1.0 Introduction

Classes are an important concept in object-oriented programming methodologies. Smalltalk-80, C++, Objective-C and other contemporary object-oriented languages all include a class construct of some sort. The intended utility of such a construct is to support encapsulation, information-hiding and specialization. Each class defines its internal state variables and external operations, which can be invoked by message sending.

Classes are a useful construct in programming for a number of reasons. Firstly, since classes separate interfaces from implementations, interfaces can stay the same even when implementations change. Secondly, since classes have their own name spaces, there are no name clashes of either variables or operations across different classes. Thirdly, inheritance and polymorphism potentially simplify coding, leading to conciseness and clarity of programs. In reality, these strong points of object-oriented languages work fairly well in small-scale programming tasks. Their benefits for medium- to large-scale software development projects, however, are not convincing at best, nor is the evidence of benefits for other aspects of software life-cycle such as understandability and maintainability.

Object-oriented software development requires more design effort in order to take full advantages of the methodologies. The primary task in designing an application in object-oriented framework can be divided into two distinct phases: identify subsystems for the application and identify object classes, their structure and behavior for implementing the subsystems. Even if a design is functionally sound and conceptually clear, an implementation for the design may completely distort its conceptual clarity. For example, additional classes irrelevant to external behavior may be introduced during implementation.

Ideally, classes should be treated as functional components (or functional subsystems). They should be directly pluggable into large systems and replaceable by components with similar interfaces. Then, a software development process becomes one in which new systems are obtained by selecting, adapting and connecting functional components. The theme of this position paper is to discuss the possibilities of viewing software construction activities as identifying software components and wiring them together to perform prescribed functions.

2.0 Software components

(In this discussion, the users of a class refers to other classes that make use of the functions of the first class.) What are meaningful software components? Classes are an obvious candidate. Nevertheless, there are a number of problems when treating classes as software components. Firstly, high-level functionalities (from a user's view) of a single class is seldom well defined: It is often the case that several classes are needed to collectively implement a user functionality in a meaningful and manageable way (for example, as hierarchically related classes). Secondly, the interfaces of classes are not always defined in their best forms in terms of their conciseness, clarity and minimality. In fact, if multiple classes are employed in implementing a high-level functionality, very likely there is a need to provide variables and operations for communications among these classes. These internal variables and operations can be totally irrelevant to their users. Their presence in the interface definitions simply make the code more difficult to understand or use.

Based on these observations, it is reasonable to assume that an additional construct to correspond to the concept of software components can serve the purpose better. We propose such a construct for object-oriented programming languages here. A class-module (or CM) is a language construct for grouping object classes and for hiding their implementation detail or internal communication protocols. Similar to the concept of modules in languages supporting the concept of modules (such as Modula-2), a CM contains a module interface and a module implementation. A module interface would contain import and export specifications for module variables, module operations, classes, class variables and class operations. A module implementation would con-
sist of class definitions and class implementations, as are supported in today’s object-oriented programming languages. It may also contain rules for deriving the values of exported variables.

For example, consider the task of developing a compiler, which typically consists of seven subsystems (a lexical analyzer, a syntactical analyzer, a semantic analyzer, an intermediate code generator, an optimizer, a code generator and a symbol table manager). Realizing each of the subsystems with a class would not be considered a good design by any standard. That would make the interface definition for the classes extraordinarily large, and thereby difficult to understand or use. In this case, all instance variables and operations (including private or internal ones) would have to be declared in the class interfaces. Furthermore, the seven classes were likely to be organized into a flat structure, since it is difficult to design a sensible type hierarchy for these subsystems with distinct functionalities.

With the concept of class modules, on the other hand, each of the subsystems could be realized by a class module. This alternative is potentially advantageous in several ways. Firstly, the module interfaces would likely be cleaner and smaller as they only need to contain information that is relevant to communications among the subsystems (e.g., the interface of the module implementing the function of a scanner perhaps only contains one operation, for returning the next lexical token; whereas the syntax module interface only need to contain operations for obtaining and querying abstract syntax trees.) Secondly, within each class module, object classes can be organized into hierarchies to exploit inheritance and polymorphism. Thirdly, component reuse could be achieved at both class level (e.g., by import/export classes) and module-level (e.g., reuse the lexical analyzer and the syntactical analyzer in a different project). An additional benefit of these class modules is that they provide a natural lexical scope for names, which can be very useful in software development projects involving multiple packages from different vendors.

3.0 Requirements for specifying OO software with CMs

Class modules provide a means for constructing software at a high level of abstraction. They make it possible to describe system-level integration based on module interfaces. To describe software construction with class modules, appropriate programming languages are needed. These languages would include constructs for expressing system implementation with class modules. They should also be capable of describing implementation detail at subsystem-level (within modules).

Several requirements for such a specification language are summarized below. First, the language must be formal: its syntax and semantics must be precisely defined. Second, the language must be Turing-complete as it should be able to describe every detail of an implementation. Third, the language must provide constructs for selecting software components with compatible functionalities. Fourth, the language must be able to express adaptations on module interfaces for functionally compatible software components (identified manually or based on certain module attributes) but with variations in interfaces. Fifth, the language must be powerful enough to describe inter-connections among class modules, including directions as well as the type of information flow across the module interfaces.

4.0 Summary and concluding remarks

Classes are a powerful mechanism in object-oriented style of programming. However, in a large-scale software development effort, they provide little help for controlling the complexities of the class interfaces, so that classes implementing one subcomponent can easily be understood and integrated with classes for other components. Furthermore, it is not uncommon at all for software projects to use existing class libraries (either commercial or public domain) for construction of subsystems. It can be disastrous to use libraries that have class name clashes, especially when only binary libraries are available.

Class modules provide an extension to object-oriented programming languages. They can be used to naturally encapsulate external functionalities of subsystems. With such a construct, the interface complexity can be controlled at its minimum: Only information pertinent to inter-module communications need and should be present in a module’s interface definition.

More importantly, class modules could potentially make subsystems easier to understand and maintain. Class modules also encourage the use of existing packages for new software development, by virtually eliminating the difficult name clash problem among class libraries from different vendors or research labs. As more and more domain-specific class libraries are becoming available, we predict that software development in general will be heavily dependent upon these pre-defined packages; and more and more functionalities in an application will be the result of reusing or adapting existing packages rather than being developed from scratch.