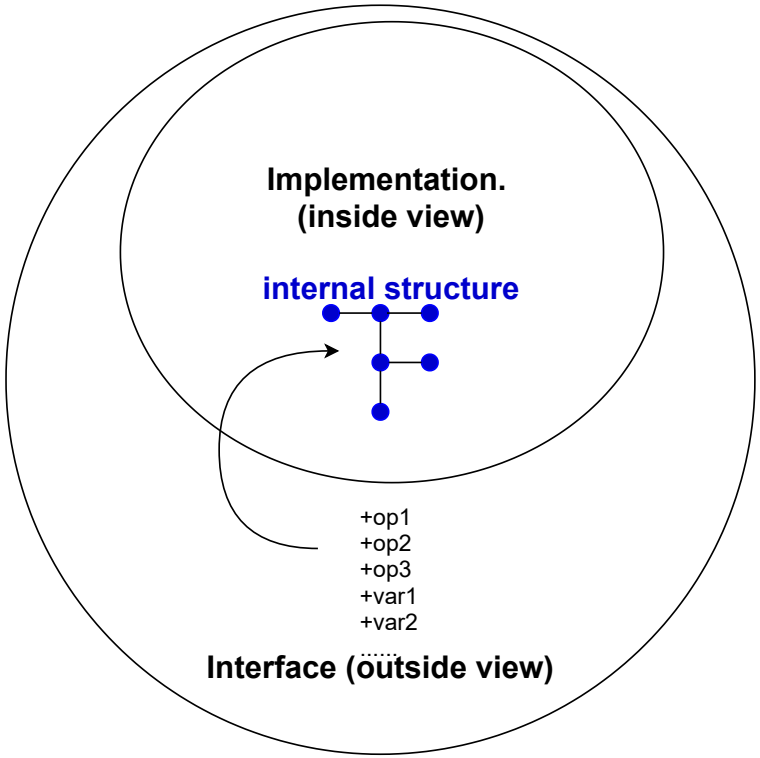


Figure: The concept of implementation and interface of a class.

The implementation includes the structure and how the details of the behaviour are implemented. The internal structure can be known and accessed by the operations defined in the interface, but can not be directly accessed from the outside.

Class

The interface of a class can be divided into

- 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

To represent the state of an object, we usually declare constants and variables which are placed in the protected or privated part of a class's interface. In this manner, the representation is encapsulated, and changes to this representation do not affect any outside clients.

Different object-oriented programming languages may support different mixtures of public, protected, private, and package parts. For example, Java supports all the four parts, and Ada does not support protected.

Relationships among Classes

Relationships between classes may indicate

- some sort of sharing
- some kind of semantic connection

Consider the following example:

- A daisy(雛菊) is a kind of flower
- A rose is a kind of flower
- A petal(花瓣) is a part of flowers
- Red roses and yellow roses are both kinds of roses
- Ladybugs eat certain pests(害蟲) which may infest certain kinds of flowers

Daisy and rose share properties. For example, they both have petals and emit fragrance. For semantic connections, red roses and yellow roses are more alike than are daisies and roses. In addition, ladybugs protect flowers by eating pests, so flowers server as a food source for the ladybug.

Relationships among Classes

The basic kinds of class relationships are

- association
- inheritance: "is a" relationship
- aggregation: "part of" relationship

Association indicates some semantic dependency among otherwise unrelated classes. For example, ladybugs is not a kind of flowers and is not part of any flower, but they are related as mentioned in the previous slide.

The concept of inheritance and aggregation have being introduced in previous topics. We will give the illustration of them in the view point of classes.

Association

Association is the most general but semantically weak relationship of classes. Specifically,

- Association is used to capture the participants in a semantic relationship, the roles they play, and their cardinality (數量)
- Association does not state the direction of the dependency
- Association does not state exact way in which one class relates to another

Association is usually used in the early stage of system analysis and design, and is refined by turning it into more concrete class relationship.

Association

The figure below gives an association relationship between two classes:

Figure: Association.[8]

This example shows the association relationship between wheels and vehicles. The asterisk($*$) symbol and the value 1 under the line gives the kind of multiplicity of the association relationship. There are three common kinds of multiplicity across an association:

- one-to-one: for example, one sale has one corresponding credit card transaction
- one-to-many: for example, one vehicle has many wheels
- many-to-many: for example, a salesperson can interact with many customers, and each customer can also interact with many salespeople

[8] Figure from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance

Inheritance expresses the generalization/specialization relationships. It is a relationship among classes wherein one class shares the structure and/or behaviour defined in one or more other classes:

- Single Inheritance: A class inherits from one class
- Multiple Inheritance: A class inherits from more than one class

The class from which another class inherits is superclass. A class that inherits from other class(es) is subclass. Inheritance defines an "is a" hierarchy among classes.

Inheritance: Single Inheritance

Using space probes(太空探測器) as an example. Space probes send information regarding the status of important subsystems and different sensors to ground stations. These information is called telemetry data. ElectricData is a kind of telemetry data. See the following figure:

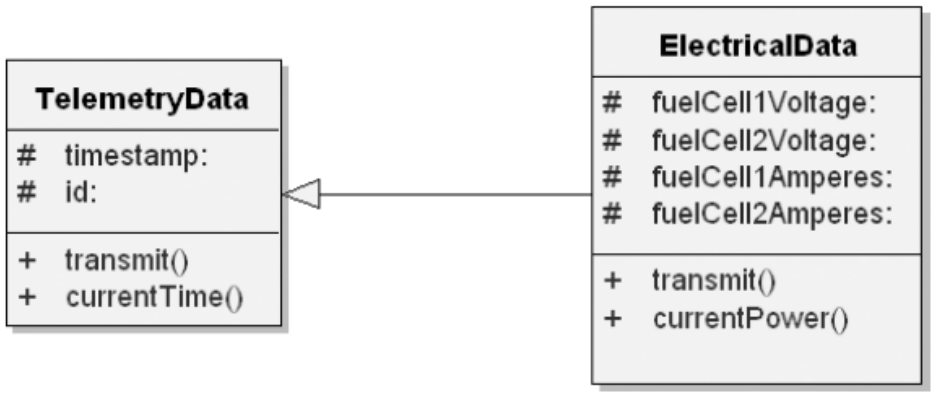


Figure: ElectricalData inherits from TelemetryData.[9]

Electrical data is a special kind of telemetry data. Therefore, as the figure shows, TelemetryData is the superclass of ElectricalData, which inherits the structure and behaviour of TelemetryData but adds its structure (voltage and ampere data), refines the behaviour (define its own transmit function), and adds new behaviour (the currentPower function).

[9] Figure from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance: Single Inheritance

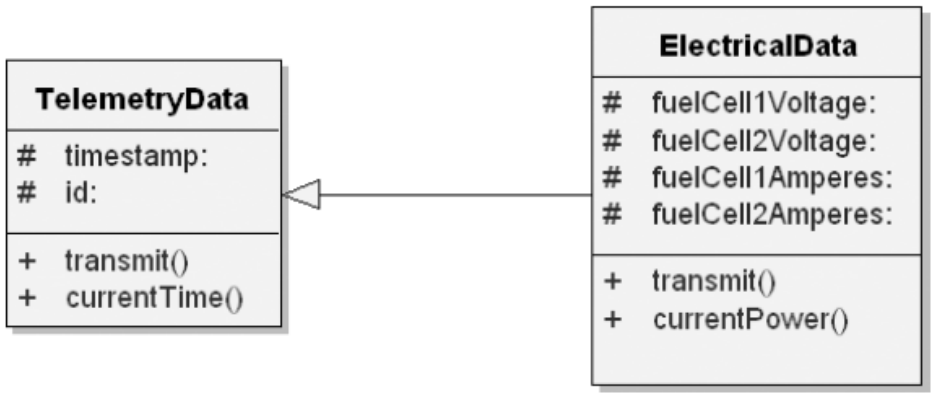
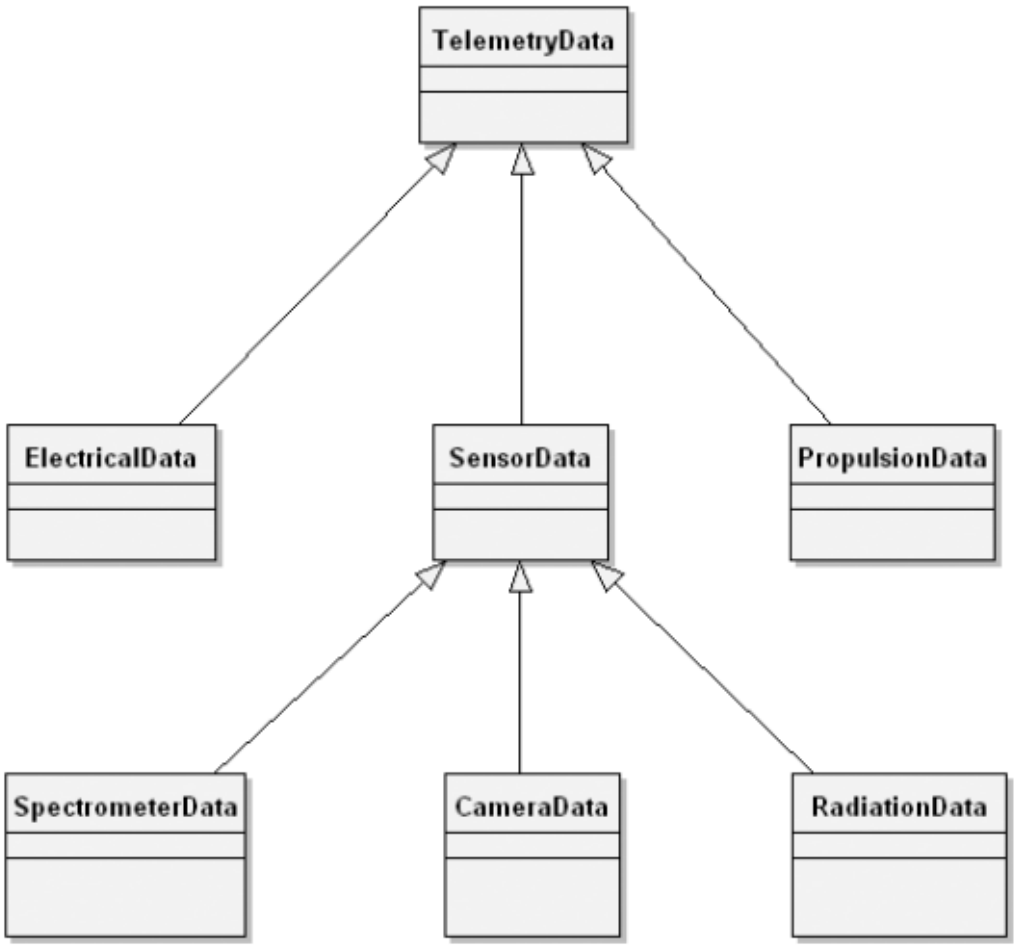


Figure: ElectricalData inherits from TelemetryData.[10]

This figure shows an example of single inheritance. A subclass can enlarge or restrict the structure and behaviour of its superclasses. In this example, ElectricalData defines new properties and provides a new operation. It augment its superclass.

[10] Figure from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance: Single Inheritance



11

Some classes may be used only to be the superclass of other classes. This kind of class is called abstract classes. See the figure in the left.

Generally speaking, more specialized classes (leaf classes or concrete classes) such as ElectricalData and CameraData are expected to have instances. On the other hand, SensorData and TelemetryData are not likely to have any instances. We only use them to define abstractions. They are abstract classes.

[11] Figure from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance: Single Inheritance

Polymorphism (多形) is the concept wherein a name may denote instances of many different classes as long as they are related by some common superclass[12].

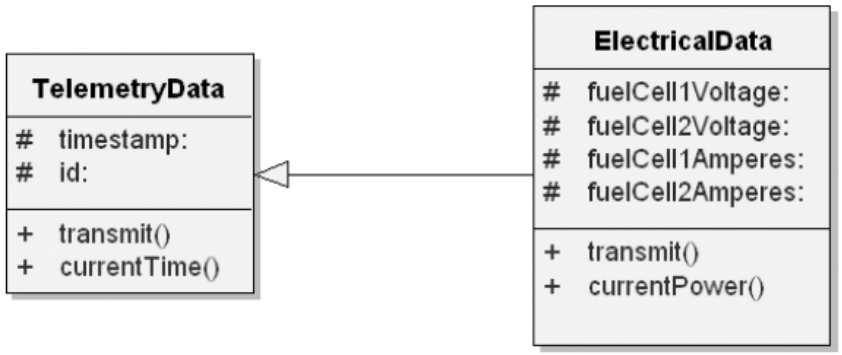


Figure: ElectricalData refines the function transmit() provided by TelemetryData.[13]

In this figure, transmit() in Telemetry is a general function to transmit telemetry data. ElectricalData can define its own transmit() to transmit additional data. The common superclass is Telemetry.

[12] "Polymorphism" can also be used to illustrate that a function can deal with parameters of different data types. For example, add(x,y), where x,y can be integer, float, or double. This is called "ad hoc polymorphism" or "overloading".

[13] Figure from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance: Single Inheritance

```
#include<cstdio>
#include<cstring>

class Date {
    private:
      char date[256];
    public:
      Date(char* d) { strcpy(date,d); }
      void showWhen() { printf("Date: %s\n",date); }
};

class DateTime: public Date {
    private:
      char time[256];
    public:
      DateTime(char* s) : Date(strtok(s," ")) {
          strcpy(time,strtok(NULL," "));
      }
      void showWhen() {
          Date::showWhen(); // call superclass
          printf("Time: %s\n",time);
      }
};
```

datetime.h

The code above gives an example of polymorphism in C++. Class Date is the superclass of DateTime. Subclass DateTime redefines a function showWhen(). By this, the function can deal with a string contains information of date and time.

Inheritance: Single Inheritance

```
#include "datetime.h"
int main() {
    char s[256]="2020-03-10 09:20:30";
    DateTime dt=DateTime(s);
    dt.showWhen();
    return 0;
}
```

The output of the code will be:

```
Date: 2020-03-10
Time: 09:20:30
```

Inheritance: Multiple Inheritance

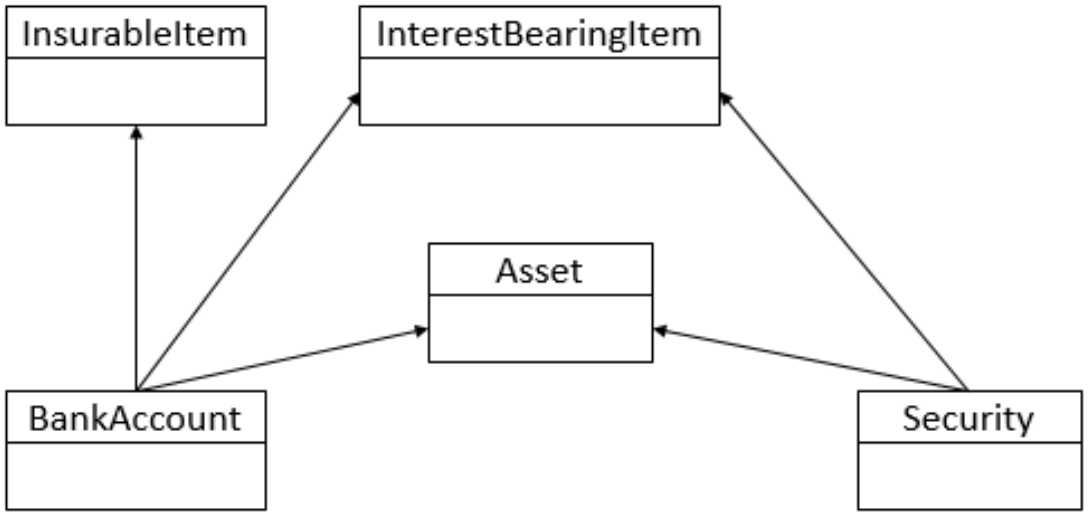
In single inheritance, a class sometimes needs to choose which class to inherit. Consider the following example:

- A security(證卷) is a kind of asset(資產), and it can bear interest(產生利息)
- A bank account is a kind of asset. It can bear interests and is insurable

If there are classes Asset, InterestBearingItem, and InsurableItem. To define a new class Security, one has to choose Asset or InterestBearingItem to be the superclass; to define BankAccount, one has to choose from three classes. If Asset is chosen, the required members (variables or operations) in InterestBearingItem and InsurableItem have to be defined again in Asset.

By using multiple inheritance, a subclass can have more than one superclasses to avoid redundancy.

Inheritance: Multiple Inheritance



[14]

We can use multiple inheritance to define a style of classes called mixins. A class defined as a mixin provides only operations to augment the behaviour of other classes. Generating instances of a mixin is usually meaningless.

In the figure above, only defining operations related with insurance and interest in InstrableItem and InterestBearingItem respectively results in two mixins.

[14] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Inheritance: Multiple Inheritance

Multiple inheritance can cause two issues:

- Clashes: name collisions from different superclasses
- Repeated inheritance: peer superclasses share common superclasses

Clashes may introduce ambiguity in the behaviour of the subclass. For example, both InsurableItem and Asset have an attribute presentValue(現值). Which one is BankAccount's presentValue?

Three possible approaches can resolve clashes:

- Compilation error
- Referring to the same attribute
- Requiring all references explicitly give the name of the superclass (use fully qualified names)
- Defining the order of the superclasses to access

Inheritance: Multiple Inheritance

The approaches to dealing with repeated inheritance are similar to those that dealing with clashes:

- Treat repeated inheritance as illegal
- Treat multiple references to the same class as denoting the same class
- Use fully qualified names
- Define the order of the superclasses to access

Inheritance: Multiple Inheritance

The following example shows how python resolves clashes[15]:

```
class A:
  def hello(self):
    print("A says hello!")

class SubA(A): #Subclass of A
  pass

class B:
  def hello(self):
    print("B says hello!")

class C(B,SubA):
  pass

c=C()
c.hello()
```

The output is "B says hello"

```
class A:
  def hello(self):
    print("A says hello!")

class SubA(A): #Subclass of A
  pass

class B:
  def hello(self):
    print("B says hello!")

class C(SubA,B):
  pass

c=C()
c.hello()
```

The output is "A says hello"

[15] Python decides the order of accessing the superclasses

Inheritance: Multiple Inheritance

The following example shows how python resolves repeated inheritance:

```
class A:
  def hello(self):
    print("A says hello!")

class SubA1(A): # Subclass of A
  pass

class SubA2(A): # SubClass of A
  def hello(self):
    print("SubA2 says hello!")

class C(SubA1,SubA2):
  pass

c=C()
c.hello()
```

The output is "SubA2 says hello"

```
class A:
  def hello(self):
    print("A says hello!")

class SubA1(A): # Subclass of A
  pass

class SubA2(A): # Subclass of A
  def hello(self):
    print("SubA2 says hello!")

class C(SubA2,SubA1):
  pass

c=C()
c.hello()
```

The output is "SubA2 says hello"

Notice that both codes have the same result.

The mechanism provided in python to solve these issues is called "method resolution order" (MRO). We will illustrate it in later topics.

Aggregation

Aggregation provides the whole/part relationships. See the following example:

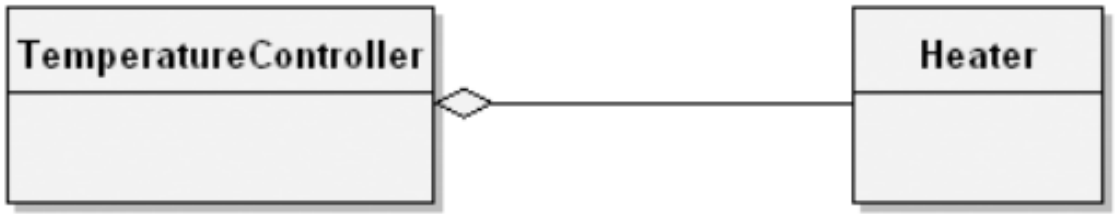


Figure: The aggregation relationship of class TemperatureController and Heater

16

A temperature controller can contain a heater in it. Therefore, we define a heater class Heater in the internal structure of TemperatureController.

[16] Example from: Object-Oriented Analysis and Design with Applications, 3rd, 2007

Aggregation

There are two kinds of aggregation relationships in classes.

- Composite Aggregation (Composition): Class B is a part of class A and does not independently exist
- Shared Aggregation (Aggregation): A Class B is a part of class A, but class B can exist by itself

Composition is a stronger form of aggregation. When an instance of class A is created, an instance of class B is also created. If we destroy A's instance, the corresponding B's instance is also destroyed. For example, the leg, hand, and head are part of a person and "the" person.

On the other hand, shared aggregation is a less directly kind of aggregation. For example, a car has wheels and wheels are one part of a car, but wheels can exist without car. Class Car and Wheel are not physical containment, they are aggregation.

Classes and Objects

The relationships and difference between classes and objects are

- Every object is an instance of some class, and every class has zero or multiple instances
- Classes are static. Their existence, semantics, and relationships are fixed before executing a program
- The class of most objects is fixed once an object is created. However, objects are created and destroyed frequently during the lifetime of an application

For example, planes, flight plans, runways, and air spaces are important concepts in an air traffic control system. Classes and their relationships of these abstractions are static. However, we can build or deactivate runways (create/destroy objects), and planes can enter or leave a particular air space (relationships between the objects may change)

Classes and Objects

Given the problem, here are two primary tasks during analysis and the early stage of system design

- Identify the key classes (key abstractions)
- Invent the structures such that objects can work together to provide required behaviours for the problem (mechanisms of the implementation)

The focuses during the initial and later stage of system design:

- Initial stage: outside view of the key abstractions and mechanisms
  - logical framework of the system
  - class and object structure of the system
- Later stage: inside view of the key abstractions and mechanisms
  - the physical representation of classes

Measuring the Quality of Abstraction

It is seldom for a developer to define the abstractions exactly right the first time. Refining is usually necessary. The following metrics measure the quality of an abstraction:

- Coupling(耦合): the strength of association established by a connection from one module to another
  - Strong coupling complicates the system, and highly interrelated modules make it hard to understand and maintain
  - Inheritance introduces coupling but also helps to exploit common abstractions
  - Weak coupling is preferred
- Cohesion(凝聚): the degree of connectivity among the elements of a single module (class or object)
  - Coincidental cohesion (least desirable form): entirely unrelated abstractions are thrown into the same class/module
  - Functional cohesion (most desirable form): all elements of a class/module work together to provide some well-bounded behaviour

Measuring the Quality of Abstraction

- Sufficiency: the class/module captures enough characteristics of the abstraction to provide required functionality
  - A minimal interface is preferred
- Completeness: the interface of the class/module captures all of the meaningful characteristics of the abstraction
  - Covers all aspects of the abstraction
- Primitiveness: unless necessary, restrict operations to access the underlying representation of the abstraction
  - For example: adding one item to set can be a primitive operation, but adding more items can be realized by calling the primitive operation
  - If it's possible and feasible, build operations by using existing primitive operations

Design Operations

To implement a behaviour, we can choose to

- design a simple but large interface with complicated methods
- design a complicated interface using many simple methods

A designer has to find the appropriate balance between these two choices.

The developer should also decide which class to place a behaviour in. For example, the process of enrolling a student in a course should be put in class "Student", "Course", or a new class "Enrollment"?

The following criteria can help to make the decision:

- Reusability: Would this behaviour be useful in more than one context
  - If it is useful to more than one type (class), factor out the code and create a new class to place it in
- Complexity: How difficult is it to implement the behaviour
  - Place complex behaviour in new class to avoid cluttering up the original class
- Applicability: Put specialized behaviour in specialized class
- Implementation knowledge: If the behaviour highly depends on the internal details of a type, put it in this class

Design the Relationships

Relationships between classes also affect how to design the operations:

Object Y is accessible to object X if X can see Y and reference resources in Y's outside view. Moreover, if Y is accessible to X, then X can send message to Y.

A guideline for choosing the relationships among objects is given below:

The Law of Demeter

The methods of a class should not depend in any way on the structure of any class, except the immediate structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only.

Therefore, if X requires Y's operations which require some behaviour defined in Z, instead of directly calling Z's methods, X should only invoke the methods in Y which indirectly call the methods in Z.

Applying the Law of Demeter can create loosely coupled classes, whose implementation secrets are encapsulated.

Design the Relationships

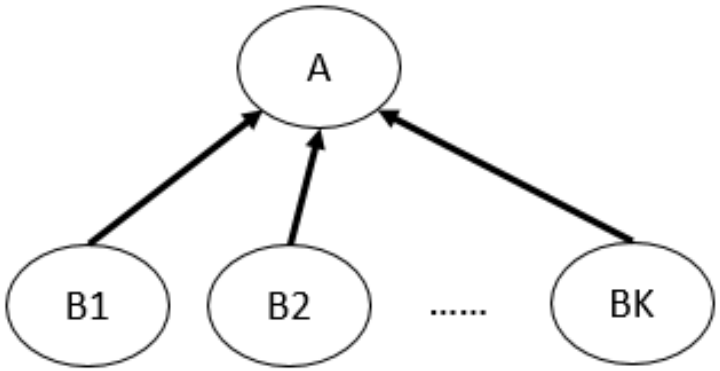


Figure: A wide and shallow hierarchy

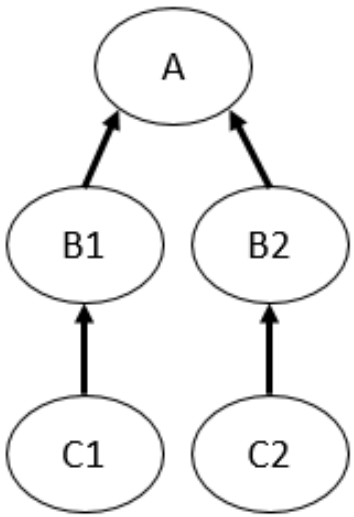


Figure: A narrow and deep hierarchy

Wide and shallow, and narrow and deep inheritance hierarchies have their own pros can cons:

- Wide and shallow: loosely coupled since subclasses share little common structure, but may be redundant (codes and internal structure)
- Narrow and deep: smaller individual classes and structure is inherited, but may need to understand all the ancestors to understand a class

The proper design is problem dependent.

Design the Relationships

How to decide the relationship between classes A and B be inheritance, aggregation, or association?

- Inheritance: if every instance of B may also be viewed as an instance of A.
- Aggregation: if the behavior of A is more than the sum of its individual parts (many Bs)
- Association: every instance of B simply accesses one or more attributes of A

Choosing Implementations

Only after the outside view of a given class/object is decided, we can turn to its inside view. The inside view includes

- The choice of representation for a class/object
  - Encapsulate internal structure
  - Detailed implementation depends on what/how the operations is used
    - Should you use array or linked list to implement a set?
  - Decide using a field to store the information or calculating it every time
    - Store the solution of the system of linear equations or not
- The placement of the class/object in a module
  - Generally, cohesive and loosely coupled modules are preferred
  - Minimizing the expected cost of maintaining the software is the purpose