ABSTRACT DATA TYPES AND OBJECTS FOR DEVELOPING COMPONENT BASED SOFTWARE

Abstract: *Abstract data type (ADT) and objects are the key factors for building reusable components for an object oriented programming language (OOPL). In this paper, we first propose a method to study the properties of an object and use them to build components of an OOPL. Finally we compare the proposed method with the conventional one.*

Keywords: *Abstract data types, component-based software development, object-oriented programming language, objects, components, reusability.*

Introduction

Component-based software development (CBSD) is an emerging paradigm of software development (Aoyama, 1998). This approach may be used to develop more reliable, portable and low cost software. In CBSD approach a lot of existing software components may be used with or without modification. In object-oriented programming (OOP) components are very crucial for developing software. Abstract data types are the key factors for building reusable components. A reusable software component is a collection of operations designed to aid programmers in the development of application programs. The use of software components saves both time and money. This savings along with assured accuracy and reliability is the main reason for using components.

The use of component in developing software is being widely used in recent times. A large number of software industries in various countries of the world have been motivated by the effectiveness of the reusable components. Component-based software development using existing software component has several advantages over traditional or conventional software development. Component reusability offers the software developers or software engineers a large degree of flexibility to develop software with more reliability and low cost. As more and more systems are built from existing component, it has become increasingly important to have proper characterization of components (Han, 1998).

A reusable component may be (McClure, 1992):

- *Program code* (whole program, code fragments)

DOI: https://doi.org/10.53808/KUS.2001.3.1.0143-se

- *Design specification* (logical data model, process structures, and application models)
- *Plans* (project management plans, test plans)
- *Expertise experience* (life-cycle model reusable, quality assurance, application area knowledge)
- Any information used to create software and software documentation

A reusable component has the following properties (Meyer, 1994):

- *Additivity:* Able to combine components with minimal side effects and without destructive interaction.
- *Expressiveness:* The formalism should express all possible kinds of components.
- Formal mathematical basis: Allow correctness conditions to be stated and component combination preserve key properties of components.
- *Easily describable:* Easy to understand and explain
- *Programming language independent:* Not unnecessarily specific about superficial language details.
- *Verifiable:* Easy to test.
- *Simple, simple, precise interface:* minimal number of parameters passed and parameter passed explicitly.
- *Defined protocols:* Protocols should exist for the use of components, so that a software developer can employ component licensed from many companies.
- *Easily changed:* Easy to modify with minimal and obvious side effects.
- *Reusable:* Has high reuse potential; likely to be usable in many systems.

Sources of components

Libraries of modules which manipulate simple data type and which have single well-defined functions e.g. subroutines, libraries for numerical analysis, where the complex operations are performed. A library of reusable components requites very little work to incorporate them into a new system. When a developer builds an application from existing components, these components come from three different sources:

- Tools vendor
- Software companies that sell object oriented applications
- Internal development

Components that are created as part of an application system development are unlikely to be immediately reusable (Sommerville, 1998). These components are general towards the requirements of the system in which they are originally included. To be reusable, they have to be generalized to satisfy a wider range of requirements. These types of generalization may be *name generalization, operation generalization and exception generalization.* After generalization, the quality of the generalized component should be checked.

Figure 1 shows a model of an abstract data type illustrating the different types of operation, which might be included in a generalized reusable component.

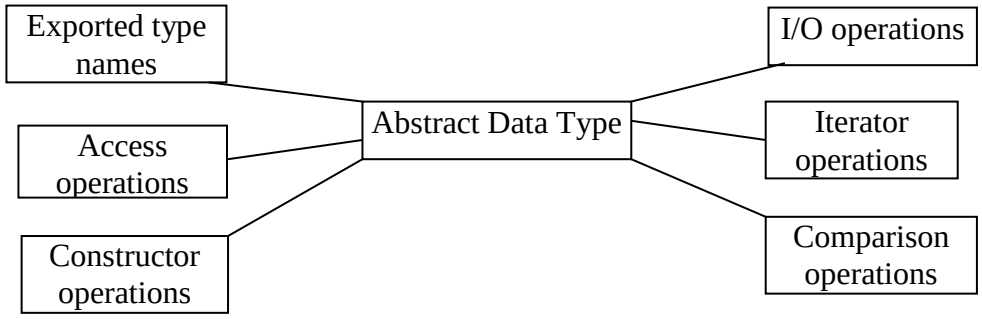

Figure 1. A model of reusable data type

- *Exported type names* are the names of types declared within the ADT which are available for reuse.
- *Access operations* are used to inspect the value of the elements of the component.
- *Constructor operations* are used to add or subtract elements to or from the component.
- *I/O operations* are used to read or write the component to and from disk and to print the component.
- *Iterator operations* allow each component of the ADT to be inspected without removing it from the data type.
- *Comparison operations* are used to compare one instance of the ADT with another.

In a component some operations may be implemented by the combination of other operations. If the component exhibits poor performance, it can not be reused. On the other hand very complex component can not reused because it is hard to understand. So it always a difficult decision for the reusable component designer to find an acceptable balance between providing a minimal and an efficient set of operations.

So far we discus the component and their general properties. To build a component we need to closer look at the construction level of a component.

Abstract data types (ADT)

Abstract data type (ADT) is the extension of the user-defined data type in conventional programming languages with encapsulation. ADT contains representation of data and operation on data. The encapsulation feature of the ADT not only hides the implementation of data but also provides a protective wall that shields its data from improper use. All interfaces occur through operations defined within the ADT. The operations then provide a well-defined means for accessing the objects of a data type. ADT gives objects a public interface through its permitted operations (Martin and Odell, 1992).

In ADT objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a customer, a table of data or any item that the program must handle. Programming problem is analyzed in terms of objects and nature of communication between them. Objects should be chosen so that they match closely to the real-world objects. When a program is executed, the objects interact by sending messages to each other. Each object contains data and code to manipulate the data. Objects can interact without having to know the details of each other's data or code.

ADT was first introduced in Simula 67. In Simula 67 ADT was named as class. A class is an implementation of an object type. It specified the data structure and permissible operational methods that apply to each of its objects. Different names are used in different programming languages. All ADT implementations provide the developer a way for developing systems for identifying real-world data types and combine them in a more convenient and concise form. Various data types and methods may be accommodated in ADT. Once defined, the developers can directly access ADTs later. That is why object technology has been widely adopted and has become the standard for much of the software industry in which objects are the core concept.

Objects take on different forms at different levels of abstraction. In OOP, objects are characterized by attributes and operations i.e.

$$object = attributes + operations$$

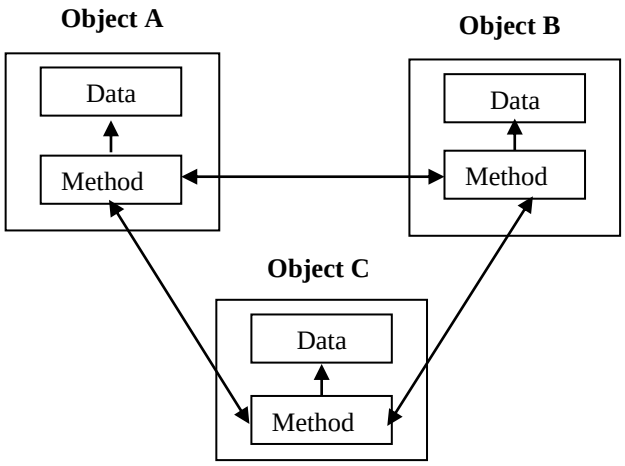

Figure 2. Organization of data and methods in OOP

A method in a class or object manipulates only the data of that class. Methods can not directly access the data structure of different class. To use the data structures of different class, they must send a request to that object.

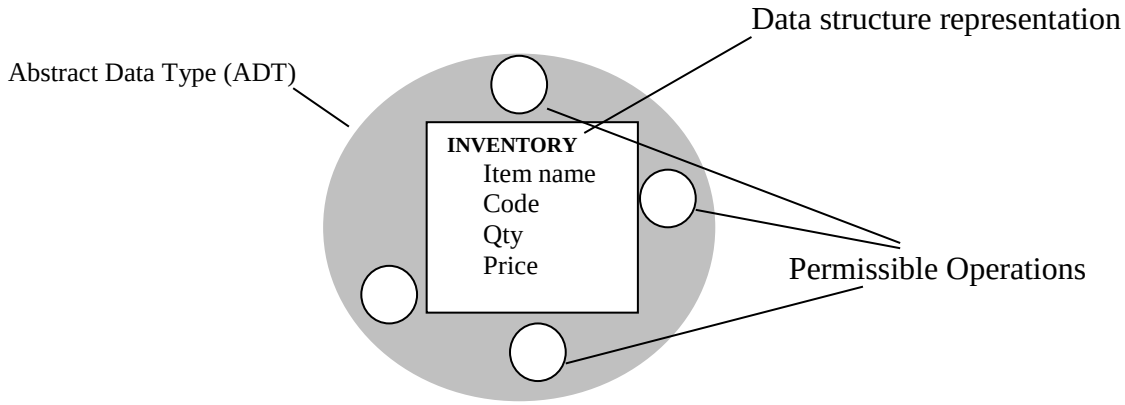

Figure 3. Data structure and Operation within an ADT.

The heart of an ADT is defined by its data structure and its representation. For example, in the above Figure 3.the ADT has **INVENTORY** data structure in which there are *Item name, Code, Qty* and *Price* fields. The operations defined within the ADT provide permissible ways for accessing the data within the ADT. Operations also protect the ADT from arbitrary and unintended use. Simply speaking operations provide the only way for accessing and maintaining the data within an ADT.

Operations within an ADT are processes that can be requested as units, which are called methods. Methods are procedural specifications of an application within a class. The methods in a class manipulate only the data structures of that class. They can not directly access the data structures of a different class. To use the data structures of a different class, they must send a request to that class.

```
class INVENTORY
{
char item_name[30];
int code;
int qty;
float price;
public:
      void getdata();
      void writedata();
      void purchase();
      void sell();
      void stock();
}
```

The above definition in C++ provides the way of binding data and operations within a class. This way encapsulating data and operations protect the object from arbitrary and unintended use.

Component Building Method

To build reusable component several concepts are applied on ADT. These are inheritance, polymorphism, dynamic binding, message passing etc. Successfully applying the concepts on ADT enables the ADT to be a reusable component. Now we discuss each concept separately.

Inheritance

It is an important concept, which is applied to object to make it reusable. Inheritance means the data structures and operations of a class physically available for reuse by its subclasses.

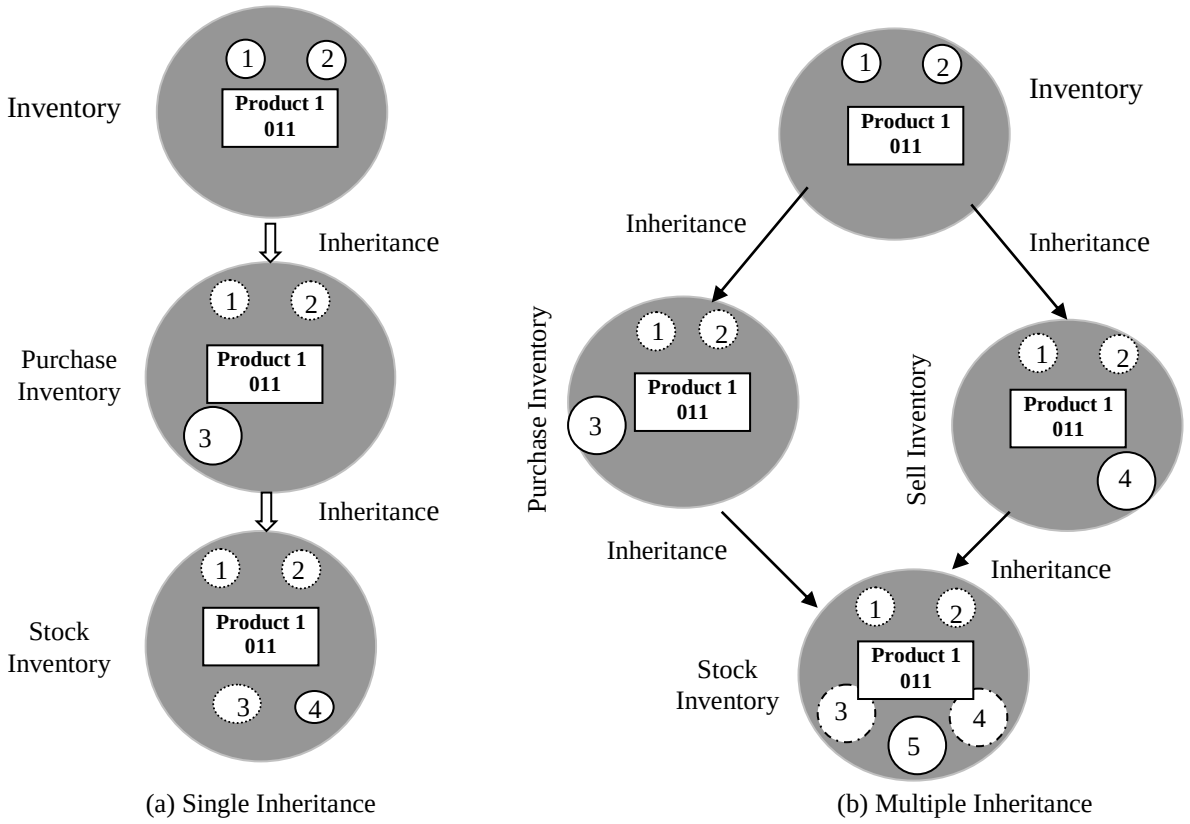

Figure 4. Inheritance of an ADT

Using this concept it is possible to add new features to an existing class without modifying it. This is done through deriving a new class from the existing one. The new class will have the combined features of both the classes. Inheriting the operations from a super class enables code sharing rather than data redefinition among the classes. Inheriting the data structure enables structure reuse.

Inheritance may be in different form. It may be single or multiple.

- *Single Inheritance* refers to inheriting data structure and operation of a class from one super class.

- *Multiple inheritance* means inheriting data structure and operation form more than one super classes.

In Figure 4 (a) depicts single inheritance and (b) depicts multiple inheritance. Base class *Inventory* has been used as super class in both cases. In single inheritance the *Purchase Inventory* inherits methods 1 and 2 from the class *Inventory*. *Purchase Inventory* also has its own operation 3. The class *Stock Inventory* inherits methods 1, 2 from *Inventory* and method 3 from *Purchase Inventory*. It has also its own method 4.

In case multiple inheritance *Purchase Inventory* and *Sell Inventory* classes inherit methods 1 and 2 from *Inventory* class and they have own method 3 and 4 respectively. The bottom class *Stock Inventory* inherits methods 1, 2, 3 from *Purchase Inventory* and methods 1, 2, 4 from *Sell Inventory*. *Stock Inventory* has also its own method 5.

In both cases of inheritance the bottom class has all methods available for reuse. In object oriented design this generalization hierarchy is implemented using inheritance.

Polymorphism

Reuse of code is one of the major goals in object oriented techniques. However all operations may not be used in the same form of the super class. There need customization of some operations. On the other hand an organization may have different methods for reusing an existing object. Although the methods for reusing are different they accomplish the same operational purpose.

To develop a computer system, even systems for widely different areas, there is a high degree of commonality between the abstract data types that are used (Ince, 1991). For example, table is a data structure, which is used, in different names in different programming languages. The items in the table may be different but the processing of items in the table is almost the same in every programming language. As a result the structure may be replicated from one application to another. Object oriented programming languages allow objects or classes to

describe general objects that can be oriented towards a particular application. The feature is known as *polymorphism* or *genericity*.

Polymorphism means the ability to take more than one form (Balagurusamy, 1999). For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation.

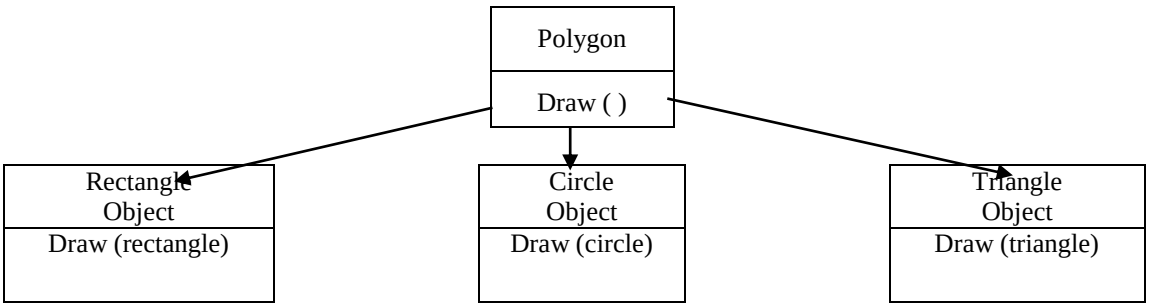

Figure 5. General form of polymorphism

Figure 5 illustrates that a single function name can be used to handle different number and different types of arguments. When the function Draw() is invoked with arguments, the required polygon is drawn according to the arguments passed. This is somewhat similar to a particular word having several different meanings depending on the context. Thus via polymorphism, object oriented programming languages enable a library of basic abstract data types to be written which can then be reused from application to application.

Inheritance and Polymorphism

Inheritance and polymorphism features of OOP provide reusability of objects in various ways. Different OOPLs have different inheritance mechanisms When a request for an operation goes to a sub class, all of its permissible operations are checked whether the request to be served or not. If the required operation is found on the list, it is invoked. Otherwise, the parent classes are examined to locate the operation

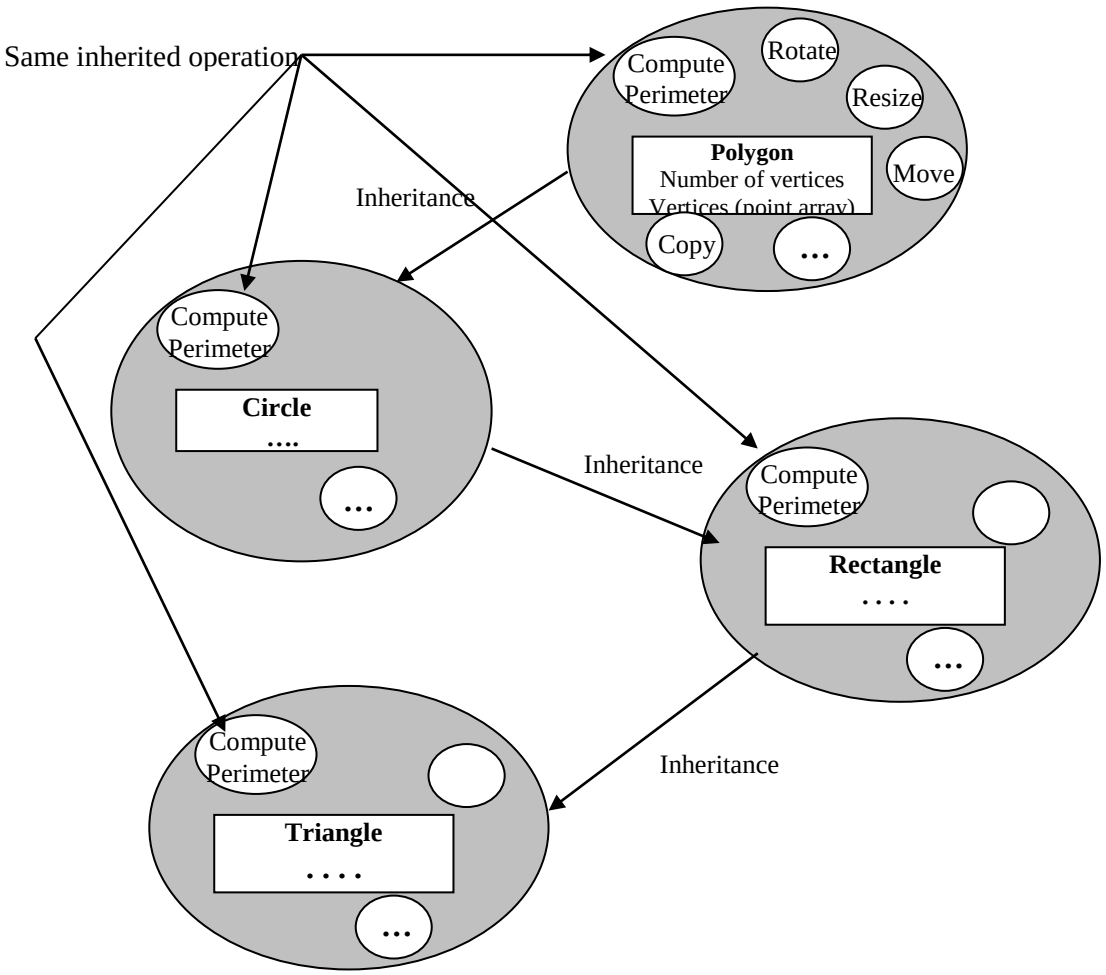

Fig.5 Inheritance and Polymorphism of objects

An important feature of inheritance is the ability of and ADT to override inherited features.

```
template <class P> class polygon
{
int vertices[n];
public:
    ploygon();          //Automatic constructor
    ~polygon()          // Automatic destructor
    P obj();
    int count()
    void compute_perimeter(P Obj1, iff_err, &Err);
    void rotate(P Obj1, iff_err, &Err );
    void  resize(P Obj1, iff_err, &Err);
    void  copy(P Obj1, iff_err, &Err);
    void move(P Obj1, iff_err, &Err);
    void print(iff_err, &Err)
}
template <class P> class iterator
{
    iterator();
    ~iterator();
    void create(polygon<P> Obj, iff_err, &Err);
}
```

Figure 6. C++ Implementation of reusable polygon abstract data type.

The processing of method or algorithm of an inherited operation can be redefined at the subtype level. The example in Figure 6 illustrates four ADTs. The most general, **Polygon**, contains the data structure and permissible operations for polygons. All operations defined within the **Polygon** class are same for all sub classes except *Compute Perimeter*. The method of computing perimeter differs from object to another. For this reason *Compute Perimeter* has been inherited in each object. Although the operation for computer perimeter is inherited from the **Polygon**, the method selected for **Circle** is located in the **Circle** ADT.

Dynamic Binding

It refers to the linking of a procedure call to the code to be executed in response to the call. The code associated with a given function call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function all associated with a polymorphic reference depends on the dynamic type of that interface.

In Figure 5 the function *Computer perimeter* is called in every object. The method of computing perimeter is unique to each object. So the function will be redefined in each class that defines the object. At run-time, during the code matching the object under current reference will be called.

Comparisons with the Conventional Approach

Although object-oriented technology have promoted software reuse, there is big gap between the whole systems and classes. To fill the gap, many interesting ideas have emerged in object-oriented software reuse for last several years. These include software architecture (Gamma, 1995), design patterns (Fayad and Schmidt, 1997) , and framework (Meyer, 1994).

CBSD approach takes different reuse techniques in the following manner:

- **Plug & play:** Component should be able to plug and play with other components and/or frameworks so that component can be composed at run-time without compilation.
- **Interface-centric:** Component should separate the interface from the implementation details so that they can be composed without knowing their implementation.
- **Architecture-centric:** Components are designed on a pre-defined architecture so that they can interoperate with other components and/or frameworks.
- **Standardization:** Component interface should be standardized so that they can be manufactured by multiple vendors and widely reused across the corporations.
- **Distribution through market:** Components can be acquired and improved through competition market and provide incentives to the vendors.

The nature of CBSD suggests that it should be different from the conventional development model in many respects. Table 1 summarizes major characteristics of conventional software development and CBSD.

Table 1. Comparisons of Conventional and CBSD Models

Characteristics	Conventional Model	CBSD Model
Architecture	Monolithic	Modular
Components	Implementation and White-Box	Interface and Black-Box
Process	Big-bang and Waterfall	Evolutional and Concurrent
Methodology	Build form Scratch	Composition
Organization	Monolithic	Specialized: Component vendor, Broker and Integrator

Conclusions

Software reusability is the best way to develop large size software with low cost and minimum effort, which concentrates on using existing components by adding new features to

them. To make reusable components we need to apply various concepts on objects such as inheritance, polymorphism, dynamic binding etc. In this paper, we proposed a method to study the properties of an object and used them to build a component of an object oriented programming language (OOPL). We also presented a comparison of our method with the conventional one.

References

Aoyama, M., 1998. *New age of Software Development: How Component-Based Software Engineering changes the way of software development*, International Workshop, 1998.

Balagurusamy, E., 1999. *Object-Oriented Programming with C++*. Tata McGraw-Hill Publishing Company Limited 1999.

Fayad, M. E. and Schmidt, D. C., 1997. *Object-Oriented Application Frameworks.* CACM, Vol. 40, No. 10, oct. 1997.

Gamma, E., 1995. *Design Patterns,* Addison-Wesley, 1995.

Han, J., 1998. *Characterization of Components.* International Workshop on Component Based Software Engineering, 1998.

Ince, D., 1991. *Object-Oriented Software Engineering with C++*. Tata McGraw-Hill International Series in Software Engineering, 1991.

Martin, J. and Odell, J., 1992. *Object-Oriented Analysis and Design.* Prentice Hall, Englewood, New Jersey, 1992.

McClure, C., 1992. *The Three Rs of Software automation: Reengineering, Repository, and Reusability.* Prentice-Hall Inc., New Jersey, 1992.

Meyer, B., 1994. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall International(UK) Limited, 1994.

Shaw, M. and Garlan, D., 1996. *Software Architecture.* Prentice Hall, 1996.