UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 0: Introduction and Background

"Nothing requires a rarer intellectual heroism than

willingness to see one's equation written out."

George Santayana

This course is about *object orientation*. If we could capture the "essence" of object orientation in a single statement it might take the following form:-

abstract data types + polymorphism = object orientation [1]

The advantage of this particular statement is that it embodies a particularly "high level" view of the whole notion of object orientation and expresses it in the form of a simple equality. We can use such a statement as the basis for furthering our understanding by ensuring that we understand the notion of *abstract data types* and of *polymorphism* and how these two notions can be usefully combined together.

How, then, do we ensure that we understand the notion of object orientation ? Well, first we should ensure that we understand the notion of a *data type* in general and of an *abstract data type* in particular. Secondly, we should ensure that we understand the notion of *type systems* in general, *type checking* in particular and how the limited expressive power of *monomorphic* type systems has led to the development of more "powerful" type systems based on the notions of *overloading*, *polymorphism* and *inheritance*.

Values, Types and Type Systems

"Yes, ... but what type is it ?"

A question often usefully asked of someone

learning to program who has a "problem".

In "computing" we find it particularly useful to group *values* together into *types* because it allows us to abstract over individual values (in a group) and to develop operations that apply to whole groups of values, i.e. we insist that all values of a type exhibit uniform behaviour under operations associated with the type[2]. In doing so, we inevitably characterise a type in two ways, first, by defining the set of values that denote the type and secondly by defining the operations on those values.

Three kinds of type are usually provided by formal languages for expressing computations, i.e. programming languages. *Primitive types,* as their name suggests, have no "internal structure", i.e. they are "atomic", and cannot therefore be decomposed into simpler values. *Composite types,* as their name suggests, are types whose values are composed of other types (either primitive or structured). *Recursive types*, as their name suggests, are types whose values may be composed from other values of the same type.

Having grouped values into types we then (hopefully) impose a *type discipline* that ensures that semantically meaningless operations may not be performed[3]. This involves *type checks* on *operands* in *expressions*. Where operands are composite types we must check both the forms of the composition and the types of the individual components. One key aspect of type checks is when they are performed. One possibility is to check at "compile time" and another possibility is to check at "run time" and hence the associated notions of *static* and *dynamic* type checking.

It is from an understanding of the notion of a type system that we can develop fundamental and general principles[4] which aid in the design of formal languages for describing computations. Thus, for example, the *type completeness principle* can be given as:-

"No operation shall be arbitrarily restricted in the types of the values involved"

The designer of a formal language will develop a language with such principles in mind. In the case of the type completeness principle, for example, adhering to this principle will raise the *expressive power* of the language since arbitrary restrictions, in general, tend to reduce a language's expressive power.

Abstraction, Types and Abstract Data Types.

'"Don't tell me what it is, tell me what it is supposed to be..."

A plea often usefully made of someone

learning to program who has a "problem".

Thus far, we have considered types in the context of formal languages for expressing computations, i.e. in the context of programming languages, and in particular the type systems that programming languages embody.

We can abstract away from considerations of programming languages by considering types in a more general context, i.e. as *abstract data types*. In this context, an abstract data type may be realised using the types provided by a programming language but is independent of any particular programming language realisation. One means of defining an abstract data type is to specify the type *algebraically*. In such a description an abstract data type is a "module" that specifies the *values* and *functions* belonging to the type.

A correctly written algebraic specification is a maximally abstract description of the behaviour of an abstract data type, i.e. additional definition does not add to (and may even subtract from) the meaning of the type. However, the behaviour of *elements* of an abstract data type and of *objects* is characterised by *state changes* and these state changes may affect the *value* given by operations of the type. One example of this phenomenon is where the *value* of an object (on some attribute) depends upon the history of the operations applied to that object and not, simply, on the structure of the object itself.[5]

A fundamental distinction can be made between *types* and *classes*, i.e. types may be used as *predicates* when type checking whereas classes arose from the need to have some means of enforcing a "structure" upon the creation of objects. However, we can associate the two notions by regarding a *description* of a class as implicitly characterising the type of the instances (objects) of class !

Polymorphism

"I have a 'bug' in my program..."

Simultaneously a statement of fact and a plea for assistance

often made by someone learning to program".

The notion of a type system as having a single "form", i.e. the type system is *monomorphic*, implies that every "entity" describable within that system, e.g. every *constant*, *variable*, *parameter* and *function result*, has a unique type. In practice, the type systems embodied by programming languages are not strictly monomorphic because they include types that cannot be expressed directly in that language's own type system[6]. Such types must, as a consequence, be treated differently by the compiler for that language.

In programming languages in general, the notion of *overloading* enables a (hopefully) small number of distinct abstractions to have the same *identifier* even though these abstractions do not need to have related types nor do they need to perform similar operations upon their arguments[7]. Conversely, the notion of *polymorphism*, in general, is that a single abstraction embodies a (hopefully) large collection of related types and operates uniformally on its arguments irrespective of their individual type.

A programming language with a truly (intentionally) polymorphic type system need not necessarily be an object oriented language since the notion of *parametric polymorphism* may be supported by any programming language in which a function's type may be defined in terms of a *type variable* rather than in terms of a unique (specific) type.

The kinds of polymorphism associated with (intentionally designed) object oriented programming languages include overloading (ad-hoc polymorphism) and parametric and inclusion polymorphism (inheritance).

Beyond Programming Languages

"Where is your specification or design for this program ?"

A question often usefully asked of someone *learning to program who has a "problem".*

The notion of object orientation is not exclusively confined to considerations of programming languages, not least, because the development of a software system will (usually) require a variety of different kinds of description to be developed. The notions of a software specification, a requirement's specification, a software design, a user-interface design, etc., provide alternative contexts within which object orientation may be used as a means of reasoning about some "thing" and representing that "thing". Underpinning such notions is the (not unreasonable) hope that a description arrived at by exploiting concepts from the object oriented paradigm will (ultimately) be easier to translate into a program or larger-scale software system written in an object oriented programming language.

The means by which a translation from one description into another may be made depends upon the languages or notation/techniques used. Where a *formal language* is used it may be possible to develop a finite set of *rules* that define how all descriptions written in the "source" language may be systematically translated into descriptions written in the "target" language and hence the translation process may (ultimately) be "automated". Where the language or notation techniques used are "informal" it will (usually) not be possible to do the same and a single description may be open to a variety of alternative interpretations.

What Is There To Learn ?

"Well, that's finished then..."

A statement often made by someone learning to program who has just managed to get their program to compile successfully.

The is indeed much that can usefully be learned about object orientation. The purpose of this course is to develop an understanding of what object orientation is and how it may be applied. To this end, this course will examine object orientation in a variety of different contexts, more specifically, in the context of programming languages, of software design languages and notation techniques and of specification notation techniques.

Chris Harrison, January 1997.

References

Programming Language Concepts and Paradigms

David A. Watt, Prentice-Hall International Series in Computer Science

Principles of Object-Oriented Software Development

Anton Eliens, Addison Wesley

Object-Oriented Software Construction

Bertrand Meyer, Prentice-Hall International Series in Computer Science

----

[1] With acknowledgement to Wirth, see *"Algorithms + Data Structures = Programs"* .

Prentice-Hall Series in Automatic Computation, 1976 (page xv) in which he states:-

"Programming is a constructive art. How can a constructive, inventive activity be taught? One method is to crystallise elementary composition principles out of many cases and exhibit them in a systematic manner. But programming is a field of vast variety often involving complex intellectual activities. The belief that it could ever be condensed into a sort of pure "recipe teaching" is mistaken. What remains in our arsenal of teaching methods is the careful selection and presentation of master examples."

[2] Thus the notion of *type* in this context is an example of *abstraction* applied to reasoning about collections of *values*.

[3] It makes no sense, i.e. is semantically meaningless, to "subtract" the Boolean value "true" from the Boolean value "false.

[4] See also:- abstraction principle, correspondence principle, qualification principle, etc.

[5] Consider, for example, a "system" abstract data type that abstracts over the facilities provided by a conventional programming language implementation for collecting together related components of a software system. One useful property that such an ADT might embody is that a "system" may not be "deleted" until all the components in its substructure have themselves first been deleted.

[6] Typically, input-output operations are both parametric and polymorphic in that they may take variable numbers of arguments and arguments of different types. How does the compiler type check statements containing calls to such operations ?

[7] Overloading is a means of "relaxing" typing constraints within certain well-defined limits, i.e. we must (ultimately) be able to determine the type(s) of the values involved.