UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 8: Software Specification: Initial Considerations

Introduction

This lecture is concerned with the notion of software specification. First, the fundamental and general notions of an *algorithm*, a *computation* and a *specification* are examined and related to one another. Secondly, practical considerations associated with the development of specifications are examined in the context of two complementary "styles" that specifications may adopt. Thirdly, the notion of *verification* is considered. Finally, limitations of approaches to combining object oriented notions within formal methods are examined.

Algorithms, Computations and Specifications

The very notion of what precisely an *algorithm[1]* "is" led Alan Turing in 1936 to propose a definition in terms of *Turing Machines*. Other mathematicians proposed different definitions at about the same time, however, it is Turing's definition which has become the most influential because it embodies the notion of a *universal machine* which can achieve any *algorithmic action* and which led to the notion of *a general purpose computer*. A computation in this context is literally the *action* of a Turing machine, thus, whether a computation (or computational procedure) is described in a "top-down" manner (e.g. Euclid's algorithm for finding the highest common factor of two natural numbers) or a bottom-up manner (e.g. an **artificial** neural network implementing a "learning" algorithm), or even some combination of both, it is still both computational and algorithmic, not least, because each manner leads to a *solution* which can be *implemented*, i.e. *programmed for*, a general-purpose computer.[2]

Programming[3] can be characterised as a form of thinking in which the *abstractions* developed reflect an *imperative viewpoint*[4], i.e. *executional abstraction* is used to map different computations upon one another by abstracting away from the mutual differences between *members* of a class of computation[5] (by concentrating upon the properties of the *class* as a whole) and, as a result, *assertions* are made about each member of a class. In this context, programming is a form of *reasoning* in which all manipulations are (or could be) formalised by mathematical techniques such as arithmetic, formula manipulation and symbolic logic, i.e. mathematics and logic are used in Computer Science and Software Engineering for exactly the same reason as in other sciences - as a means of reasoning formally and as a means of expressing formal reasoning.

The notion of a specification has one basis in the notion of an algorithm being "knowable", i.e. the algorithm has a specification which can be *achieved in practice*. In principle, any formal system, e.g. a Turing Machine, can be specified, in practice (because humans have finite abilities) there will be some systems which are beyond human specification![6]

Practical Considerations

In practice, specifications associated with software components or collections of components, i.e. software systems, may be operational or descriptive in style (or some combination of both styles). Operational specifications capture the desired or intended behaviour of a component or system usually by embodying a model of some abstract device that can "simulate" that behaviour. Descriptive specifications state the desired properties of a component or system in a purely declarative fashion.

Examples:-

Operational Definition

E is the path of the point that moves such that the sum of its distances from two fixed points P1 and P2 is constant.

Descriptive Definition

ax2 + by2 + c = 0

One fundamental difference between such definitions is the use that they may be put to, i.e. the operational definition easily allows us to check whether the specification describes the desired kind of curve when we gave the specification (by experimentation we can use the specification as a means of drawing curves). The descriptive definition allows us more easily to determine if a given point P lies on the curve.

The notion of a specification being more (or less) appropriate for some use is fundamental to the notion of specifications serving as a reference against which implementations of what was specified can be verified - and its corollary, i.e. that the specification itself must be verified[7]. There are, in general, two ways of verifying a functional specification, i.e. observing the dynamic behaviour of the specified system in order to check whether it conforms to an intuitive understanding we (hopefully) had of the behaviour of the "ideal" system, or, alternatively, reasoning about the properties of the specified system that can be deduced from the specification by comparing the deduced properties with the expected properties. Both approaches are made more effective when the functional specification is given in a formal language. It is also necessary to verify the completeness and consistency of specifications.

Examples of techniques for developing operational specifications include data flow diagrams, finite state automata, Petri-Nets, etc. Examples of techniques for descriptive specifications include E-R diagramming, Logic (constructive or model-based) specifications and Algebraic specifications.

We must also be aware that constructing specifications using any of the techniques enumerated above (or any other specification technique) will be as complex an activity as designing the implementation of what has been specified if the specification language provides no means of abstracting over and modularising larger-scale specifications.

Verification

By their very nature Humans are fallible and make mistakes. No technique yet known permits Humans to avoid erroneous results *a priori*. Consequently, the product of any Human endeavour must be verified against some description of what was intended. In principle, each of the processes which were used to derive an implementation design (and ultimately an implementation) of a software component or system and all of the products of these processes must be verified. Even verification itself must be verified in the sense that we must check if our "experiments" were themselves properly conducted, i.e. if the "experiments" were *valid*. Verification by experimentation is usually termed *testing*, especially in the context of software development.

An alternative to verification by experimentation is to develop an *abstraction* which takes the form of a *model* of some software component or system and "exercise" this model. Of course, the very nature of a model means that we may well have abstracted over (i.e. failed to include or take proper account of) some vital element or aspect of the actual component or system, yet we have the advantage, when "exercising" such models that we are abstracting over individual executions (associated with a particular corresponding test) and reasoning, in effect, about classes of corresponding executions.

Formal Methods and Object Orientation

Formal methods have been the subject of research and of practical application for many years. Practical experience has led to the recognition that formal methods must embody some means of abstracting over and modularising the specifications which result from their application. Initially, researchers and practitioners were led in the direction of object-oriented techniques because these seemed to naturally complement model-based specification languages, e.g. "Z' and VDM. Subsequently, algebraic approaches were shown to be amenable to extension via object-oriented concepts. Unfortunately, incorporating object-oriented notions by extending existing formal methods leads to complications when reasoning about the resulting specifications, and until relatively recently there was not even a formal semantics for an object oriented specification language available.

Many problems remain to be resolved, for example, the ambiguity associated with the notion of *aggregation* used in diagrammatic object-oriented (design) methods; the fundamental notions of polymorphism and dynamic binding make it difficult to "control" (in a formal sense) which "version" of a method is actually to be executed for a particular method application; the fundamental notion of inheritance causes problems because it has the potential to cause a fragmentation of a definition of a method across several classes and leads to difficulties in reasoning about its meaning.

Currently, precise definitions can be given to particular forms of aggregation and to the notion of *subtype migration*, e.g. a semantically meaningful definition of subtyping can be given which enables results to be inferred about subtype objects on the basis of theorems proved about a supertype, or, alternatively, inheritance can be relegated to a purely syntactic rôle (i.e. code reuse and sharing, and module importation).

Summary and Conclusions

This introductory lecture has examined the background from which approaches to specifying concepts drawn from the object oriented paradigm have developed. Fundamental and general notions which underpin the specification of software components and software systems have been considered and shown to be a basis for a variety of alternative and complementary notations and techniques for developing such specifications. The very nature of the notions which underpin the object oriented paradigm have been shown to cause difficulties when they are incorporated into "conventional" formal methods.

The next lecture will examine these issues, and other considerations, in more detail.

Chris Harrison, March 1997.

----

[1] For those readers who prefer a less "philosophical" treatment of the notion of a algorithm we can define the term within the more pragmatic context of five *characteristics*, i.e. algorithms are "procedures" with (1) a unique starting "step" followed by a sequence of steps executed in default order until an explicit termination step is executed or until the list of steps is exhausted, (2) each step is performable individually with finite effort, (3) each step individually is unambiguous, (4) the algorithm allows for a variable number of steps to be executed (e.g. by "looping" or "branching" through the steps listed - but **not** infinitely), and (5) when termination is reached, the execution sequence has correctly computed the desired "object" of the algorithm.

[2] The fundamental difference between "top-down" and "bottom-up" is simple, i.e. in a "bottom-up" approach some "record" of previously computed values is incorporated into subsequent computational actions.

[3] The term *programming* is used to denote a combination of human problem-solving and symbol manipulation skills. The term programming is assumed to be synonymous with the term *software development*, and the term *software* is assumed to be synonymous with the term *software system* and hence with a composition (denoted by *) of one or more *software components*, i.e.

software = software system = (software component)*

A software component may be a module, or a program, or a routine, or any other description written in a formal language, i.e. (with alternative denoted by |):

software component = module | system | program | routine | ...

[4] Classical Mathematics is concerned with the study of the structure of knowledge from a declarative viewpoint, i.e. Mathematics provides a framework for reasoning precisely with notions of 'what is', whereas Computation provides a framework for reasoning precisely with notions of 'how to', i.e. the term imperative is used in this context to denote the notion of 'how to' and not the more specific imperative programming language paradigm.

[5] In this context, a *computation* is what an *algorithm* abstracts over, i.e. an *algorithm* embodies a *class* of *computations* which may take place "under its control".

[6] There is no real suggestion that something a simple as a natural number could be "unknowable" in principle since natural numbers like algorithmic actions can be enumerated, e.g. 0, 1, 2, 3, 4, 5, ... i.e. any specific natural number will in principle be encountered eventually no matter how large that number may be. In practice there will, however, be natural numbers that are so large there is no prospect of them ever being encountered in this way, i.e. by algorithmic action.

[7] In a science, a *hypothesis* is developed in order to explain some *phenomenon*. In Computer Science, a model of a phenomenon can be represented by a *program* Creating the program such that it conforms to some specification is a sub-problem (which can be resolved by *verification*) of the larger problem of explaining the phenomenon by *testing* the hypothesis, i.e. testing the hypothesis is synonymous with testing the *execution behaviour* of the program.