B39HV2

SOFTWARE ENGINEERING II

UNIT 1: Introduction to Object-Oriented Programming

Recommended Books

- **C++ Programming Today**
  - ***C++ Programming Today*, Barbara Johnston, Prentice Hall, 2001**
    - **Text Book**
    - **Easy and Friendly**
    - **Incomplete**
  - ***How to Program in C++ 4th edition*, Deitel & Deitel, Prentice Hall, 2003,**
    - **Reference Book**
    - **Complex**
    - **Complete**
  - **Shaum's outlines Programming with C++, John Hubbard, 2nd Edition**
    - **Summary book**
    - **Lots of examples**

Aims

- In this unit we will consider the following topics:
  - The need for object-oriented software development.
  - Objects
  - Classes
  - Relationships between objects.
  - Inheritance
  - Encapsulation
  - Polymorphism

Why object-oriented?

- This unit will discuss what is meant by the term object-oriented and why it is of such interest to programmers.

- At the dawn of programming:
  - Computers and storage devices were very expensive, programmers were cheap.
  - Programming tasks were incredibly simple by today’s standards.
  - There was little marketing of software - organisations wrote their own code to solve their own problems.

- Nowadays:
  - Hardware is cheap and programmers are expensive.
  - Programming tasks are becoming phenomenally complex.
  - There has been an explosion in the use of computers and a subsequent explosion in the software market.

How it was . . .

- In olden days, the emphasis was on getting the job done with the minimal available resources:
  - The emphasis on code was to make best use of the limited speed and memory available. Little consideration was given to issues such as portability, maintainability, understandability etc.

- There are many legacies from this age that come back to haunt us:

  - Some institutions use programs that cannot be properly maintained or upgraded as the programmer has retired or died!.

  - The millenium bug that we face today, came about because programmers in the 60’s wanted to save 2 bytes in representing a date, by removing the figures denoting the century:
    - Hence does 1/1/00 represent 1/1/1900 or 1/1/2000?.

How it is . . .

- Nowadays, programmers work in a highly competitive market place. The challenge is to create code that:
    - works correctly
    - operates efficiently
    - contains no bugs
    - can be maintained and updated
    - ships before its rivals.
- All this, and the problem domain is becoming more and more complex.
- In practice, It is often the *first* system to reach the market that sells, not the *best* system.
    - Hence there is a trade-off between quality and development speed.
    - This explains why so much of today's software contains bugs.

Managing complexity

- It can take several years (~5) to bring a new model of a car to the market.
  - This is despite the fact that the design and construction principals in the motor industry have changed relatively little over a century.
  - Also, a new model will behave in roughly the same way as previous ones in testing. The properties of the materials will remain the same, as will the laws of physics.
- In contrast, a piece of software can take as little as 6 months from brainstorming design to shrink-wrapping.
  - It may use entirely different principals to previous software. It will contain much greater complexity and offer much less well understood behavior than mechanical systems such as cars.
  - As a result, we have come to expect less reliability from software products than for cars, televisions, etc.

Importance of reliability

- The safety and reliability requirements of cars are obvious.
- Nowadays, software is being placed in more and more sensitive areas:
  - Control of industrial processes such as chemical engineering and nuclear power plants.
  - "Fly-by-wire" control of airplanes and air-traffic control operations.
  - Medical diagnosis and intensive care control.
- Consequently it is necessary to approach such applications with more rigor than early computer applications such as word processing etc.

What is required . . .

- What is required is a *methodology* that allows us to manage the complexity of the software development process and enable us to produce reliable, high quality software that will ship on time.

  - In the dark ages of programming, code was unstructured and data was global. This led to convoluted and fragile code, that was not easily maintained or updated.

  - It didn’t take long before such programs became unmanageable. The use of the `goto` statement led to unstructured "*spaghetti code*".

  - Later, *structured* programming languages (e.g FORTRAN, C) used procedural abstraction to create more reliable programs.

Structured programming

- ❍ **Structured programming languages provided a great improvement over the first programming languages.**
  - ❒ **The use of conditional and repetition constructs (`if`, `while` etc) ensure that code was structured and the thread of execution was easily decipherable.**
  - ❒ **Although these languages contained the `goto` statement, its use was greatly frowned upon.**
  - ❒ **Importantly, the ability to create functions greatly enhanced the ability of programmers to use abstraction in tackling complex applications.**
  - ❒ **Consequently programming became a more systematic and engineered discipline.**

However . . .

- However as is often the case, new approaches may solve old problems but can introduce new ones:
  - Structured programming languages provided a good solution to the relatively simple problems encountered by early programmers.
  - However, just as road-building increases *not* decrease traffic, the new efforts to tackle complexity simply increased the complexity of what programs became expected to do.
  - Applications were becoming much larger and more critical. Hence the problems to be solved were more complex and the consequen-ces of program bugs and crashes were becoming more significant.
  - It became apparent that the costs of correcting bugs increased dramatically, the later in the development process that they were detected.

Bugs

- Legend has it that software errors become known as *bugs* after an error in an early program was tracked down to an insect trapped in a relay in the magnetic (core) memory.
  - Developers spend much of their time finding bugs in programs. Consequently much of the costs of development are concerned with finding and correcting bugs.
  - The later the bugs are found in the process, the more expensive it becomes to correct them.
  - Bugs found after a release of software can by 100 times as expensive to correct as bugs found early in the development.
  - Consequently, software developers began looking for a method that would prevent bugs from creeping in to programs.

A funny story about bad software

In March 1998 a man living in Newtown near Boston Massachusetts received a bill for his as yet unused credit card stating that he owed $0.00. He ignored it and threw it away.
In April he received another and threw that one away too. The following month the credit card company sent him a very nasty note stating they were going to cancel his card if he didn't send them $0.00 by return of post.
He called them, talked to them, they said it was a computer error and told him they'd take care of it. The following month he decided that it was about time that he tried out the troublesome credit card figuring that if there were purchases on his account it would put an end to his ridiculous predicament. However, in the first store that he produced his credit card in payment for his purchases he found that his card had been cancelled.
He called the credit card company who apologized for the computer error once again and said that they would take care of it. The next day he got a bill for $0.00 stating that payment was now overdue. Assuming that having spoken to the credit card company only the previous day the latest bill was yet another mistake he ignored it, trusting that the company would be as good as their word and sort the problem out.
The next month he got a bill for $0.00 stating that he had 10 days to pay his account or the company would have to take steps to recover the debt. Finally giving in he thought he would play the company at their own game and mailed them a check for $0.00. The computer duly processed his account and returned a statement to the effect that he now owed the credit card company nothing at all.
A week later, the man's bank called him asking him what he was doing writing a check for $0.00. After a lengthy explanation the bank replied that the $0.00 check had caused their check processing software to fail.
The bank could not now process ANY checks from ANY of their customers that day because the check for $0.00 was causing the computer to crash! The following month the man received a letter from the credit card company claiming that his check had bounced and that he now owed them $0.00 and unless he sent a check by return of post they would be taking steps to recover the debt.
The man, who had been considering buying his wife a computer for her birthday, bought her a typewriter instead.

A new methodology

- Software developers sought a new approach for developing software. One that would . . .
  - Produce highly maintainable software, where changing one part of a program would not break the rest.
  - Produce reusable software, where similar programming tasks could share common elements, without having to reinvent the wheel.
  - Make it easier to tackle complex programming problems and reduce the chances of introducing bugs into programs.
- These ideas led to the development of *object-oriented* (OO) software development.

Object-oriented software development

- **The key idea behind object-oriented development is a <u>fundamental</u> change in approach:**
  - **Structured programming reflects the way that *<u>computers</u>* process information. They implement sequential execution, conditional branching and repetition. As such, a problem is tackled by decomposing it into a *<u>flow</u>* of *<u>operations</u>*, data is of secondary importance. Because *we* do not naturally think in this way, dealing with complex problems is difficult and bugs may not be obvious.**
  - **The object-oriented approach attempts to find a methodology and language that reflect the way that *<u>people</u>* think about problems. In fact, it appears that people tend to think in terms of *<u>things</u>* not operations. In OO development we refer to these "things" as objects.**

Objects

- What the objects are, will be determined by the problem <u>domain</u>, i.e. the context in which the software is to be used:
  - In a banking application, our objects will be customers, accounts, sums of money etc.
  - In a racing simulation game, our objects could be players, cars, barriers, race-tracks etc.
  - In a library application, our objects will be books, journals, CD-roms, members, etc.
  - In an air-traffic control application, our objects will be planes, air-corridors, radar screens, etc.
- In a "nutshell", we start by considering what "things" are involved in a problem, then we consider what these objects can do and represent. We go on to consider how these objects are related and interact and eventually implement a solution.

Objects

- **An important thing to realise, is that OO development starts by thinking about the *problem* and what it entails, and not about the *program* that will implement the solution.**
  - **In the real world, "things" have *characteristics* and *behavior*. A car is a "thing" that has a make, model, registration number and can accelerate, steer, brake, change gear etc.**
  - **We think about the objects involved in the problem, about what are their *characteristics* (*attributes*) and *behavior*.**
  - **An object-oriented language will allow us to *model* these attributes and behaviors.**
  - **We construct a solution to the problem from these objects.**

What is an object?

- An object has *state*, *behaviour* and *identity*:
  - The state of an object is the particular *values* of its attributes.
    - In an air-traffic control application each plane may be an object with several attributes (height, heading, speed, call-sign) , but an individual plane object will have its own state (current height, heading, speed).
  - The behaviour of an object is the actions that the object can perform.
    - A plane can climb, descend, accelerate, decelerate, land, take-off, etc.
  - *State can affect behaviour* (if *height* reaches a value, the landing carriage is lowered).
  - *Behavior can affect state* (as a plane *climbs* its *height* will increase).
  - The identity of an object distinguishes it from all other objects - two BA 747's are not the same plane, they merely share some attributes.

Classes

- As we can see, we may have several objects that represent different versions of the same "thing".
  - This "thing" is known as a *class*, and is the "template" from which the objects are "stamped".
- In C++, a class is implemented as a special kind of user defined data type, similar in some regards to structures.

```
class plane {
   // attributes
   // behaviour
};
```

- An object can be declared as an *instance* of a class:

```
plane  spirit_of_st_louis, airforce_one;
```

Classes

- The *class* will describe what attributes an *object* can have, and what behaviour it can perform. In C++, . . .
  - the <u>attributes</u> are known as *member variables*.
  - the <u>behaviour</u> is known as *member functions* (also called *methods*).
- All objects (instances) of a class will have the same attributes and behaviour, but will have their own *state* and *identity*.

Relationships between objects

- The real power of the object-oriented approach is to do with the *relationships* ***between*** objects. These include:

- Dependency
  - A dependency is where one object must *know about* another.
  - E.g. An object ***plane*** must know about an object ***ground***.

- Association
  - An association is where the *state* of one object *depends* on another.
  - Association is stronger than dependency, it specifies that two objects have a strong connection but neither is a part of the other.
  - E.g. A ***pilot*** flies a ***plane***, a ***carrier*** owns a ***plane***.

Relationships between objects . . .

- **Aggregation**
  - **Aggregation specifies how objects can be *constructed* from other objects.**
  - **E.g. a *plane* has *engines*, *wings*, *instruments* etc. Each component is an object in its own right.**

- **Composition**
  - **Composition expresses a closer relationship than aggregation, where one object is composed of others, and these *component objects only exist during the "lifetime" of the whole*.**
  - **E.g. a *book* is composed of *chapters*, *contents* etc.**
  - **Aggregation specifies a looser relationship - a *plane* may have it's *engines* changed.**

Inheritance

- Inheritance is a very important relationship that can exist between *classes*.
  - The type of relationship that is represented by inheritance, is commonly called an "*is a*" relationship. e.g. a dog *is a* mammal.
  - The class *dog* can be said to *inherit* attributes and behaviour from the class *mammal*, as well as adding new ones of its own.
    - All mammals are warm blooded, air breathing animals that bear live young. Hence we know that dogs share these attributes and behaviours with other mammals.
    - The class *dog* adds to these inherited attributes and behaviours, the attributes "furry" and "legs" and the behaviours "chase cats", "wag tail", amongst others.
    - A *whale* is also a mammal, and as such will share some attributes and behaviours with *dogs*, but add new ones of their own, e.g. "fins", "blow hole", "swim", "migrate".

Inheritance

- It is easy to see that the concept of inheritance allows us to organise a hierarchy of classes, where only the specialisations of new classes need be specified and inherited attributes and behaviour can be assumed.

- This is an example of *reuse* and it allows us to compactly describe classes without having to repeat ourselves:
  - A *cat* is a carnivorous *mammal* with a furry coat and retractable claws that hunts smaller animals.
  - A *tiger* is a large stripey *cat* that lives in Asiatic forests and grassland. Has been known to eat people!
  - A *tiger* will inherit the features of *mammals* and the specialisations of *cats* and hence we know a lot about it before we start to mention its particular specialisations.

Inheritance in programming

- **Just as inheritance is a useful concept in taxonomy and genetics, it is an important part of the OO approach to programming.**
  - **You write a class `account` for a banking application. This class may have *attributes* such as "customer details", "balance" etc, (some of these attributes may be objects of other classes), and *methods* such as "open", "withdraw", "deposit" etc.**
  - **If the bank also offers an account with an overdraft facility, you can write a new class `account_with_overdraft` that will inherit all the attributes and behaviour from `account` and add new attributes "overdraft limit", "interest", and behaviour "charge interest".**
  - **By using inheritance, you now have two classes for less than twice the work!**

Some terminology

- The class `account` is known as the *base class*, as it forms the top of the hierarchy.
- The class `account_with_overdraft` is known as a *derived class*, where it has been *derived* from the base class `account`.
- The class `account_with_overdraft` is said to have *inherited* the member variables and methods from the class `account`.
- Just as a duck-billed platypus lays eggs, it is possible for some derived classes to *override* inherited behaviour.

The "three pillars" of OO programming

- Inheritance is such an important concept that it is regarded as being one of the "three pillars" of OO programming.
  - The other two are *encapsulation* and *polymorphism*.

- <u>Encapsulation</u>
  - As its name implies, encapsulation ensures that each object is *self contained*.
  - We want to hide the internal workings of the object and only present the user of the object with those features that they require to use it.
  - Early cars were poor examples of encapsulation, with motorists having to prime the fuel pump, set an idle angle and engage a starter crank, before starting the car, and when changing down a gear they had to double de-clutch. Nowadays we don't need to understand the internal workings of a car to drive it.
  - Hence, if presented with a diesel engined car, we are able to drive it.

Encapsulation

- The principal behind encapsulation is:

  "The more you expose the workings of an object, the more likely someone is to use it in the wrong way".

  - **If you allow someone to use a software component in the wrong way, then you are in danger of introducing a bug into the program.**
  - **Hence a well encapsulated object will only allow the user to employ it in the way in which it was intended.**
  - **This principal removes the "ripple effect" experienced in software projects, where a minor change in one part of the program creates a chain reaction that affects the correct operation of the program.**

An example to explain encapsulation

- For example, suppose that you have created a class `date` that has member variables `day`, `month` and `year`, and methods to compute differences between dates etc.
  - Initially you decide to use `ints` to represent the `day`, `month` and `year` data members, and release a library containing the implementation of this class for use by other programmers on the project.
  - Later, for some reason, you decide to represent the `month` data member as a string instead, i.e. "January", "February" etc. You make this change and update the library.
  - Very shortly, the prototype system starts to perform strangely giving unexpected operation and inaccurate results!

WHY?

An example to explain encapsulation . . .

- ❒ Unknown to you, another programmer has decided to substitute his own routine for computing the difference between dates, including the line:

```
if (start.month > end.month)
```

- ❒ This operation will now be comparing the *addresses* of where the strings for the member variables are stored, and not the months themselves.

- ❒ The problem came about, because you allowed other programmers to access the internal workings of your class.

- ❒ Encapsulation ensures that the only methods that they have for using the objects are the ones that you give them. Hence when you make a change to the class, you simply update the methods to ensure correct operation.

Encapsulation

- This example illustrates a key advantage of encapsulation.
  - By decomposing the task into a set of objects and <u>testing and debugging these objects individually</u>, we can construct a larger application that is much less susceptible to bugs.
  - The properties and actions of materials and components in car manufacture are well understood, and consequently the properties and actions of the assembled vehicle are predictable.
  - Likewise the behaviour and operation of a large software application can be better predicted when it is assembled from well encapsulated objects.
  - Most bugs are ironed out in the development and testing of the objects. New ones are limited due to restrictions on the use of these objects.

Interfaces and implementations

- **We implement this concept of encapsulation by ensuring a well defined distinction between the object's *interface* and it's *implementation*.**
  - **The *interface* of an object is the set of accessible member function declarations.**
  - **The *implementation* of an object is all the data and function members that are required to actually perform the object's tasks.**
  - **An object that interacts or uses your object, is known as a *client* of your object. Likewise your object will be known as the *server*.**
  - **A *client* object needs only know the <u>interface</u> of the *server* object in order to use it's services.**

Encapsulation in C++

- **Encapsulation is achieved in C++ by allowing different levels of membership, including *public* and *private*.**
    - **The <u>interface</u> of the class is placed in the *public* section.**
    - **The <u>implementation</u> of the class is placed in the *private* section.**
- **The implementation is entirely hidden from view. Hence you are free to change the way in which a method is implemented without affecting the way the object is used.**
- **This is known as *information hiding* - only the parts of a class that a programmer needs to know to use it, are exposed.**

Polymorphism

- The third of the "three pillars" of object-oriented programming is *polymorphism*.
  - The word "polymorphism" means literally "*many forms*".
  - We have already met a type of polymorphism in C++, namely *function and operator <u>overloading</u>*.
  - Remember that overloading mimics the ability to use the same *word* to represent similar but distinct *operations* in different contexts:

    You can *push* a button, mouse, car, your luck etc.
  - In C++ we wrote `display()` functions for money, dates and times.
  - In C++ this type of polymorphism is known as *function polymorphism*.

Function overloading

❍ **Consider a program that requires functions to display a sum of money, a time, and a date:**

```
void display(float amount)
{
   cout << setiosflags(ios::fixed|ios::showpoint);
   cout << setprecision(2) << "£" << amount;
}
```

```
void display(int hour, int minute)
{
   cout << setw(2) << setfill('0') << hour << ":";
   cout << setw(2) << setfill('0') << minute;
}
```

```
void display(int day, int month, int year)
{
   cout << day << "/" << month << "/" << year;
}
```

Function overloading

- **Without function overloading, we would be required to provide individual names for each function.**
  - **`E.g. display_money(),display_time(),display_date().`**

- **The question remains:** ***"how does the compiler tell which `display()` function we mean?"*.**

The answer is in the *type* and *number* of the arguments.

```
cout << "The sum of ";
display(withdrawal);
cout << " was removed from your account at ";
display(hour,min);
cout << " on ";
display(d,m,y);
```

Function overloading

- In this example, we have 3 functions called `display()`.
  - The compiler can determine which function to use by comparing the number of arguments in each call to the function declarations.

```
cout << "The sum of ";
display(withdrawal);                // 1 argument
cout << " was removed from your account at ";
display(hour,min);                  // 2 arguments
cout << " on ";
display(d,m,y);                     // 3 arguments
```

```
The sum of £7.85 was removed from your account at 17:05 on 2/6/98
```

Function overloading

❍ **Using structures called `time` and `date`, would still be OK as the *types* of the single arguments will differ:**

```
void display(float amount)
{
   cout << setiosflags(ios::fixed|ios::showpoint);
   cout << setprecision(2) << "£" << amount;
}
```

```
void display(time t)
{
   cout << setw(2) << setfill('0') << t.hour << ":";
   cout << setw(2) << setfill('0') << t.minute;
}
```

```
void display(date d)
{
   cout << d.day << "/" << d.month << "/" << d.year;
}
```

```
cout << "The sum of ";
display(withdrawal);   // float argument
cout << " was removed from your account at ";
display(t);            // time argument
cout << " on ";
display(day);          // date argument
```

Polymorphism

- **Another type of polymorphism exists in C++, namely *object polymorphism*.**
  - **This type of polymorphism is used with derived classes.**
    - **It is possible for objects from classes in an inheritance hierarchy to implement a particular method in the way that suits it best.**
  - **Let us first consider a non-programming example:**
    - **A base class *vehicle* has the behaviours *accelerate*, *decelerate* and *steer*.**
    - **We derive the classes *car*, *bicycle* and *boat*, i.e. they are all specialisations of *vehicles*.**
    - **Each class inherits the behaviours *accelerate*, *decelerate* and *steer* and implements them in the *appropriate way*.**
    - **Hence if we want a *car* object to *steer* we turn the steering wheel, a *bicycle* object we turn the handlebars and a *boat* object we turn a rudder.**
    - **In each case the meaning is the same but the actual action differs.**

Polymorphism in C++

- **To explain polymorphism in C++ consider the following example:**
  - **In the banking example, we have a base class `account` and a derived class `account_with_overdraft`.**
    - **Both classes have the member function `open()`.**
  - **When we open an object of class `account` we need to specify the customer details and initial balance.**
  - **When we open an object of class `account_with_overdraft`, we need to also specify the size of the allowable overdraft.**
  - **Hence the implementation of the member function `open()` will differ for the two classes.**
  - **In C++, we implement this type of polymorphism by using *virtual functions*.**

A function that is declared *virtual* in the base class can be *overridden* in a derived class.

Polymorphism in C++

- Let us consider another example:
  - Suppose that you are working on a graphics application. You write a class called `shape`.
  - You then *derive* the classes `square`, `circle` and `triangle`.
  - Each class requires the member functions `set_size()` and `compute_area()`.
  - We cannot provide an implementation for these functions in the base class `shape`, as we do not know the form of the shape.
  - Instead we declare them as *pure virtual* functions (no definitions are provided). This makes `shape` an *abstract base class*.
  - In each of the derived classes we can provide an appropriate implementation for the member functions.
  - We cannot declare any objects of class `shape`. The class `shape` only exists to represent the relationship between the classes `square`, `circle` and `triangle`.

SUMMARY

- Objects can have state, behaviour and identity.
- A class represents a "template" for creating objects, and details the attributes and behaviour that the objects can possess.
- An object can be related to another object by the relationships of dependency, association, aggregation and composition.
- Another, stronger type of relationship is inheritance, where a derived class can inherit characteristics from a base class and add its own specialisations.

SUMMARY

- Inheritance is the first of the "three pillars of object-orientation". The other two are encapsulation and polymorphism.
- Encapsulation is the combination of state and behaviour into an object.
- Information hiding is implemented by placing the implementation of an object in a private section whilst presenting the interface of an object in the public section.
- Polymorphism allows your software elements to work in the appropriate way.