www.educational-computing.co.uk/OOPSample.doc

Structured Programming versus Object-Oriented Programming

Software engineering is a discipline that is concerned with the construction of robust and reliable computer programs. Just as civil engineers will use tried and tested methods for the construction of buildings, software engineers will use accepted methods for analysing a problem to be solved, a blueprint or plan for the design of the solution and a construction method that minimises the risk of error. The discipline has evolved as the use of computers has spread. In particular, it has tackled issues that have arisen as a result of some catastrophic failures of software projects involving teams of programmers writing thousands of lines of program code. Just as civil engineers have learned from their failures so have software engineers.

A particular method or family of methods that a software engineer might use to solve a problem is known as a **methodology**. During the 1970s and into the 80s, the primary software engineering methodology was **structured programming**. The structured programming approach to program design was based on the following method:

- To solve a large problem, break the problem into several pieces and work on each piece separately;
- to solve each piece, treat it as a new problem that can itself be broken down into smaller problems;
- repeat the process with each new piece until each can be solved directly, without further decomposition.

This approach is also called **top-down program design**.

The following is a simple example of the structured programming approach to problem solving.

Write a program for a computer to execute to display the average of two numbers entered through a keyboard connected to the computer. The average is to be displayed on a VDU that is also connected to this computer.

The top-down solution is arrived at as follows:

Top level: 0. Display average of two numbers entered through keyboard

Next level: 0.1. Get two numbers through keyboard
0.2. Calculate average of these two numbers
0.3. Display average on VDU

The three steps in *next level* can now be coded in a programming language such as Pascal.[1]

[1] See appendix at end of this guide for a structure chart solution to this exercise.

Top-down program design is a useful and often-used approach to problem solving. However, it has limitations:

- It focuses almost entirely on producing the **instructions** necessary to solve a problem. The design of the **data structures** is an activity that is just as important but is largely outside of the scope of top-down design.
- It is difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down program design tends to produce a design that is unique to that problem. Adapting a piece of programming from another project usually involves a lot of effort and time.
- Some problems by their very nature do not fit the model that top-down program design is based upon. Their solution cannot be expressed easily in a particular sequence of instructions. When the order in which instructions are to be executed cannot be determined in advance, easily, a different approach is required.

Top-down design was therefore combined with **bottom-up design**. In bottom-up design, the approach is to start "at the bottom", with problems that have already been solved and for which a reusable software component might exist. From this position, the software engineer works upwards towards a solution to the overall problem.

It is important in this approach that the reusable components are as "modular" as possible.

A module is a component of a larger system that interacts with the rest of the system in a simple and well-defined manner.

The idea is that a module can be "plugged into" a system. The details of what goes on inside a module are not important to the system as a whole, only that the module fulfils its function correctly. For example, a module might contain procedures to print a list of students, to add a new student, edit a student's details and to return a list of specified students. How the module stores the master records of student details is hidden from applications/systems that use this module. Similarly, the detail of how the various procedures are coded is also hidden. This is called **information hiding**. It is one of the most important principles of software engineering. Applications only require knowledge of what procedures are available from the module and the data that can be accessed. This information is published. It is often called the **module's interface or interfaces**.

A common format for a software module is a module containing some data, along with some subroutines (subprograms/procedures/functions) for manipulating that data. The data itself is often hidden from view inside the module forcing a program using the module to manipulate the data indirectly, by calling the subroutines provided by the module for this purpose. The advantages of this approach are as follows:

- the data is protected, since it can be manipulated only in known, well-defined ways;
- it is easier to write programs to use a module because the details of how the data is represented and stored need not be known;
- the storage structure of the data and the code for the subroutines in a module may be altered without affecting programs that make use of the module as long as the published interfaces and the module's functionality remain the same.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. The concept has been developed since then into a central platform of software engineering called **object-oriented programming**, often abbreviated as **OOP**.

The central concept of object-oriented programming is the **object**, which is a kind of module containing data and subroutines. An object is a kind of self-sufficient entity that has an **internal state** (the data it contains) and that can respond to **messages** (calls to its subroutines). A student-records object, for example, has a state consisting of the details of all registered students. If a message is sent to it telling it to add the details of a new student, it will respond by modifying its state to reflect the change. If a message is sent telling it to print itself, it will respond by printing out a list of details of all registered students.

The OOP approach to software engineering is to begin by

- identifying the objects involved in a problem;
- identifying the messages that those objects should respond to.

The solution that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other.

There is not much "top-down" in such an approach. People used to the structured programming approach find OOP hard at first. However, people who have mastered OOP claim that object-oriented programs tend to be better models of the way the world itself works. They claim that this produces solutions that are

- easier to write
- easier to understand
- contain fewer errors

Why is the object-oriented approach considered to be better at modelling the way the world itself works? Consider the following exchange between two people:

First person: **Jack, are you hungry?**
Second Person: **Yes I am, Jill.**

Classifying this exchange in object-oriented terms, there are two messages flying between two objects. The objects are Jack and Jill. The messages are "are you hungry?" and "yes I am". Jack knows how to respond to the message he receives. His set of responses might be [Yes I am, No I am not]. Jack chooses the appropriate response by examining his internal state, i.e. the contents of his stomach.

Objects are created knowing how to respond to certain messages. Different objects might respond to the same message in different ways. If Jack is a robot, the response to the message "Jack, are you hungry?" might be "that is a silly question, I do not have human characteristics". This property of objects – that different objects can respond to the same message in different ways – is called **polymorphism**.

Another important concept in object-oriented programming is the concept of a **class**.
It is quite common for objects to belong to the same family. These are objects that contain the same type of data and that respond to the same messages in the same way. Such objects are said to all belong to the same class. In programming terms, a class is created and then one or more objects are created using that class as a template. For example, the clock family consists of devices that keep track of the passage of time. When their internal state changes they update a display consisting of hours, minutes and possibly seconds. The only message that a member of the clock family might know how to respond to is one that tells it to reset its internal state to some hour and minute. There are many clocks in existence – clock objects. They all belong to the clock family or class.

The classification of all clocks into one big family called the **clock class** is an oversimplification. We can divide the clock class into two **subclasses** called **analogue display clocks** and **digital display clocks**. They are related but differ in the way that they display time.

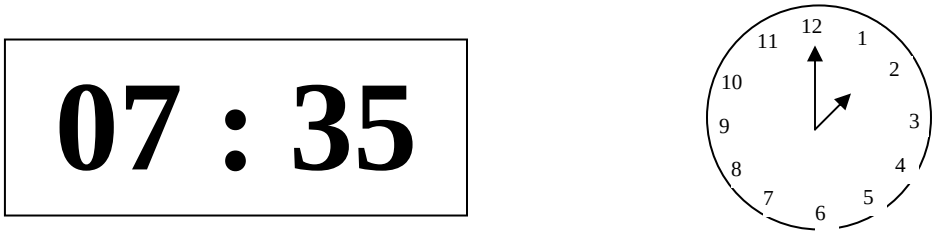

Figure 1 Digital and Analogue Clocks

Both subclasses have inherited the basic behaviour and characteristics of clocks but each has its own way of displaying time. A subclass of a class is said to **inherit** the behaviour and characteristics of that class. The subclass can add to its inheritance and can even "override" part of that inheritance by defining a different response to some message that is known to the parent class.

Inheritance is a powerful programming tool. It also supports the reuse of software components. A class is the ultimate reusable component. It can be reused directly if it fits exactly into a new program that is being constructed. If it nearly fits, it can still be reused, by defining a subclass and then making the necessary changes in the subclass to make it fit exactly.

A good illustration of class hierarchies – class, subclass, sub-subclass, et cetera – is a simple drawing program that lets its user draw lines, rectangles, polygons and curves in one thickness of brush on a screen. Each visible object on the screen could be represented in software by an object in the program. There would be four classes of objects in the program, one for each **type** of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on.

What is the relationship between these classes?

All of these classes represent "drawing objects". They would all know how to respond to a "draw yourself" message.

What is the relationship between the classes line and rectangle?

Lines and rectangles need only two points to specify them - a line its two end-points and a rectangle its top-left and its bottom-right corners.

What is the relationship between the classes polygon and curve?

Both classes represent multiple-point objects.

We can now see that there is a hierarchy of classes belonging to the simple drawing program. Two-Point Object and Multiple-Point Object are subclasses of the class Drawing Object. The classes Line and Rectangle are subclasses of the subclass Two-Point Object. The classes Polygon and Curve are subclasses of the subclass Multiple-Point Object. The class relationships can be shown on an inheritance diagram - see Figure 2 on the next page.

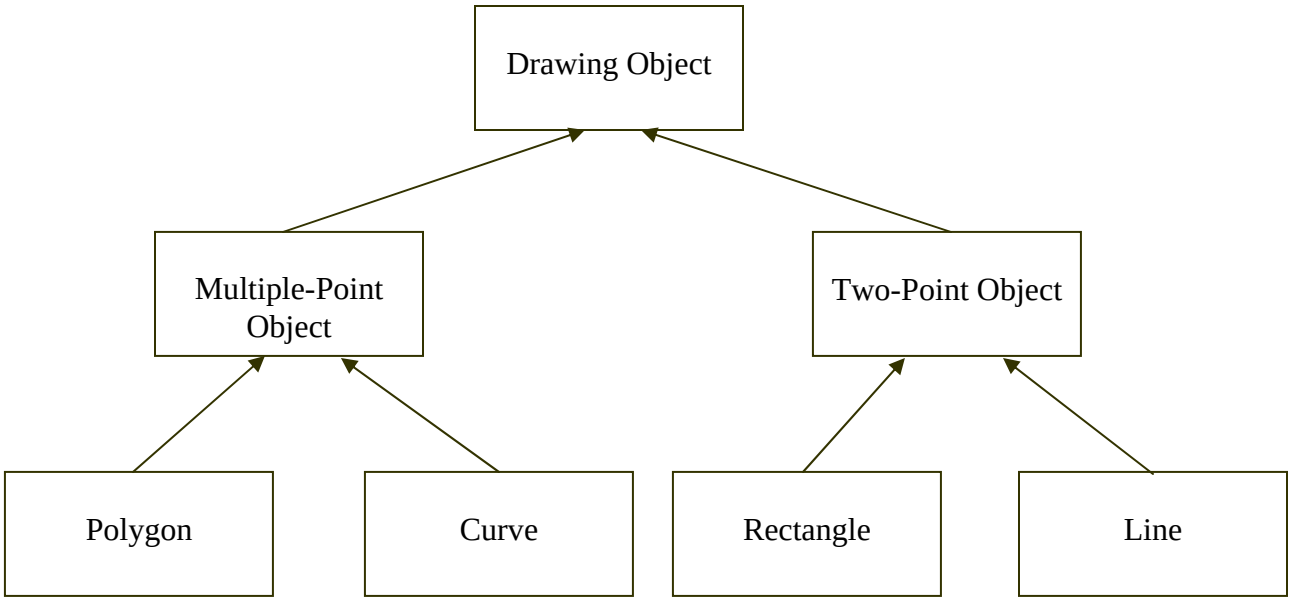

Figure 2 Inheritance Diagram

Reusability

A main platform of object-oriented programming is code reusability. Rather than starting from scratch with each new application, a programmer will consult libraries of existing components to see if any are appropriate as starting points for the design of a new application.
These components will exist in libraries as class definitions. A programmer will select an appropriate class definition from a library and then create a subclass for the application. The subclass will inherit the methods and properties of the library class, add some new ones of its own and possibly redefine the actions of others.

Reusability is the ability of software elements to serve for the construction of many different applications.

In visual programming languages such as Delphi, Visual C++ and Visual Basic the libraries store the class definitions for components which allow Graphical User Interfaces (GUI) to be built. The class inheritance diagram shown below illustrates just one such library.

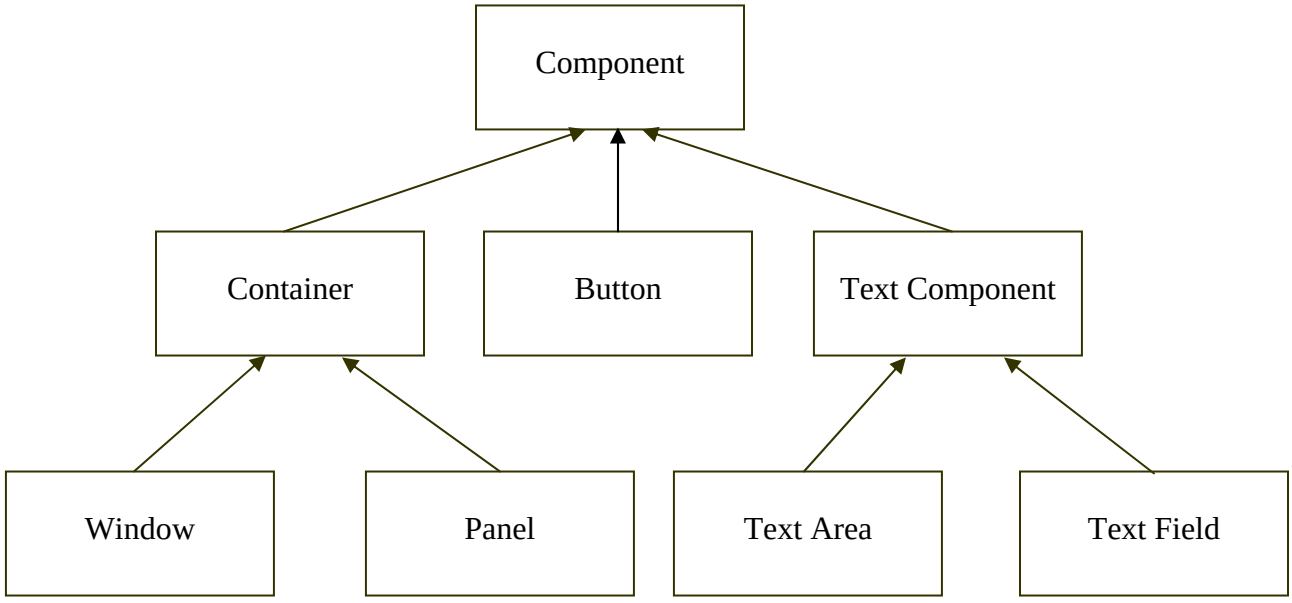

Figure 3 Component Inheritance Diagram

The popularity of OOP stems from the support it gives to a software development process that relies upon pre-existing reusable software components.

The spirit of reusability requires that amongst programmers a culture prevails in which software is developed under the assumption that it will be reused.

Benefits of Reusability

- **Reliability**. Components built by specialists in their field are more likely to be designed correctly and reliably. The reuse of these components across many applications will have given the developers of the components ample feedback to deal with any bugs and deficiencies.
- **Efficiency**. The component developers are likely to be experts in their field and will have used the best possible algorithms and data structures.
- **Time Savings**. By relying upon existing components there is less software to develop and hence applications can be built quicker.
- **Decreased maintenance effort**. Using someone else's components decreases the amount of maintenance effort that the application developer needs to expend. The maintenance of the reused components is the responsibility of the component supplier.
- **Consistency**. Reliance on a library of standard components will tend to spread a consistency of design message throughout a team of programmers working on an application. The library is the basis of a standard that will lend coherency and conformity to the design process.
- **Investment**. Reusing software will save the cost of developing similar software from scratch. The investment in the original development is preserved if the developed software can be used in another project. The most reusable software tends to be that produced by the best developers. Reusing software is thus a way of preserving the knowledge and creations of the best developers.

Three main properties characterize an object-oriented programming language:

- **Encapsulation**
  - Combining a record with the procedures and functions that manipulate it to form a new data type-an object.
- **Inheritance**
  - Defining an object and then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors' code and data
- **Polymorphism**
  - Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.