UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture -1: Some Perspectives on Object Orientation

"We must never forget the essential limitation of a machine:

It cannot create a thought, or make a snap estimate, or follow a hunch.

It can only do precisely what it has been instructed to do."

Quoted in The Computer Bulletin September 1998 from an

edition of the same journal 40 years earlier.

Introduction

This brief lecture provides some "background" to the course proper and seeks to demonstrate how the notion of *object orientation* can be placed in a variety of different perspectives.

-1.0 Simulations

Discrete-event simulation provides a means of simulating the operation or behaviour of components at a higher level of abstraction than those same operations or behaviours are simulated in a continuous simulation[1]. In a discrete event simulation, an "incident" causes the system modelling the simulation, e.g. a software system, to change its *state* in some way. A simple example of such an "event" occuring is when a component in the simulation generates "output". A *sequence* of such events provides a sufficiently accurate *dynamic* model of the system being simulated for many purposes. Events in a discrete-event simulator can occur only during a distinct unit of "time" during the simulation.[2]

A simple arrangement of logic gates, i.e. a *logic circuit* provides an example of a discrete event simulation.

[pic]

If the NAND gate has a delay of two "time units" and the NOT gate has a delay of one "time unit", then the logic circuit above will produce the outputs shown below when supplied with the specified inputs. (It is assumed that prior to the simulation, both gates are generating undefined outputs represented in the diagram by the horizontal dashed lines).

[pic]

A discrete event simulation proceeds (executes) by performing events on its "internal model" of the system being simulated. In an imperative language each event is a procedure; in a functional language each event would be a function which is applied to the internal model to produce a new internal model. Typically, the internal model of the system being simulated is realised as a data structure constructed from the simple types and type constructors provided by a programming language. The language Simula provides support for writing discrete event simulation programs via its *simulation class*[3].

See: http://www.dcs.ed.ac.uk/home/rjp/bookhtml/chap19.html

for an example of a simple simulation written in Simula.

-1.1 Simula and Object Orientation

For (obvious?) reasons Simula is often referred to as the "first" object-oriented programming language. A justification for this supposition lies in the language's embodiment of the notion of a *class*, i.e. an means of defining a collection of "behaviourally" or "structurally" related *objects* (or class instances).

See: http://www.dcs.ed.ac.uk/home/rjp/bookhtml/chap09.html

for a more detailed explanation of Simula's class construct.

However, Simula's support for the notion of a class may be seen as limited in the context of more recent object oriented programming languages, e.g. Eiffel, SmallTalk (the language) and C++(?), etc.

-2.0 Other Prespectives on Object Orientation

"Clarity is our only defence against the embarrassment felt on the completion of a large project when it is discovered that the wrong problem has been solved."

C. A. R. Hoare

One example of a problem which remains to be solved completely lies at the heart of what has become known as "software engineering" and other related activities, namely, that *"as the size of software increases the number of potential interactions between components increases exponentially until they cannot be understood, maintained or documented effectively. This effect dominates the cost of software and limits its applications".*

One approach which has proved relatively successful and which is applicable to a wide variety of of programming language styles including functional, logic and other programming language paradigms is to provide support for developing complex software systems by systematically **engineering** them from smaller simpler components. This approach can supported without direct recourse to the object oriented paradigm, i.e. by an extension to a programming language typically termed a *module*. Although, a *module* is a greatly simplified realisation of a more general *class construct*, it can be used in a variety of different ways, including as a means of providing a library of pre-developed software components, as a means of *encapsulating* the *state* of an "object" and also to provide a realisation of an *abstract data type*.

The significance of the last sentence lies in the terms used, in particular, the terms *encapsulation*, *state* and *abstract data type* which are, as we shall see in later lectures vital notions which underpin what has become known as the object oriented programming language paradigm. Of course, other notions are also vital to the object oriented language paradigm, in particular the *polymorphism* in its various guises.

-2.1 Types and Type Systems

A program written in some programming language exploits the *type system* provided by that programming language in order to do something useful. Typically, the types provided by a programming language include simple "built-in" or pre-defined types such as the types integer, Boolean, char etc, and also some means of constructing "user defined" types via *type constructors*, e.g. cartesian products (records), disjoint unions(variants), arrays, etc.

The ease with which a data structure <u>and</u> its associated operations can be realised is one useful measure of the expressive power of a given type system, thus for example, it is rather more difficult to build a data structure and associated operations to support a parse tree representation of a COBOL program using COBOL itself than it is to construct a data structure and operations in, say, Pascal <u>because COBOL's type system is unsuitable (insufficiently powerful) for that application.</u>

The very nature of type systems, in particular the way they seem to be bound to programming languages (and of course specification languages) means that they are difficult to reason about in isolation, hence, much of the research that has led to "better" programming languages has its theoretical basis in the need for ever more powerful type systems and includes the development of object oriented programming languages.

-2.2 Specification, Design and Implementation

The need to support alternative forms of description for software systems (to satisfy different requirements and constraints) has resulted in a proliferation of notations, notation techniques, and languages for specifying, designing and implementing software systems.

The very nature of a specification, i.e. that ideally it has a single unambiguous meaning, means that it needs to be written in a formal language, similarly an implementation must be written in a formal language (a programming language) in order that it can be translated into an executable representation. Interestingly, some components of a specification can be written such that an implementation which satisfies the specification can be derived <u>directly</u> from the specification automatically. Software tools which support the automated translation of specifications into implementations provide a means of raising the level of abstraction at which some problem and its solution (set) can be reasoned about and also reduce the cost of developing a system.

Software design techniques tend to a less formal approach with corresponding benefits and disbenefits.

There are object oriented "versions" of specification notation techniques, e.g. Object-Z, VDM++, object oriented design languages e.g. Booch, and also versions of "existing" programming languages, e.g. Pascal, C, Prolog (and even COBOL!) in addition to more recent languages designed specifically to elegently embody the object oriented paradigm, e.g. Eiffel, SmallTalk, Java(?).

-3.0 Summary and Conclusions.

This brief lecture has examined some different examples of perspectives on object orientation. There are, of course, many other alternative perspectives, including theoretical perspectives, e.g. as an extension of the notion of a "record type", "applications oriented" perspectives, e.g. "business objects", etc. It is part of your task as a student to ensure that you read the literature and gain as wide an understanding of how we "arrived" at the object oriented paradigm as possible.

The remainder of this course will examine further some of the notions identified in this lecture, introduce other fundamental and general notions, and also explore the ways in which we can actually <u>develop</u> software systems in an object oriented manner.

Chris Harrison, September 1998.

Appendix A

A Comparative Overview of Languages

One estimate of the number of programming languages that have been developed suggests that it exceeds one thousand. Most of these languages have been "experiments" in language design and implementation, but many have also become widely used in commerce and industry. It is inevitable that many more languages will become available in the future, not least because all languages are a compromise between several often conflicting requirements and constraints, and also because the is no "ideal" programming language, i.e. there is no language which is universally applicable to all possible computations.

The overwhelming majority of languages embody the imperative (programming language) paradigm, i.e. they embody the notion of (shared[4]) variables or attributes whose value is changed via assignment statements. Until the mid 1980's the majority of widely used imperative languages were procedural, i.e. the language supported the notion of procedural abstraction as a mechanism for organising and managing complexity. Some procedural languages were "extended" (always a bad idea.) to support the object-oriented features, e.g. C -> C++, Pascal -> Delphi, Ada -> Ada95, etc). A more radical approach was to develop an entirely "new" language (actually to design and implement a language based on exiting notions but to embody/support these notions more regularly), e.g. Eiffel and Java.

Other approaches to the same fundamental problem (shared variables and the complexity of their interactions) led to alternative language paradigms, e.g. functional (Lisp and ML) and logic (Prolog).

Now there are many possible perspectives on the reason that languages are developed, for example, from a theoretical viewpoint a programming language is an "experiment" and hence many such experimental languages never become widespread in their use (they remain within the community that developed them). Alternatively, a programming language can be seen as a practical tool for describing computations such that when translated and executed the resulting software is reliable and robust (we hope!).

The <u>best</u> known way of ensuring that a programming language enables reliable and robust software (and larger-scale software systems) to be developed is to ensure that the language is *strongly typed* and that the majority (or all) *type checking* is done at compile time[5][6]. Other considerations (which I suppose are important, include ease of use but I don't know how to measure such notions accurately), support for potential re-use (note the use of the term *potential*), etc.

Language designers vary in their ability to ensure that the languages they develop exhibit fundamental and general properties, i.e.

? Expressive power

? Simplicity due to orthogonality

? Engineered implementations with integrated support tools

? Effective and comprehensive error detection and support for correction

? Completeness (computationally) and adherence to "standards" (whatever they are.)

A.1. Expressive Power

There are many measures of *expressive power* ranging from the vague (e.g. "solutions can be expressed in terms of the problem being solved rather than in terms of the system used to implement the solution") to the mathematically exact (where type systems are compared directly using mathematics and logic to determine what can and cannot be described using the language). A reasonably effective measure of the expressive power of a language is provided by an examination of the types (i.e. pre-defined and type constructors) provided by the language and how associated operations are performed upon values of pre-defined and user-defined types. Modern languages provide direct support for realising new (user-defined) types as *abstract data types*, e.g. as a *package* in Ada, or a *class* in C++ and Java. This form of support enables such languages to have libraries of pre-developed types which can be "standardised" and which can be used directly to implement other types required by a software developer.

A.2. Simplicity Due To Orthogonality

I'm not sure how to measure simplicity (in the context of a programming language) but the notion of orthogonality is rather more precise, i.e. an orthogonal language enables any combination of its language constructs to be to used and, hence, such languages have few basic constructs, e.g. Algol68 and SmallTalk (the language). However, the sheer scale of software, especially larger-scale software systems, means that developing non-trivial applications in pure orthogonal languages is problematic - not necessarily intractable but it requires a discipline on the part of the software developer that is very rare.

More generally, language constructs should be consistent, i.e. the fundamental and general principles that underpin "good" language design should be strictly adhered to:-

? Type completeness principle

? Correspondence principle

? Abstraction principles

A.3. Well-engineered Implementations

It is impossible to overemphasise the need for language implementations to be well-engineered, indeed, it is because such implementations must be reliable and robust that many of the (actually very few in number) techniques that are know for actually engineering software have been developed (e.g. *program design by recursive descent* and hence *data structure driven software design*).

Software developed using an imperative language is typically compiled (i.e. the *source description* is supplied to a compiler) and the resulting *execution representation* (usually and erroneously referred to as "code") is then executed on some machine. This need to support effective compilation is a major constraint upon a language's design (especially where people are obsessed with the "speed" of the resulting "code" whatever that is.) most significantly because languages only become widely adopted if they can be effectively translated and the resulting execution description can be executed "efficiently".

Alternatively, a language can be interpreted in the sense that it is "directly executed" by an interpreter, although in practice this does not occur and an intermediate description is derived from the source description and this intermediate description is then executed. One means of enabling the execution of the intermediate representation is for the interpreter to be a virtual machine, i.e. a software implementation of a processor. Although in practice such an approach is "slower" that the alternative approach of compiling (well, that depends what you mean by slower.) it has several important advantages including support for interactive software development, decoupling of the intermediate representation from its execution "engine", i.e. it may be executed on different machines provided that they implement exactly the same semantics, and other more subtle notions that you may seek to discover yourselves.

A.4. Error Detection and Correction

All software must satisfy some preceding specification, and ideally this specification is itself written in a formal language. In practice however, most software is simply written without regard to its need to be correct with respect to a formal specification - although increasingly even the most mundane applications must be reliable and robust and the most lazy of software developers are required to learn and adopt the notion of formally specifying software rather than simply writing it or developing some informal "design" (whatever that is) and trying to adhere to it.

As a result of the generally lazy attitude of software developers with regard to (formally) specifying their software prior to its construction, it is necessary to exhaustively test the resulting software to try to give some kind of assurance that it is free of errors (c.f. "testing can only show the presence of 'bugs' and not their absence" with acknowledgements to Dijkstra). Thus, if we wish to adopt the approach whereby we test software it is useful if the language provides some direct form of support for "testing", **but the best form of testing is simply to ensure that the program (or larger-scale software system) is syntactically-correct (and partially semantically-correct) at compile-time.**

A.5. Completeness and Adherence to Standards

Any language designer who knows what he or she is doing will ensure that a language is computationally complete (try expressing a transitive closure directly in SQL!) so we hope that we have complete language in the computational sense. Standards are, of course, only as good as the people who write them and may have the adverse effect of precluding small but important changes that make a language "better" - so if you wish to develop a language and have it widely adopted you may well need to have a standard for it (although this will inevitably go though several versions) and then all software developed using this "standard" will be guaranteed to compile and run **without any change whatsoever** by a standard compiler and run-time system (Oh Yeh, I hear you say.)

A.6. An Historical Perspective

We can see a simple linear development for a language, e.g.

[pic]

Alternatively we can associate the development of languages in "time", e.g.

[pic]

Unfortunately, even the most complete "historical" description simplifies what has happened in that it does not show the complex interplay between ideas and notions developed by language developers as they have attempted to develop "better" languages.

Thus, we need to examine significant language developments in detail in order to understand and appreciate their importance. Perhaps the single most important language to date is Algol(58 then 60) not least because it still has ramifications for even the most "modern" programming languages. Its later/parallel developments (Algol-W and then Algol-68) were less "successful" but it is possible to abstract over such problems and show a direct lineage (albeit as a simplification) from Algol-60 to Oberon (a "laboratory" language), and also to Ada, Delphi and C++ and Java, e.g.

[pic]

A.7 A Brief Consideration of Past Languages

The significance of past programming languages can sometimes be lost to those who have not used them, especially those who did not use them when such languages were literally the "state-of-the-art". Whilst is also important to appreciate the kind of computing machines that were available during the heyday of such languages, it must also be remembered that the "better" languages sought to abstract away from the limitations of such computing machines and to be implementable on a variety of different platforms.

First, one must consider the late 1950's and early 1960's - a period of fundamental change in both computing machines and the design of programming languages. At this time, there were several hardware manufacturers who vied with one another for a small but growing and lucrative market in commercial and especially scientific computing. The advent of more "powerful" hardware (which seems puny by the standards of even today's desktop machines) provided the impetus for language designers to consider entirely new kinds of language, but several existing languages were well established and their users were loathe to learn another way of expressing their computations. The two most commonly available languages at the time were COBOL and Fortran which were intended for commercial and scientific use respectively. A committee (usually a very bad idea as a basis for effective language design) was set up to consider the notion of a "common language" that would be suitable for a wider variety of applications. Fortunately, the committee[7] included a few members whose ability enabled them to radically change the way in which a language was defined - and this led to the design and implementation of Algol.

Whilst the first description of the language remained unimplemented (i.e. Algol58) the criticisms it caused enabled the designers to identify and resolve significant problems and to produce another report that defined the programming language Algol60 (and finally a revised report in 1963)

Algol60 was intended to embody the following notions:-

? A "standard" mathematical-like notation for expressing a computation

? Suitable for expressing computations in a form which enabled algorithmic processes to

make clear to the reader of the program without extensive additional natural language descriptions, i.e. suitable for communicating algorithms between (skilled) people.

? Implementable - it should be possible to translate the language into a form that enabled programs to be executed on a variety of different machines. Although Algol was not widely used in the largest marketplace, i.e. the United States where Fortran and COBOL remained pre-eminent due to proprietary support, it was never-the-less a vital step in the development of programming languages.

Reasons why Algol was such an important development include:-

? Its definition. Algol was the first language to have its syntax formally defined, and this involved the use of a new notation technique called BNF. The importance of this change in the way that a language's syntax is defined had many ramifications, for example, it made it possible to use the definition to organise the front-end of the language's compiler to control the compilation process - so-called syntax-directed compiling, and this led directly to very important techniques for ensuring that compilers were properly "engineered" (cf. Recursive descent parsing, data structure driven design etc.)

? Its structure. Algol is a (nested) block-structured language with scoped declarations. Other languages had only global scope (e.g. COBOL) of linear scoping mechanisms (e.g. Fortran) which made certain computations difficult if not impossible to implement directly.

? Its support for conformant (i.e. dynamically sized) arrays.

? Its support for "structured programming"[8]. Algol provided several well-defined and complementary control constructs, and those constructs which were already provided by other languages were more advanced in Algol's implementation (e.g. IF/THEN and FOR statements)

? Its support for the direct description of recursive computations. Algol60 was the first language to support procedures which may call themselves either directly or indirectly, and this enables a variety of algorithms to be expressed very elegantly including algorithms for parsing strings in the language itself.

A short Algol program below provides some indication of its relationship syntactically and semantically to many other languages that have adopted its basic notions:-

```
begin
    integer n;
```

```
    read(n);
    begin
      integer i, sum;
      sum:=0;
      for i:=1 step 1 until n do sum:=sum + i;
      write("sum of 1 to n is: ", number
    end
end
```

A.7.1 Pascal

By the late 1960's Algol had developed via Algol-W and then Algol-68 into a more sophisticated language with record types (i.e. a cartesian product type), a CASE statement, a corrected FOR statement and additional WHILE statement, support for calling procedur and function parameters by value, result and also value-result (Algol60 only supported call be name), long(yuk), real(yuk) and complex data types, a bit data type (yuk) and some intrinsic string processing operations. Most importantly, later versions of Algol supported embedded assertions.

A direct development from Algol-W was produced by Wirth as a vehicle for teaching a systematic approach to programming. The language was intended to be compact, efficient and suitable for a wide range of potential applications. At about the same time, a vast increase in the use of programming languages in colleges and universities made the resulting language, called Pascal, very widely used by both students and academics.

Later implementations of Pascal, e.g. the widely used UCSD implementation, provided support for modular abstraction (i.e. a module construct) which enabled much larger scale software systems to be effectively constructed. The elegance of Pascal is best demonstrated by its suitability for constructing a compiler entirely in the language itself - and hence the UCSD implementation which was almost entirely written in Pascal.

The general structure of a recursive descent parser written in Pascal for processing strings in Pascal is shown below:-

```
PROGRAM parser;
     TYPE symbol = (sym_program, sym_begin, sym_end, sym_const, sym_type,
sym_proc, sym_func,
                   Sym_open, sym_close, sym_comma, sym_colon, ..
                  );
    VAR current: symbol;
    FUNCTION first_declaration(current: symbol): Boolean;
    BEGIN
      first_declaration:= .
    END;
   .
   .
   .
   .
   PROCEDURE rec_block(VAR current: symbol);
        PROCEDURE rec_declaration(VAR current: symbol);
        BEGIN
          .
          .
        END;
```

```
    BEGIN
      .
      .
      .
    END;
BEGIN
  read_sym(current);
  IF current = sym_program THEN rec_block(current)
  ELSE error('PROGRAM expected')
END.
```

A.7.2 C

By the late 1960's programmers concerned with what was termed "systems programming" remained convinced that execution efficiency was the overriding concern for the software systems that they developed. The very nature of the classic(al) systems program, i.e. an operating system, meant that efficiency considerations included considerations of multi-processing/tasking, concurrency and synchronisation, etc. Any "team" of systems programmers who considered themselves worth of the name consisted of members who still preferred to implement directly in assembly language, and others who had decided to develop their own higher-level language. The resulting plethora of systems programming languages included some interesting examples of the kind of thinking that goes on in a systems programmer's mind.

The UNIX operating system was written in a language that evolved throughout the development process, from a language called BCPL, then B, and for obvious (and presumably what were thought to be rather witty reasons) C. Because BCPL and B were not (strongly) typed languages, C is weakly typed (since it does have types). Most of the later developments to C including the advent of C++ have been attempts to "sort out" its typing, i.e. correct its type system in order to support better type checking.

Those who "like" C usually allude to its combination of high-level and lower-level facilities, and hence its suitability for generating "efficient code". Detractors point out its arcane syntax and the difficulty of ensuring that programs and especially larger-scale software systems written in C are robust and reliable, indeed, it can be argued that the language is inherently "unsafe" due to its poor type system and the ways in which it can be "misused", and hence, that it is intrinsically unsuitable when "engineering" software.

(I vowed many years ago to refrain from using C hence the lack of an example here.)

A.7.3 Modules, Abstract Data Types and Classes

The distinction between the representation of the logical components in a software system and the corresponding source descriptions led to the identification of the need to be able to separate these two concerns in a way that enabled them to be usefully combined at a later stage, i.e. it because clear that it was necessary to provide direct support in programming languages for modules whose two "parts" comprised a *visible component* (i.e. the declarations within the visible component are available to other modules which reference that module) and a *hidden component* (i.e. the declarations within the hidden component are not visible outside the scope of the modules hidden part). These two components are typically (though not always) bound together as part of the same source description and termed a module (or unit, or package etc depending on the language).

The more subtle distinction between a *type* and a *type constructor* led to the need to provide a more powerful mechanism than that supported by a conventional module construct, and hence the notion of abstract data types (as realised by modules), and latterly the notions of a class construct, inheritance, method interfaces, messages etc, which underpin the object-oriented programming language paradigm.

The module shown below, written in a hypothetical language with a Pascal-like syntax[9], defines (part of) a parametric `stack` type.

```
MODULE stack_definition{item_type = integer};
```

```
  INTERFACE
    TYPE stack = HIDDEN;
    PROCEDURE empty;
    FUNCTION  is_empty(s: stack): Boolean;
    PROCEDURE push(item: item_type);
    FUNCTION  top: item_type;
    PROCEDURE pop;
  IMPLEMENTATION
    CONST max_cardinality = 10;
    TYPE  stack   = ^element;
          element = RECORD
                      this: item_type;
                      next: stack
                    END;
    .
    .
END.
```

[pic]

----

[1] The terms continuous and discrete (or discontinuous), denote fundamental ways of reasoning about abstract and concrete "things".

[2] Other forms of simulation are, of course, possible, for example, Monte Carlo simulators usually make extensive use of random number generators in order to simulate the desired system. Unlike discrete-event simulators, which are often used to model deterministic systems, Monte Carlo simulators can be used to effectively model systems in which *probability* and *nondeterminism* plays a major role.

[3] Simula was developed as a language with features which allowed certain kinds of simulation to be performed more easily on computers. These features were not then available in other languages. Some, like hierarchical types and virtual specification, are still almost unique to Simula. However, Simula was intended as a general purpose language, i.e. it's authors always aimed at a design with the widest range of use.

[4] And therein lies a fundamental problem, i.e. as the number of potential interactions between such data objects increases it becomes difficult even to document them, let alone reason about all of the interactions. This is the problem which the discipline of software engineering seeks to address, i.e. the "COMPLEXITY PROBLEM" limits the uses that software can be effectively put to.

[5] Hence the need to have well-defined *type systems*, and the development of languages like C#, etc

[6] But some language paradigms result in the need to do some type checking at run-time, e.g. object-oriented languages have dynamic binding and this requires some type checking to be done during execution.

[7] The committee for the revised report comprised J. W. Backus, F. L. Bauer, C. Katz, J. McCarthy, P. Naur, J. Perlis, H. Rutishauser, K. Samuelson, B. Vaquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger. You may recognise some of these names.

[8] Unfortunately, for reasons that the committee were unable to control, the language included a GOTO statement which is hardly conducive to structured programming.

[9] Several "liberties" have been taken in this example to keep its representation simple. These include the notion of "type expressions" as formal parameters in a module construct, an "extended" semantics for USES declarations, etc. In general, if an abstraction, e.g. a module abstraction, is parameterised with respect to a *value* it can use the argument value even if nothing is known about the value except its type. Similarly, if an abstraction is parameterised with respect to a *variable* it can inspect and update the argument variable even if nothing is known about the variable except its type. However, type parameters are fundamentally different because they denote an possibly unknown argument type, i.e. nothing useful (e.g. type checking) can be done with the type parameter until *something* is known about the argument type, more specifically, what operations are applicable to values of the argument type.