Object Model

Yi-ting Chiang

Department of Applied Mathematics
Chung Yuan Christian University

September 19, 2023

Overview of Topics

- Topology of programming languages
- Object-oriented language, analysis, and design
- Object model
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy
    - Typing
    - Concurrency
    - Persistence

Topology of Programming Languages

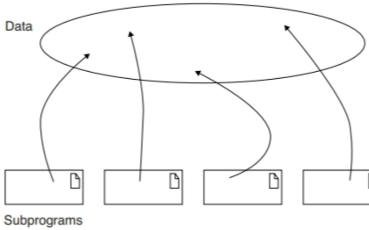

Figure: Topology[1] of the programming languages in the early generation.[2]

In the early generation, applications consist only of global data and subprograms. Subprograms depend on the data that are shared among all other subprograms. As a result, an error in one program can affect the whole system.

[1] "Topology" means the basic physical building blocks and how these building blocks can be connected.

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

Topology of Programming Languages

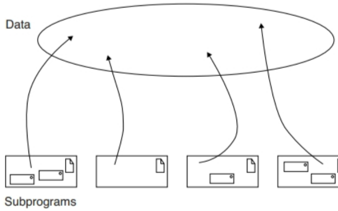

Figure: Making subprograms as an abstraction mechanism.[3]

Using subprograms as a way to abstracting program functions leads to the following consequences in modern programming languages:

1. Languages support variety of parameter-passing mechanisms (參數傳遞機制)
2. Languages support nesting of subprograms and control structures, and the development of the theory of scope and visibility of declarations (各種宣告的可見範圍)
3. Structured design methods help to build large systems using subprograms as building blocks

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

Topology of Programming Languages

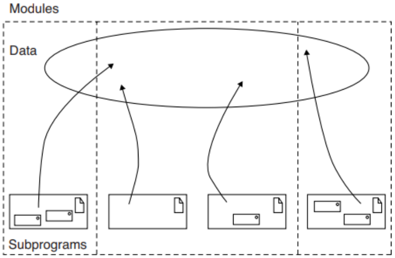

Figure: Separate a large project into compiled modules.[4]

As the scale of systems grows, it is required to separated the programming project into different parts (modules) such that each modules can be developed independently by different teams at the same time.

Programming languages in this generation can separate a program into several modules which are merely the containers of data and subprograms. Modules are usually composed of subprograms perform specific kinds of tasks and are most likely to change together.

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

Topology of Programming Languages

The following example gives a C program with four modules:

The first module defines a data (v) and a function (subprogram) to print information:

```
extern int v;
void show_data(const char*);
```

data.h

```
#include <stdio.h>
#include "data.h"

void show_data(const char* s) {
  printf("%s %d\n",s,v);
}
```

data.c

Topology of Programming Languages

The following example gives a C program with four modules:

The second module defines a function to increase a value defined in the data module.

```
#include "data.h"
extern int v;
int add();
```

increase.h

```
#include "increase.h"

int add() {
    v=v+1;
}
```

increase.c

Topology of Programming Languages

The following example gives a C program with four modules:

The third module defines a function to decrease a value defined in the data module.

```
#include "data.h"
extern int v;
int subtract();
```

decrease.h

```
#include "decrease.h"

int subtract() {
    v=v-1;
}
```

decrease.c

Topology of Programming Languages

The following example gives a C program with four modules:

The fourth module defines the main function that calls the subprograms defined in other modules.

```
#include "increase.h"
#include "decrease.h"

int v;
int main() {
  v = 100; //init
  show_data("The init value of v: ");
  add();           // increased by 1
  show_data("After add(): ");
  subtract();    // decreased by 1
  show_data("After add() and stbstract(): ");
  return 0;
}
```

Topology of Programming Languages

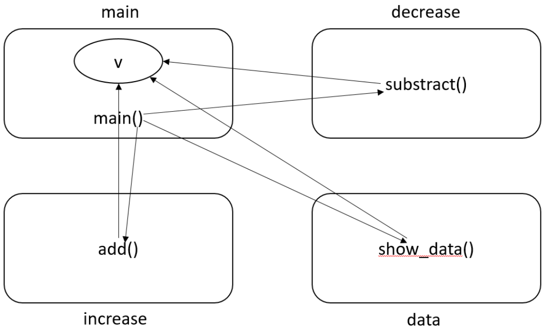

Figure: The topology of the example program

Each module in this example represents a subprogram and the data the subprograms require.

Topology of Programming Languages

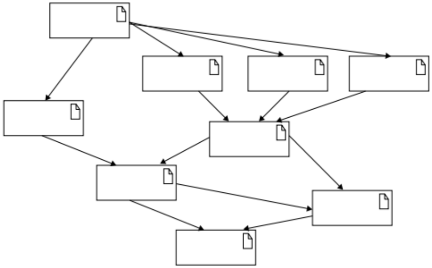

Figure: The topology of small to moderated-sized application using object-based programming language.[5]

In this generation, the building blocks are not subprograms but modules, which represent a logical collection of classes and objects. There is little or no global data.

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

Topology of Programming Languages

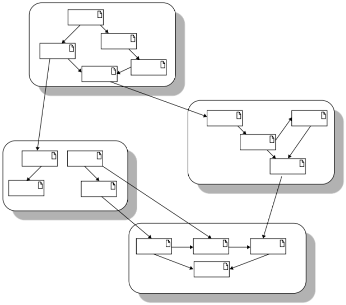

Figure: The topology of LARGE applications using object-based programming language.[6]

The object model scales up easily. A set of modules can compose a large module to construct a very complex system. In large systems, we can find meaningful collections of objects that collaborate to achieve some higher-level behaviour at any level of abstraction.

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

Object Model

Object

Entities that combine the properties of procedures and data since they perform computations and save local state

That is, objects serve to unify the ideas of algorithmic and data abstraction.
Objects can only change state, behave, be used/accessed, or stand in relation to other objects in ways appropriate to that object. In addition, there exist invariant properties that characterize an object and its behaviour[7].

[7] For example, an elevator can and only can travels up and down inside its shaft.

Object-Oriented Programming

Object-oriented Programming (OOP) can be defined as follows

Object-Oriented Programming

Object-oriented programming 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 relationship.

That is, in OOP

- Objects are the building blocks
- Each object is an instance of some class
- Classes may be related to one another via inheritance relationship ("is a" hierarchy)

A program written by object-oriented languages (for example, Java) doesn't mean the program is object-oriented program!

Object-Oriented Programming

A language is object-oriented if and only if[8]

- It supports objects that are data abstractions with an interface of named operations and a hidden local state
- Objects have an associated type (class)
- Types (classes) may inherit attributes from supertypes (superclasses)

To support inheritance, the language must be able to express "is a" relationships among types.
Some programming languages that can define classes but do not direct support inheritance are not considered "object-oriented" but only "object-base". However, one can still use object-oriented design methods on them.

[8]Cardelli, L., and Wegner, P. December 1985. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys vol. 17(4), p. 481.

Object-Oriented Design and Structured Design

The difference of structured design and object-oriented design (OOD) in logically structuring systems is:

- OOD uses class and object abstractions
- Structured design uses algorithmic abstractions

That is, the building blocks in the two design methods are different. Object-oriented design leads to an object-oriented decomposition.

Object-Oriented Analysis and Structured Analysis

The difference of structured analysis and object-oriented analysis (OOA) is:

- OOA emphasizes the building of real-word models, using an object-oriented view of the world
- Structured analysis focus on the flow of data within a system

OOA, OOD and OOP: OOA is the view when we build the model of the real world. Then this model (objects and classes) can be used to logically structure the system (OOD). Finally, the structure will be the blueprint for completely implementing a system using OOP methods.

Elements of the Object Model

There are seven principles (elements) of the object model:

- Major elements: a model without any of these elements is not object-oriented
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Minor elements: each of these elements is a useful, but not essential, part of the object model
  - Typing
  - Concurrency
  - Persistence

A model without any one of the major elements is not object-oriented. On the other hand, minor elements are useful, but not essential, parts of the object model.

We will only introduce the major elements.

Abstraction

Abstraction (抽象化)

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

That is, a good abstraction emphasizes details that are significant and suppresses details that are unimportant (to the viewer).
Notice that abstraction focuses on the outside view of an object. So it helps to separate an object's essential behaviour from its implementation.

Abstraction

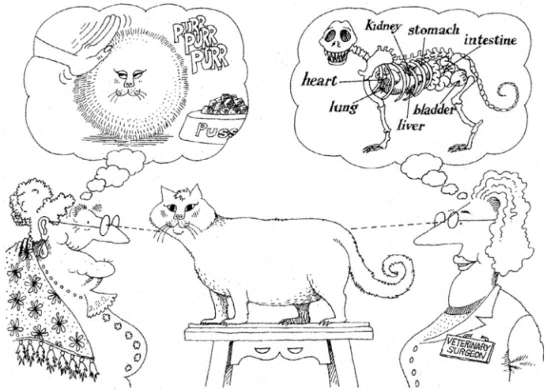

Figure: The abstraction of a cat by different viewers can be different.[9]

For the veterinary surgeon, the organs of a cat are important, but a grandma just need to touch the cat and hear the cat purrs

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

Abstraction

An abstraction serves to separate an object's essential behavior from its implementation. There are two principles about the behavior/implementation division:

- principle of least commitment: the interface of an object provides its essential behavior, and nothing more
- principle of least astonishment: an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction

Abstraction

Here we define some concepts and terminologies related to abstraction

- A client is any object that uses the resources of another object
- A server is the object that provides resources to other clients
  - An object can be both a client and a server
- The contract model: the outside view of each object (what resource this object may requires and what this object can provide, including its responsibilities) defines a contract on which other objects may depend, and which in turn must be carried out by the inside view of the object
  - The contract establishes all the assumptions a client object may make about the behaviour of this (server) object
- The protocol of an object: the entire set of operations that a client may perform on an object and the legal orderings these operations may be invoked in

Abstraction

- An invariance is some Boolean (true/false) condition whose truth must be preserved. For each operation in an object:
  - preconditions: the invariants assumed by the operation (must be true before perform this operation)
  - postconditions: the invariants satisfied by the operation (must be true after performing this operation)
  - Violating an invariant breaks the contract associated with an abstraction
    - Violate preconditions: the client does not satisfy its part of bargain (for example, the input parameters are not sufficient), and the server will not processed reliably
    - Violate postconditions: the server do not carry out its contract, and so its clients can no longer trust its behavior
- An exception is an indication that some invariant has not been or cannot be satisfied

Abstraction

Here we use an example from one of the reference materials to illustrate the concept of object model:

A hydroponics farm (水耕農場)

Plants on a hydroponics farm are grown without sand, gravel, or other soil. One must control diverse factors such as temperature(溫度), humidity(濕度), light, PH, and nutrient concentrations(養分的濃度). We have a automated system that constantly monitors and adjusts these elements. The purpose of the automated gardener is to efficiently carry out, with minimal human intervention, growing plants for healthy production of multiple crops.

Abstraction

Using the sensor in a hydroponic farm as an example. The abstraction of a temperature sensor is as follows:

Abstraction: Temperature Sensor
Important Characteristics: • temperature • location
Responsibilities: • report current temperature • calibrate (校正)

Characteristics of the sensor are

- temperature: the temperature this sensor senses
- location: the location of the sensor, which is an unique identity to distinguish the sensor from all the other temperature sensors

In addition, the sensor is designed to report the temperature at a given location. In addition, it can be calibrated. These are the responsibilities of the sensor. So a client can ask the sensor to report the current temperature and to calibrate it.

Abstraction

The abstraction of the sensor we have described is passive. To get the temperature, other clients have to operate on the sensor.

Another legitimate abstraction is to make the sensor an active one[10]:

Abstraction: Active Temperature Sensor
Important Characteristics: ● temperature ● location ● setpoint
Responsibilities: ● report current temperature ● calibrate ● establish setpoint

A client can invoke an operation to establish a critical range of temperature. The sensor can report the current temperature not only when is is asked by other client but when the temperature at its location drops below of rises above the given setpoint.

[10] Active sensor is more or less appropriate depends on the system design.

Abstraction

Another example of abstraction in the automatic system is the growing plan:

Abstraction: Growing Plan
Important Characteristics: ● name
Responsibilities: ● establish plan ● modify plan ● clean plan

The characteristic "name" illustrate the name of the plan. Note that each crop can has its own growing plan. We need the name to distinguish these plans. Its responsibilities are for the clients to create or change the plan.

As mentioned, there are several factors to control, so over the lifetime of this project, each responsibility (plan) may turn into multiple responsibilities such as "set PH", "modify PH" or "set light", "modify light".

Abstraction

More about the abstraction of the hydroponics farm example:

- Instead of asking the growing plan to carry out the plan, we leave this as the responsibility of another abstraction like “Plan Controller”.
- Defining the abstractions of the objects in a system is to create a separation of concerns(責任分離) among the logically different parts of the system.
  - Decide and reduce the conceptual size of each individual abstraction.
- Every object collaborates with other objects to achieve some behaviour.
- How these objects cooperate with one another defines the boundaries of each abstraction.
  - Thus defines the responsibilities and protocol of each object.

Encapsulation

Encapsulation (封裝)

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

Encapsulation

- is most often achieved through information hidden
- is the process of hiding all the secrets of an object that are not the object's essential characteristics
- hides the structure as well as the implementation of the methods of an object

Encapsulation and Abstraction are complementary concepts. Abstraction focuses on the observable behaviour of an object, where as encapsulation focuses on the implementation of this behaviour.

Encapsulation

The abstraction of a heater in the hydroponics farm can be defined as

Abstraction: Heater
Important Characteristics: • location • status
Responsibilities: • turn on • turn off • provide status

Any client of the heater needs to know the heater's available interface (responsibilities) that the heater may execute at the client's request. We may create another abstraction, a Heater controller, to maintain a fixed temperature.

Encapsulation

Encapsulation on the Heater:

- Any client only needs to know the interface of a heater and doesn't need to know how turn on/off and provide status are implemented
    - The signal from/to the heater can be sent by using any communication method which is independent with the interface of the heater
- The developer can change the detailed implementation of the heater without changing the clients
    - The essential benefit of encapsulation: the ability to change the representation of an abstraction without disturbing any of its clients
- Notice that information hiding is a relative concept
    - Assume a new abstraction of the communication method with the sensors is defined
        - For the heater: the communication method is hidden in the abstraction level
        - For the new abstraction: the communication method is the outside view in the abstraction level

Encapsulation

The two examples show how to encapsulate the same abstract with different implementation:

```
class date {
  private:
    int year;
    int month;
    int day;
  public:
    void setDate(const char*);
    void showDate();
    void modifyDate(const char*);
}; // defined in date.h
```

```
#include "date.h"
#include <cstdio>
void date::setDate(const char* s){
  sscanf(s,"%d-%d-%d",&year,&month,&day);
}
void date::showDate() {
  printf("%d-%d-%d\n",year,month,day);
}
void date::modifyDate(const char* s) {
  setDate(s);
}
```

Encapsulation

```
class date {
  private:
    char DateString[256];
  public:
    void setDate(const char*);
    void showDate();
    void modifyDate(const char*);
}; //defined in date2.h
```

```
#include "date2.h"
#include <cstdio>
#include <cstring>
void date::setDate(const char* s){
  strcpy(DateString,s);
}
void date::showDate() {
  printf("%s\n",DateString);
}
void date::modifyDate(const char* s) {
  setDate(s);
}
```

Encapsulation

```
#include "date.h"

int main()
{
    date D;
    D.setDate("2020-10-23");
    D.showDate();
    D.modifyDate("2020-12-31");
    D.showDate();
    return 0;
}
```

The C++ program uses the first date class since it includes date.h. You can include date2.h instead of date.h in the above code without modifying other lines to get the same result.
The client (in this case, the main function) only needs to know the interface (the three public functions), which is the outside view or the contract of class date, to use it.

Modularity

Related abstractions can be put together into the same module. These modules constitute the program. This is called modularity.

We can define modularity as:

Modularity(模組化)

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled components.

Make the modules

- cohesive: grouping logically related abstraction together
- loosely coupled: minimizing the dependencies among modules

In modularization, a program is divided into modules which can be compiled separately. However, there are connections between modules, so they can cooperate to perform tasks.

Modularity

About decomposing a system into modules:

- modularity should allow modules to be independently designed and revised
    - can change the implementation of a module without the knowledge of the implementation of other modules
    - can change the implementation of a module without affecting the behaviour of other modules
- module's structure should be simple enough to be fully understood
- module's interface should be as narrow as possible
    - First, hide as much as possible, and then incrementally move necessary elements only to the interface
- The data structure should be hidden
    - Other modules must call the module's program to get the information stored in the data structure

Hierarchy

Modularity clusters logically related abstractions to help manage complexity of the system. These abstractions often form a hierarchy, which can further help to simplify our understanding of the problem.

Here is the definition of hierarchy:

Hierarchy(階層)

Hierarchy is a ranking or ordering of abstractions.

Two important hierarchies in a complexity system have being mentioned before: “is a” hierarchy (class structure) and “part of” hierarchy (object hierarchy)

Hierarchy

Inheritance(繼承)

- is the most important ”is a“ hierarchy
  - a bear “is a” kind of mammal
- defines a relationship among classes
  - one class can share the structure or behaviour defined in one or more classes
  - a subclass inherits from one or more superclasses (respectively called single or multiple inheritance)
  - a subclass can extend or redefine the structure or behaviour of its superclasses
- implies a generalization/specialization hierarchy
  - if B is not a kind of A, B should not inherit from A

Hierarchy

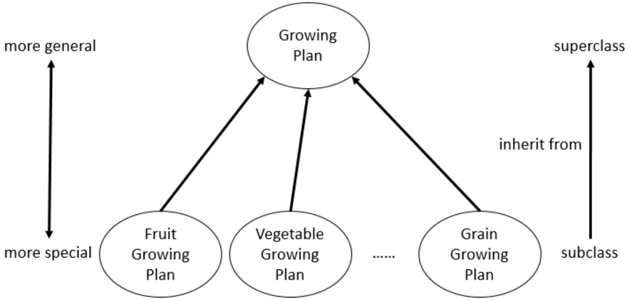

Figure: The hierarchy of the GlowingPlan classes

The figure above gives an example of the class structure in the hydroponics gardening system. Each subclass is a specialized growing plan that encapsulates the behaviour common for a kind of crop. The Growing Plan is the more general superclass.

Hierarchy

Superclass and subclass

- The structure and behaviour that are common in subclasses can migrate to common superclasses
- Superclasses represent general abstractions, and subclasses represent specialized abstractions
  - Subclasses add, modify, or even hide the fields inherit from the superclass

Hierarchy

```
class date {
  private String DateString;
  public void setDate(String s) { DateString=s; }
  public void showDate() { System.out.println(DateString);}
  public void modifyDate(String s) { setDate(s); }
}
class person {
  private int height;
  private date Birthday;
  public person() { Birthday=new date();}
  public void setBirthday(String s) { Birthday.setDate(s); }
  public void setHeight(int h) { height=h;}
  public void printBirthday() { Birthday.showDate();}
  public int getHeight() { return height;}
}
public class student extends person {
  private String school;
  public void setSchool(String s) { school=s;}
  public void showSchool() { System.out.println(school);}
  public void showData() {
    System.out.print("The student studies in ");
    showSchool();
    System.out.println("Height="+getHeight());
    System.out.print("Birthday=");
    printBirthday();
  }
}
```

This file is the implementation of the three classes, date, person, and student. Notice that neither getHeight() nor printBirthday() is defined in class student. These functions are the interface defined in the superclass person.

Hierarchy

Some programming languages allow the developers to define a subclass that inherits from multiple superclasses. This is called multiple inheritance.

Considering the following setting:

- the expected time to flower and the time to seed is important for a flowering plant
- the time to harvest is important to all fruits and vegetables

To define the abstraction of a specific kind of plant (for flowers, fruits, or vegetables), one can define two classes, Flower and FruitVegetable, both are subclasses of the class Plant.

Then, to defining a plant that can both flower and produce fruit, one can define another subclass of Plant, FlowerFruitVegetable. In this case, one have to copy information from both Flower and FruitVegetable, which is NOT a good solution!

Hierarchy

Defining class FlowerFruitVegetable can cause code duplication (you need to copy the codes from class Flower and class FruitVegetable).
Another solution is to use multiple inheritance:

- Define the class Plant
- Define the mixin classes FlowerMixin and FruitVegetableMixin
    - Mixin class means they are only for being mixed with another classes to produce new subclasses

Then, we can define class Rose as follows:

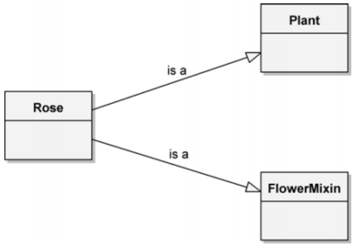

Figure: Rose is a kind of plant and flower.[11]

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

Hierarchy

The hierarchy of carrot can be:

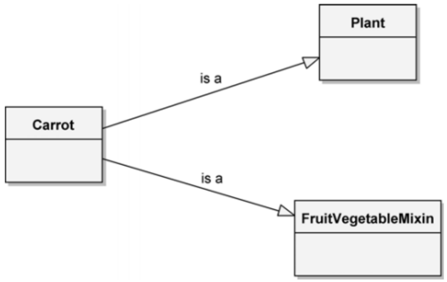

Figure: Carrot is a kind of plant and vegetable.[12]

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

Hierarchy

The flower and the fruit of cherry may both be important (depending on the species). Therefore, its hierarchy can be:

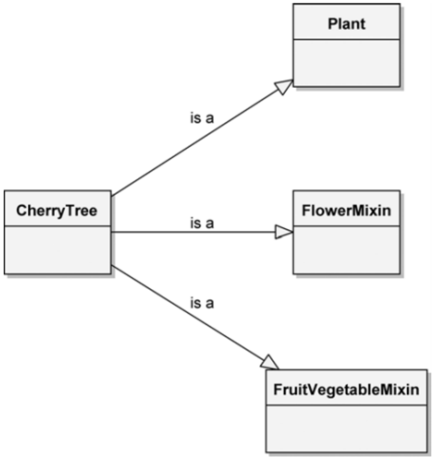

Figure: Cherry Tree inherits from two minix classes and the Plant class.[13]

Hierarchy

Two issues in multiple inheritance:

- Clashes: two or more superclasses provide a field or operation with the same name or signature as a peer superclass
- Repeated inheritance: two or more peer super classes share a common superclass

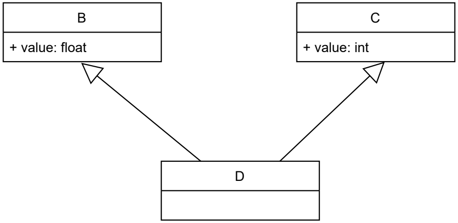

Figure: The problem of clashes. Which "value" should class D inherit?

Hierarchy

Two issues in multiple inheritance:

- Clashes: two or more superclasses provide a field or operation with the same name or signature as a peer superclass
- Repeated inheritance: two or more peer super classes share a common superclass

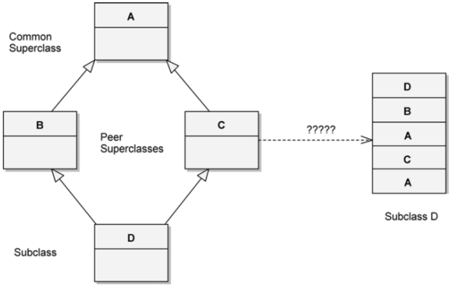

Figure: The repeated inheritance problem. Should there be one or two copies of A in D?[14]

The solution of the two issues differs depending on the programming language.

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

Hierarchy

"Part of" hierarchy describes aggregation relationships of objects. For example

- Wheels and doors are part of cars.
- The name, height, and weight are part of the properties of a person.
- The plants and growing plans are part of the hydroponics garden.

In C programming language, you can define fields in a structure. These fields are part of the structure.

Inheritance and Aggregation

- Aggregation permits the physical grouping of logically related structures.
- Inheritance allows the common groups to be reused among different abstractions.

"Ownership" is an issue in aggregation. This issue is about the lifetime of an object which is "part of" another object in a system. We will discuss this issue later.

Summary

In this topic, we introduced seven elements of the object model.

- Abstraction
- Encapsulation
- Modularity
- Hierarchy
- Typing
- Concurrency
- Persistence

The first four are major elements. A model without any one of the major elements is not object-oriented. On the other hand, minor elements are <u>useful, but not essential, parts</u> of the object model.[15]

[15]The last three kinds are the concept of object-oriented databases. In practice, object-oriented databases build on existent technology and offer to the programmer the abstraction of an object-oriented interface. Database queries and other operations are completed in terms of objects. These objects will exist even when the program terminates.

Benefits of Object Model

Using object model help to create a well-structured complex system with the five attributes mentioned in the previous topic: 1. hierarchy, 2. relative primitives(multiple levels of abstraction), 3. separation of concerns, 4. common patterns, and 5. stable intermediate forms. In addition, it has the following benefits:

- exploiting the expressive power of object-oriented programming languages
- encouraging the reuse of not only software but also the entire design
  - less codes to write and maintain
  - greater cost and schedule benefits
- being more flexible to change
- reducing development risk and increasing the confidence in the correctness of the design
- being more agreeable to the workings of human cognition[16]

[16] People who have no idea how a computer works find the idea of object-oriented systems quite natural.