Component Software: A White Paper Part II. Technical Aspects

Gary A Bundell, Gareth Lee, John Morris[a], Stuart Hope[b], Shawn Parr and R Geoff Dromey[c]

Abstract

Component software is widely recognised as the key to further improvements in software productivity, reliability and flexibility. This paper reviews the state of the art in component technology and discusses the benefits of the component approach for developers, focussing on the technical benefits in increased productivity and reliability.

In defining a software component, we require 're-use' and 'substitutability' and argue that these two capabilities capture the key properties of productive components. This definition enables us to build a taxonomy of components - arranged in a hierarchy with increasing capabilities. The characteristics of each class in this taxonomy are discussed and a simple example - a lookup table - is used to illustrate how each new capability may be introduced into a simple component.

The infrastructures needed to support distributed components are described. The development, verification and maintenance of components are also discussed.

(Part I of this series focusses on management and commercial benefits.)

I. Background

Modern software systems are becoming so large and complex that it is beyond the capability of even large teams of programmers to design, specify, build and verify them from scratch in reasonable time frames and at reasonable cost. Component software construction is an approach to the economic construction of reliable large software systems; using trusted *components* - or smaller previously written and verified software modules - makes the construction of reliable and complex systems feasible on acceptable time scales.

A. Definition

The core attributes which software components exhibit are:

- Reuse - a component is written so that it may be used in more than one program *and*
- Substitutability - a component which performs an equivalent function may be substituted for an existing one.

The reuse capability implies that components should be designed from the start with general functions that allow the components to be used in programs other than those for which they were originally written. It also requires that components conform to some well characterized model for software modules: two models are well-understood and applicable:

- dataflow models *and*
- object models (the standard object-oriented paradigm).

Conformity with either of these models is essential for resuse, because a programmer intending to reuse a component must understand how it interacts with its environment. Similarly, in order to substitute one component for another, a programmer must also understand clearly how both components interact with the program's environment. This requirement is often stated in terms of the *architecture* of the component and the program in which it is embedded: clearly they must be compatible. The key difference between the two models is the presence or absence of persistent state: this difference is highlighted further in what follows.

A.1 Alternative definitions

Numerous alternative definitions appear in the literature[1], [2][1] : all authors include re-use either explicitly or implicitly. However our focus on substitutability as a key attribute is new (although Krutchen's definition includes 'replaceable'[4]), but it is implied in all other definitions by the requirement that an architecture[4], [2] or precise interface[4], [5], [2] must

[1] Sampat[3] has collected no less than 24 on his Web page!

[a] Centre for Intelligent Information Processing Systems, Department of Electrical and Electronic Engineering, The University of Western Australia, Nedlands WA 6907, Australia email: `[bundell,gareth,morris]@ee.uwa.edu.au`

[b] Software Engineering Australia (WA), Enterprise Unit 5, Technology Park, Bentley, WA 6102, Australia email: `s.hope@computer.org`

[c] Griffith University, Nathan, Qld, Australia email: `[S.Parr,g.dromey]@sqi.gu.edu.au`

be specified. Other characteristics, such as providing a reflection interface[5], binary format[5], dynamic loading[5] are also cited as key component properties. However we have preferred to use these characteristics to distinguish various types of components. Our definition also encompasses very simple components such as library functions, which meet the 'designed for reuse' and substitutability criteria. It allows us to define a natural hierarchy of components with increasing power and flexibility.
A key benefit that developers obtain from component based software engineering (CBSE) is reliability through reuse of verified components. By including simple components in our hierarchy, we are able to illustrate some fundamental principles which can lead to reliable, exchangeable components.

B. History

There appears to be no end to the supply of stories of software-induced disasters which have populated the Risks forum started by Peter Denning in ACM's Software Engineering Notes many years ago. These stories demonstrate that - at least in everyday practice - the art or science of software production has a long way to go. They also illustrate the sometimes enormous costs that software errors can inflict.
The history of component development is reflected in the increasing capabilities seen in figure 1. The earliest reusable software components were pure dataflow (or stateless) functions which could be placed in libraries and linked into any program. However, most programs model operations that take place in the real world[2] and dataflow functions, lacking any persistent state, are unable to effectively model these objects. Thus increasingly powerful software components have been developed which more effectively model real world objects.
Whilst it is clear that components alone will not cause Denning's supply of material to dry up: there is some optimism that a focus on clearly defined *independently verifiable* components will reduce the incidence and cost of errors. There is a close link between substitutability - one of our key characteristics of a component - and independent verifiability, which requires that a component can be 'dropped in' to a test harness as readily as it is connected to other objects with which it communicates in a real system.

B.1 Object Oriented Design - a precursor

Object oriented design strategies provided a formal basis for grouping the various functions that a software system must provide into modules. The idea that a *class* models objects found in real (or virtual) worlds provides a clear basis for the construction of software modules. By focussing on models of real objects, object oriented design also made it possible

- to substitute new models of parts of a system
- to reuse models in systems which were quite different from the one for which they were originally written

in a natural way. Component Based Software Engineering (CBSE) is a natural development of these ideas. As the taxonomy in figure 1 shows, the progression from class to component is one in which certain capabilities, such as substitutability and reflection interfaces, have been added. To be effective components, classes need to be designed with reuse and substitution in mind. Fully fledged components, having 'grown' from distributed objects, can also develop their interaction patterns with components that require their services through communication of capabilities at run time, either using infrastructures (*e.g.* ORBs *cf.* section II-E) or, more directly, as in JavaBeans(*cf.* section II-E.1). This considerably enhances the scope for reuse by enabling a component to be used in more environments. Substitutability is also enhanced, by enabling components that provide the same services in different ways to interact.
The notion that components should be able to dynamically assemble connections to other components has been borrowed from earlier distributed object technologies. These technologies were built primarily for this purpose, so a great deal of this infrastructure can be reused. This realisation has lead to the rapid enhancement of distributed object technologies to support components with enhanced encapsulation and substitution capabilities. An important issue here is how much of the distributed object technology needs to be encapsulated with the components to potentially provide them with a largely autonomous distributed integration capability, and how much is left to the distributed object repositories. The likely answer is a flexible hybrid arrangement since it is desirable that components and distributed objects are fully interoperable.

C. Economics of Software Production

The accompanying paper[6] discusses component technology from a business perspective. It lists a number of 'business drivers' which favour CBSE. This paper focusses on a selection of those drivers and shows how component technology addresses the business needs.
It has been known for several decades that the production of reliable software development is expensive: reuse of verified components leads not only to fast cycle times and higher productivity but also reliability through the use of trusted components.

[2].. or did take place until a program replaced them! For example, a bank's systems model the paper ledger books that they replaced.

Package software 'bloat' leads to large, difficult-to-learn packages that *may* solve your problem. The component approach gives the potential to rapidly construct - from the best tools available - a simple, efficient program that performs a desired task well rather than at some level of compromise. Furthermore, the ability to substitute new components for existing ones enables systems to rapidly adapt to changing work patterns, business processes, legislative requirements, *etc.*.

D. Organisation of this paper

Section II sets out a taxonomy for component software; we describe the characteristics of each class within this taxonomy. Throughout this section, a simple example - a lookup table - is shown at increasing levels of capability from a simple class to a component that can select an optimal implementation for a particular application dynamically. In section III, we discuss the construction of component software and enumerate the advantages the technology provides for software developers. Standards have proved a particularly intractable problem for rapidly developing software technologies: the impact of standards, or the lack thereof, on the growth of the component industry is discussed in section IV. Some of the ideas that form the basis for component software have already been extended: the future benefits that these extensions promise are discussed briefly in section V. We conclude in section VI with a summary of the factors which we believe make component technology a vital factor in current and future software development.

II. What are Components?

The term 'component' encompasses a surprising variety of software styles: in this section, we develop a taxonomy and describe each class within this taxonomy. Our taxonomy starts with the simplest possible components - functions (generally mathematical ones) found in general purpose libraries - and builds a hierarchy of components with varying capabilities leading to binary components with reflection interfaces that are independently deployable and dynamically replaceable. At each level in the hierarchy, we describe how components in that sub-group meet our two criteria and enumerate additional capabilities often possessed by members of the group.

A. Component Taxonomy

Figure 1 shows stages in the development of the capabilities of software components from pure dataflow functions to dynamically loadable, replaceable components with reflection interfaces. These stages are described in terms of the 'industrialisation' of software production in the preceding paper[6].
These categories of functions are described in the following sections:

1. Functions
2. Modules *or* Abstract Data Types
3. Classes
4. Dynamic Classes
5. Distributed Objects
6. Components

Plug-ins, Extensions, Middleware and Filters - essentially components that appear with new names appropriate to a specific use - are also briefly discussed.

B. Functions

Extensive libraries of mainly mathematical functions have been written over several decades, for example NAG[7] is a large, commercially available library of mathematical functions which can be used in many scientific and engineering applications.
Functions are the simplest components: to be reusable, a function must conform to the dataflow model; it must have no state or memory. Functions which conform to the dataflow model are excellent components. A dataflow module can be well represented by a *dataflow diagram*, *cf.* figure 2 in which data items are depicted entering a module which transforms the incoming data into a set of data items which are depicted leaving the module. In the dataflow model, the outputs are functions of the inputs *only*: there is no *persistent state*. Every time a program supplies a set of inputs with values identical to a previously supplied set of inputs, the module will produce outputs with the same values as the previous invocation of the module. There are no 'hidden' values preserved from invocation to invocation which affect the output of a second invocation with the inputs having the same values. Dataflow modules are also described as being 'side-effect free': they have no side-effects other than the production of the specified outputs.
Having no side-effects, components constructed to conform to the dataflow model are trivially substituted for components which have the same inputs and perform equivalent functions. The new component must produce at least those outputs required by the program in which it is to be embedded. (It may produce additional outputs, but these can simply be abandoned!)
Some of the simplest examples of dataflow functions are the trigonometric functions, *e.g.* the sine function in a C library:

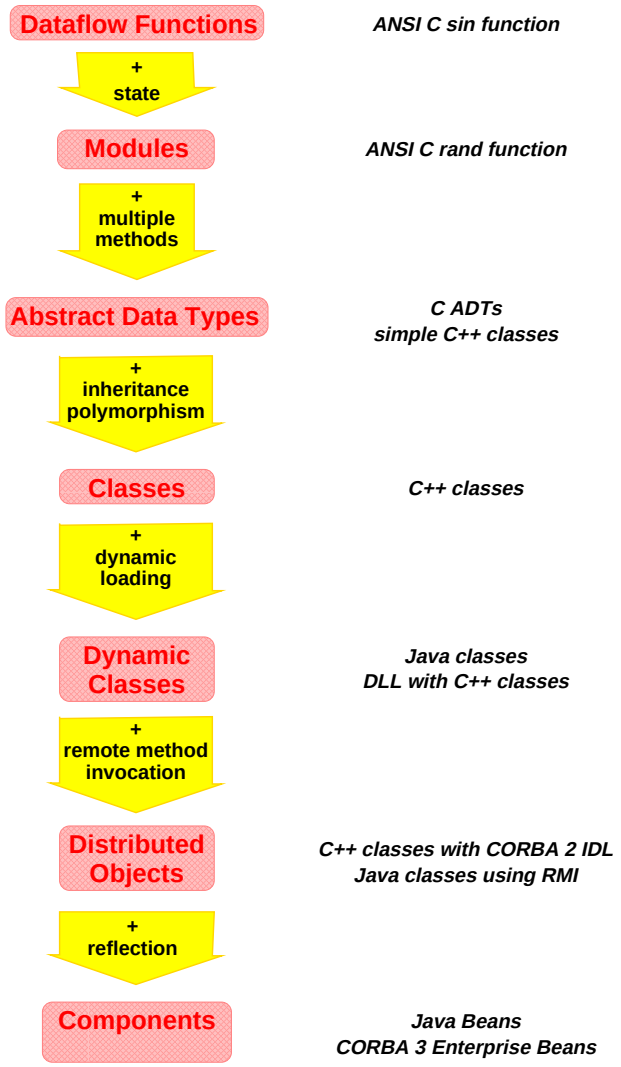

Fig. 1. Component Taxonomy

```
double sin( double x );
```

takes a single `double` input and produces a single `double` result. An existing `sin` function may be substituted with one which uses, for example, a different series expansion to produce either a more precise result or to produce a (presumably less accurate[3]) result in fewer CPU cycles. More complex functions such as those that invert matrices, calculate determinants, *etc.*, follow the same dataflow model: invoked twice with the same inputs, they will always produce identical outputs, without affecting any other part of a program. Thus they also can be replaced by functions which are faster (perhaps at the expense of accuracy) or more efficient (*e.g.* use less temporary working space) or more accurate.

C. Modules

In order to better model real world objects, programmers started to group data and functions together informally in modules. The presence of persistent data in a module meant that two calls to the same function may produce different results.

A very simple example of this is a random number generator, which keeps the previously generated number and generates the next one from it. A random number function may *appear* to be a pure dataflow one:

```
int rand( void );
```

[3]Note that here, as with any other floating point result on a machine with a limited precision, when we say that a function produces the *same* value as another, we interpret *same* in terms of some allowable error, so that we allow that a less accurate result is the same as one produced previously by increasing the allowable error.

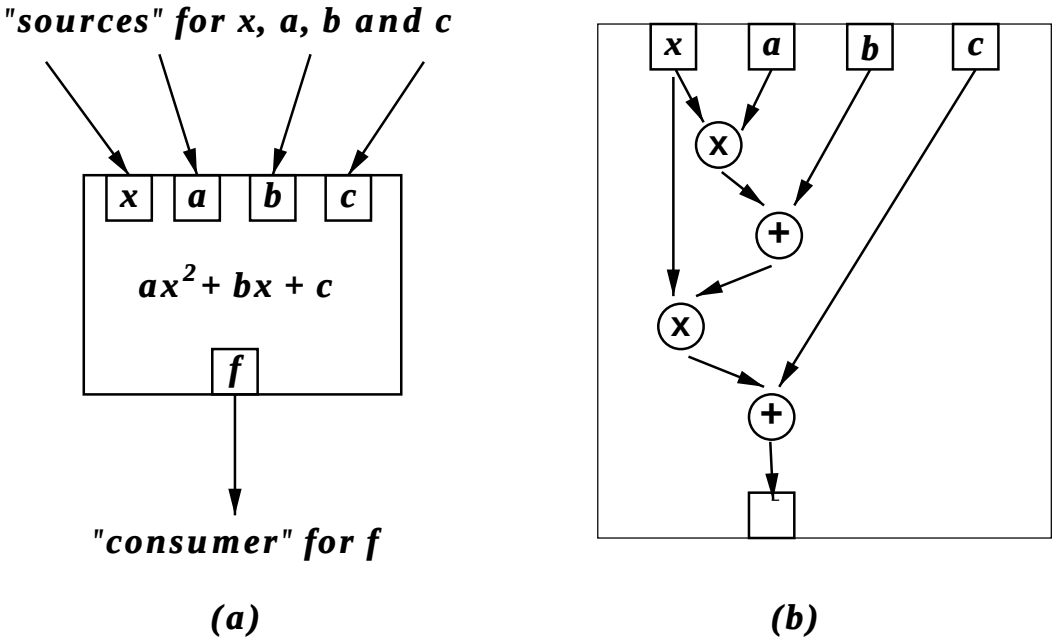

Fig. 2. Dataflow diagram. (a) shows a dataflow diagram with a 'black box' which calculates $f = ax^2 + bx + c$ from values supplied by sources for x, a, b and c and sends the result f to a 'consumer'. (b) shows the internal structure of the box in (a).

which simply produces a result, but there is a some hidden *state* - the value of the previously generated random number - which determines the next number to be produced.

Persistent State - an important qualification
Pure dataflow functions are the simplest examples of components which are *stateless*: given the same input, they always produce the same output. As will be demonstrated later, statelessness or the absence of persistent state, the 'hidden' values that persist from invocation to invocation and cause subsequent invocations to produce different outputs, not only makes the operation of a component much easier to describe but it also makes the task of verification of a component considerably easier. Thus, in the following sections, we will carefully characterise all components as being stateless or not.

D. Object Orientation

D.1 Abstract Data Types

The need for a formal basis for determining *which* functions should be grouped together in modules become apparent. The pre-cursors to current object-oriented classes were *abstract data types*: groups of functions which modelled the behaviour of a class of objects. An abstract data type consists of

- a set of attributes *and*
- a set of functions which define the *behaviour* of an object.

Together, these attributes and behaviour-defining functions form a software model of a *type* - a set of objects having a common behaviour.

The term *abstract* is used because the behaviour of the objects modelled by an ADT is defined entirely by the functions that can operate on *instances* of the ADT. An ADT thus *hides* implementation details from a user of the model.

ADTs can be implemented in many languages which provide data aggregates (structures, records, *etc.*) and functions, *e.g.* with a little programming discipline, ADTs are realisable in C[8] or Pascal.

ADTs are the pre-cursors for object oriented approaches, which simply added inheritance and polymorphism to the fundamental ADT ideas. Since the lookup table class introduced in the next section (*cf.* figure 3) does not use inheritance, it could be classified as an ADT as well as a class.

Each *instantation* of an ADT has values of the ADT's attributes associated with it: these values describe the *state* of the instance of the ADT. Thus an ADT is no longer a stateless function: every time a function belonging to an ADT is invoked, it operates on an instance with pre-existing values which may be changed by the function, changing the state of the whole program, which is, in turn, described by the state of 'active' objects within it.

```
public class LUT {
  ...
  public LUT( int min_entries ) { // Construct an LUT
    ...
    }

  public Comparable Find(Comparable key) { // Find an item in the LUT
    ...
    }

  public boolean Add(Comparable item) {    // Add an item to the LUT
    ...
    }

  public boolean Delete(Comparable key) {  // Delete an item from the LUT
    ...
    }

  public String toString() {               // Generate a printable
    ...                                    // represenation of the LUT
    }
}
```

Fig. 3. Look Up Table - Java version
This LUT will hold objects which implement a Comparable interface (defined in Java 2[9]): for simplicity, only the method specifications are shown.

Currently, the most substantial base on which to construct component technology is object technology since object technology clearly supports one of the primary attributes in the component definition, *i.e.* reuse. The second, and no less important attribute, of substitutability, is less directly supported: early object oriented systems required classes to be written in a single language and linked together at the same time. Gradually these restrictions have been relaxed and we can observe the 'growth' of object systems into component systems by the addition of capabilities as shown in figure 1. However, classes provide an ideal model for components: they have well-defined attributes and operations which may be performed on objects of the class. A well-designed class is a model for objects in a real or virtual world that a program manipulates.
To demonstrate the gradual growth in capabilities as an object system matures into a component one, we have chosen a single example - a lookup table - and sketched its implementation at different capability levels. The prototype lookup table modelled here is a simple one; it is a database of objects. We can add objects to the database and delete objects from it. We also need to be able to find objects in the database using a simple key. This application is a prototype for a growing organisation's database: it might start out as a file containing a few hundred items (*e.g.* customer contact numbers) read from a disc file and stored in memory which evolves as the organisation and number of records grows to a sophisticated relational database management system on a central server. The program which uses the lookup table has two major parts

- the lookup table (LUT) itself *and*
- a graphical user interface (GUI) which presents a friendly interface to users of the program, hiding the implementation details from users who don't want to know.

A good GUI generally takes several iterations of development and refinement and thus it's important that our organisation's investment in this part of the system is not lost as the underlying lookup table changes. This is but one example of the desirability of substitutability - a software system represents a significant capital cost and thus being able to retain parts which do not need change whilst substituting newer components to enhance the system's capabitilities is clearly a major benefit. This paper will trace the evolution of the LUT and assume that the GUI will remain unchanged. Of course, the GUI could evolve *independently of the LUT* for similar reasons.
Figure 3 shows the interface for a lookup table implemented in Java. It has a *constructor* (`LUT` in the Java code) which creates the LUT object by allocating memory, reading the data from a file and perhaps creating an index. It also has `Find, Add` and `Delete` methods[4].
In our growing organisation's first table, the implementation of this class might read a text file and construct a search table. While there are only a hundred or so records in the file, a simple array provides an implementation which is trivial to code, but provides adequate performance. However, as users add more records and performance suffers, the implementation can be enhanced to put the keys in a binary tree. Although the file may take a little while to load, once loaded, the response time is now excellent - and the users are satisfied. This is our first example of *substitutability*: as long as the interface in figure 3 is retained, no other code needs changes - and, apart from regained performance, the users (and programmers of the GUI) are blissfully unaware that anything has changed.

[4]Note that, in Java, we are able to provide a generic LUT by requiring that objects which can be stored in the LUT have a `Comparable` interface - a standard interface in Java 2, which provides a `compareTo` method.

D.3 Dynamic Classes

Conventional linkers load all the code for a program into one executable binary file which is stored and loaded into memory in its entirety when a program is invoked. The first attempts to allow a program to be 'constructed' when it was loaded used shared libraries. These contain functions which are commonly used by a large number of programs running on the same system: *e.g.* an information system's library of string processing functions or an engineering system's library of mathematical functions. Shared libraries permit faster loading of programs and reduce the load on system resources in multiple user systems as many users share one copy of commonly used code. They also permit *late binding*: the library functions are bound to the program - not at compilation time, but when the program is executed. This allows improved versions of the shared library code to be substituted after a program has been compiled and linked: only *stubs* for the shared libraries are needed at link time.

This idea has been extended in object oriented systems to allow whole classes to be loaded at program execution time. A Java program will load compiled classes from a variety of sources - local discs or remote systems - as a program executes. This not only reduces the load on a system's resources by bringing into memory only those classes which are actually used by the executing program. It is also possible for a running Java program to substitute new classes for old ones while the program is running as long as certain design disciplines (beyond the scope of this paper) are followed.

As our organisation grows, the contact file grows to contain several thousand entries and it becomes desirable to substitute a database management system (DBMS). With Java - or with dynamically linked libraries in C++ - the *implementation* of the class in figure 3 is re-written to access the database and re-compiled. The 'front-end' GUI remains the same *as long as the interface in Figure 3 is not changed*, it doesn't even need to be recompiled. When a user next runs the program, it searches for the new implementation of the LUT, loads it and starts to use the DBMS instead of the simple file that was adequate last week.

With Java, this dynamic linking capability was provided from the outset. C++ development systems now routinely provide dynamic linking to shared libraries by allowing programmers to link to the stubs of a shared library.

E. Distributed Objects

As more people need to access our organisation's database, it becomes necessary to locate it on a central server machine and enable access from desktop machines.

This goal may be achieved in a number of ways: each with greater flexibility and ability to adapt to changing needs *without changing the front-end* (which all the users have now been using effectively for several years) and *without alterations to any code except the implementation of the LUT.*

A simple approach uses two Java programs - the original user's program (which we'll now call the *client*) and another on the central machine (the *server*). The client uses *Remote Method Invocation* (RMI) to call methods of the server: Java's RMI infra-structure packages the method call arguments into a message packet and transmits it to the server. The server unpacks the information in the request, performs the requested action, packs the response into a packet which is returned to the waiting client.

This simple example illustrates one application of distributed objects: there are many more, for example, Lewandowski[10] provides a general discussion of the advantages of distributed objects in client/server systems.

Distributed Object Infrastructures

Remote method invocation works well when the client and the server 'speak the same language', but this forces a system builder to choose a single language for all parts of the system. Much more flexible solutions can be obtained by interposing an *object request broker* (ORB) between parts of the system. The ORB provides the infrastructure which links components together: all communication between objects in one component and objects in another passes through the broker. An ORB can handle requests for communication between objects written in different languages because all object interfaces must now be specified in a common Interface Definition Language (IDL). Object code is now linked with stubs, which translate requests from the object's language to the common format of the ORB. The ORB locates the target object and transmits the request to it. A receiving stub on the target object translates the message from the ORB to a form appropriate to the language of the target. Replies to requests follow a similar path in reverse.

Well-designed languages such as Ada and Eiffel now separate the interface to an object and its implementation - by requiring separately compilable specification and body (implementation) program units[5] - enabling a programmer to think in an abstract way about an object's capabilities. The ORB, of necessity, enforces this separation: access to an object can only be obtained through the interface, so objects communicate without knowing any details of their internal operations. Because all requests pass through the ORB, transfers between objects are handled in exactly the same way (as far as the objects are concerned) whether the objects are located on the same processor or on different planets of the solar system.

The IDL version of our lookup table example is shown in Figure 6. A minor complication will be noted in Figure 6, in the original Java implementation, we made use of its `Comparable` interface to ensure that the LUT had a method for

[5]Curiously, Java, despite being designed in the 1990s, has failed to enforce this separation!

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.ServerNotActiveException;
import java.util.Random;

// A class which directs requests through the LookupService
// interface to a local LUT object.

public class LookupServer extends UnicastRemoteObject
   implements LookupService {
   private LUT lut = null; // The lookup table which does the work!

   private int accesses;   // A count of external accesses.

   // Creates a server object.
   public LookupServer(LUT lut)
      throws RemoteException {
      try { // Register the server with a (pre-existing) RMI Registry
            // name server running on the localhost.
         Naming.rebind("///LookupService", this);
      } catch (Exception e) {
         System.out.println("Failed to start server:");
         System.exit(0);
      }

      if (lut == null) throw new IllegalArgumentException("lut must not be null");
      this.lut = lut;

      System.out.println("Lookup server running...");
      accesses = 0;
   }

   // This is a server wrapper which allows access to the Find
   // method of an underlying LUT implementation.
   public synchronized Comparable Find(Comparable key) {
      if (lut != null) return lut.Find(key);
      else throw new IllegalStateException("No lookup table available");
   }

   // This is a server wrapper which allows access to the Add
   // method of an underlying LUT implementation.
   public synchronized boolean Add(Comparable item) {
      if (lut != null) return lut.Add(item);
      else throw new IllegalStateException("No lookup table available");
   }

   // This is a server wrapper which allows access to the Delete
   // method of an underlying LUT implementation.
   public synchronized boolean Delete(Comparable key) {
      if (lut != null) return lut.Delete(key);
      else throw new IllegalStateException("No lookup table available");
   }

   // Start a single server running.
   public static void main(String[] args) {
      try {
         LUT lut = new LUT();
         LookupServer server = new LookupServer(lut);
      } catch (RemoteException re) {
         re.printStackTrace();
      }
   }
}
```

Fig. 4. Look Up Table - Java RMI version

The LUT Server using Java RMI. The server runs on a remote machine. The client adds the code in figure 5 to its implementation to make the connection to this server.

determining an order or sequence of objects. Using CORBA, we had to provide an additional interface in IDL to mimic the Java one. An IDL compiler compiles the specification in figure 6 to a form which an ORB uses to generate stubs through which requests from the LUT client are received by the ORB and despatched to the LUT server.

Thus we have two main approaches to distributed object management:

- Java-only technology (*e.g.* JavaBeans): all processors in the distributed system must have a Java Virtual Machine (JVM) which can interpret Java code. Component interfaces are defined in Java. Interoperation with components written in other languages is possible through bridges to other distributed object infrastructures.
- Language independent ORBs: In contrast, these infrastructures (*e.g.* CORBA and DCOM) explicitly support components written in several languages through the provision of the IDL. Interacting components are compiled from their original language to the binary format of the machine on which they are to execute. Furthermore, the same binary may be used for a server which responds to requests (transmitted through an ORB) from clients written in different languages and running on the same or remote machines.

Not surprisingly the emerging model for a "worldwide distributed object infrastructure" is largely a hybrid of these approaches which has been supported mostly through third-party bridging products - particularly between CORBA and DCOM. A recent update of the standards behind CORBA and EJB has seen the adoption of CORBA's Internet Inter-ORB Proctocol (IIOP) in EJB (specifically in RMI) and the EJB server side component model in CORBA[11]. It is expected that this trend will continue over the next few years, converging towards an industry standard that will form a more solid base on which to build a highly portable component marketplace.

Language independence is a major issue in environments where legacy applications must be supported; it allows trusted legacy code to be retained and communicate with new applications written using more powerful development tools.

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.Vector;

/** The bridge to a remote lookup table.  The bridge connects to an
   implementation on the specified server and then passes on the
   method calls through RMI to the server.
*/
public class RemoteLUT {
   /**
      The host on which the remote service will be sought.
   */
   private final static String SERVER_NAME = "some.host";

   /**
      The remote RMI server which is hosting the lookup table.
   */
   private LookupService server = null;

   /**
      Construct an instance of the bean.
   */
   public RemoteLUT() {
      try {
         // Lookup the remote server in using the key LookupService
         // using the RMI Registry on the localhost.
         server = (LookupService)
            Naming.lookup("//" + SERVER_NAME + "/LookupService");

      } catch (Exception e) {
         System.err.println("Failed to invoke remote service.");
         System.exit(0);
      }
      System.err.println("RemoteLUT: located remote server.");
   }

   /**
      Finds an item from the table with an equivalent key and
      returns a reference to the item.  It no equivalent item is
      found null will be returned.
   */
   public Comparable Find(Comparable key) throws RemoteException {
      return server.Find(key);
   }

   /**
      Adds a unique item to the table.  If an existing item within the table
      has an identical key the new item will not be added and the method
      will return false, otherwise the method will return true.
   */
   public boolean Add(Comparable item) throws RemoteException {
      return server.Add(item);
   }

   /**
      Deletes the item with an equivalent key from the table.  If no
      match is found no deletion occurs and the method return false,
      otherwise it returns true.
   */
   public boolean Delete(Comparable key) throws RemoteException {
      return server.Delete(key);
   }

   public String toString() {
      return "RemoteLUT[server=" + SERVER_NAME + "]";
   }

}
```

Fig. 5. Look Up Table - Java RMI version

A client uses this code to find the server and establish a connection so that it can remotely call the server's methods.

E.1 Distributed Component Technologies

In line with the three main infrastructures to support distributed objects, there are, not surprisingly, three evolving component technologies and some are more mature than others.

The basic Java based component technology is defined in the JavaBeans component model. This model can be used in any visual Java technology integrated development environment (IDE) to build Java classes. A JavaBeans component is just a specialized Java class that can be added to an application development project and then manipulated by the Java IDE. A bean provides special hooks that allow a visual Java development tool to examine and customize the contents and behavior of the bean without requiring access to the source code. Multiple beans can be combined and interrelated to build Java applets or applications or even new JavaBeans components. The Enterprise JavaBeans (EJB) component model just builds on JavaBeans component model by specifically supporting *server components*. Server components are reusable, prepackaged collections of tools which provide capabilities needed in an application server. They can be combined with other components to create customized application systems. EJBs are assembled and customized at deployment time using tools provided by an EJB-compliant Java application server.

CORBA 2 (released in 199x) did not explicitly support server side components, but it provided the infrastructure to allow CORBAservices to be provided on a server and accessed by clients. The assembly of server side services from multiple server components was not envisaged. CORBA version 3 has fully redressed that omission. The three major parts of the new CORBA component model (CORBAcomponents) are:

```
// lutidl.idl
// IDL defintion of LUT
//
interface Comparable {
//  Because Comparable is not a pre-defined IDL type, an
//  interface is required which will generate a Java Holder
};

interface LUT {
  void Create(in long min_entries); // Create LUT
  Comparable Find(in Comparable key); // Find an item in the LUT
  boolean Add(in Comparable item); // Add an item to the LUT
  boolean Delete(in Comparable item); // Delete an item from the LUT
  void StringtoString(); // Generate a printable representation of the LUT
};
```

Fig. 6. LUT IDL code: note that this is essentially the same as the Java code in figure ..

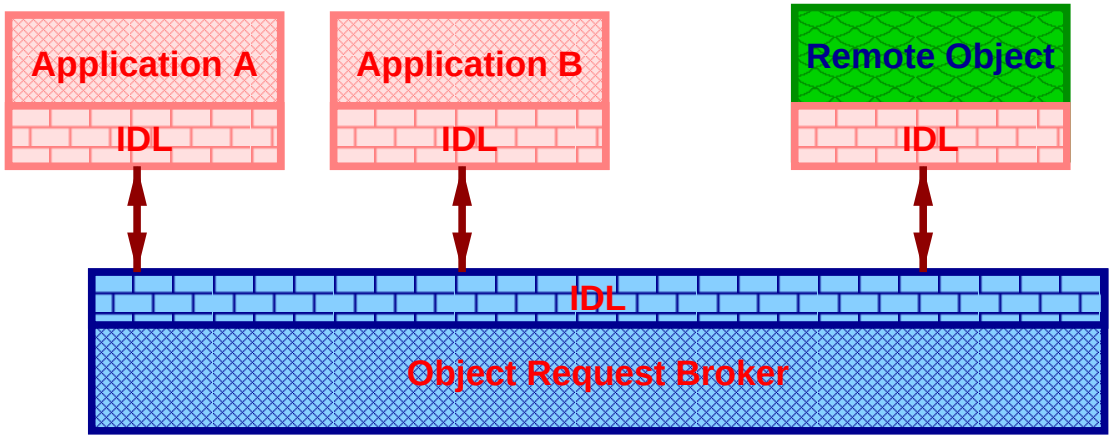

Fig. 7. Object Request Broker

- a container environment that packages transactionality, security and persistence and provides interface and event resolution,
- integration with Enterprise JavaBeans *and*
- a software distribution format that enables a CORBAcomponent software marketplace.

The CORBAcomponents container environment is persistent, transactional and secure. Component containers keep track of event types emitted and consumed by components, and provide event channels to carry events. The containers also keep track of interfaces provided and required by the components they contain, and connect one to another where possible. CORBAcomponents support multiple interfaces, and the architecture supports navigation among them. Enterprise JavaBeans (EJBs) will act as CORBAcomponents, and can be installed in a CORBAcomponents container. A major difference with EJBs, in-line with the original CORBA philosophy is that CORBAcomponents can be written in multiple languages and support multiple interfaces.

Microsoft's Distributed COM (DCOM) seamlessly extends its Component Object Model (COM) to support communication among objects on different processors. COM intercepts calls from an object and forwards them to any other object, while DCOM requires the use of an underlying network protocol for objects to interact with each other. With DCOM the details of deployment are not specified in the source code and the location of a component is undefined until execution. The way the client connects to a component and calls the component's methods is identical so that changes to the source code or recompilation is not required. DCOM also strongly supports multiple component implementation languages, since this also readily provides for rapid prototype implementation of components in one language (e.g. Visual Basic) prior to re-implementation for performance or portability reasons (e.g. C++ or Java).

A detailed introduction to distributed objects is provided by Krieger and Adler[12]. Szyperski's book[2] provides detailed comparisons of the competing technologies.

F. Components

Although we have used the term "components" extensively in the previous section, there is one further capability that adds additional flexibility and power to systems built from components. The final stage in the development of components as they now exist is the addition of *introspection* - the ability to query a component to determine its capabilities. This enables flexibility to be built into complex systems:

- components can locate other components with some needed capability from repositories and either communicate

with available instances of those components or load the code and create their own instances of the components *and*

- components which provide services can provide a number of published interfaces to meet the specific needs of individual clients.

This capability is provided through a *reflection interface* - one which allows a potential user of a component to determine its capabilities by asking the component for a list of methods which it provides. In our lookup table example, we assume that any object that provides `Find, Add` and `Delete` methods may be used in our application. Before our readers protest that this is a potential mine-field for errors, we note that a reflection interface must be able to provide the full *signature* (name, parameter list *and* return type) of a method before it matches a candidate with an application's requirements. Java's Reflection API enables us to obtain the full signature through a series of methods calls.

```
import java.beans.*;
import java.lang.reflect.*;
import java.util.*;

/** The proxy for a JavaBean with "Find", "Add" and "Delete" methods.  */
public class LUT {

  private Object instance = null; /** The bean object */

  /** References to the find, add and delete methods of the bean */
  private Method findMethod = null, addMethod = null, deleteMethod = null;

  /** Create a beanLUT and test whether it supplies the required interface.
      If appropriate, it will be used, otherwise an exception will be thrown.
  */
  public LUT(int min_entries) throws Exception {
    Object bean = new beanLUT( min_entries );
    testInterface(bean);
  }

  /** Check whether an object provides an interface comprised of "Find(Comparable)",
      "Add(Comparable)" and "Delete(Comparable)" methods.  If the object does not offer the
      interface NoSuchMethodException will be thrown.
  */
  public void testInterface(Object instance) throws IntrospectionException, NoSuchMethodException {
    BeanInfo classInfo = Introspector.getBeanInfo(instance.getClass());
    MethodDescriptor[] methods = classInfo.getMethodDescriptors();
    for (int i = 0; i < methods.length; i++) {
      Method method = methods[i].getMethod();
      Class[] parameters = method.getParameterTypes();

      // Check the number and types of parameter(s)
      if (parameters.length == 1 && parameters[0] == java.lang.Comparable.class) {
        // Check the method name
        if (method.getName().equals("Find")) findMethod = method;
        else if (method.getName().equals("Add")) addMethod = method;
        else if (method.getName().equals("Delete")) deleteMethod = method;
      }
    }

    // Check that all the required methods have been found
    if (findMethod == null || addMethod == null || deleteMethod == null)
        throw new NoSuchMethodException("Failed to locate essential method");
    // Found an acceptable class, save a reference to it
    this.instance = instance;
  }
   /** The following three methods act as proxies by calling the instance's Find method with the
     argument provided and returning the resulting value -
     cast to the correct type if necessary
   */
   public Comparable Find(Comparable key) throws Exception {
     Comparable c = (Comparable) callReflectedMethod(findMethod, key);
     return c;
   }

   public boolean Add(Comparable item) throws Exception {
     Boolean b = (Boolean) callReflectedMethod(addMethod, item);
     return b.booleanValue();
   }

   public boolean Delete(Comparable key) throws Exception {
     Boolean b = (Boolean) callReflectedMethod(deleteMethod, key);
     return b.booleanValue();
   }

   /** Calls a method on the remote object and returns its return value.
   */
   private Object callReflectedMethod(Method method, Comparable arg) throws Exception {
     try {
       Object[] arguments = { (Object) arg };
       Object returnValue = method.invoke(instance, arguments);
       return returnValue;
     }
     catch (InvocationTargetException ite) {
       throw (Exception) ite.getTargetException();
     }
   }

   public String toString() { /** Pass toString invocations directly to the bean */
      return instance.toString();
   }

}
```

Fig. 8. Using introspection to call the LUT class in figure 3

.

An example that shows the use of the reflection capabilities of a Java Bean is shown in figure 8. The LUT is changed so that it checks the bean, `beanLUT`, to find out whether it provides the required methods. (This simple example makes use of the fact that each method has a single `Comparable` object as its argument, thus it checks the argument count and type first, then checks the argument name. It is also possible to check the return type, but, for brevity, this has been omitted from this example.) If the required methods are found, references to them are captured in this object. Calling the three proxy methods 'bounces' the call through this 'wrapper object' to the target lookup table.

F.1 Finding the optimal component

The example in the previous section shows how a component could verify that it was using a component with some required capability (reflected by the provision of appropriate operations). A reflection interface also allows a potential user to select a component with the most appropriate characteristics for a particular application from a repository. For example, lookup table users might require some combination of these characteristics:

- Best average response time,
- Persistent, fault-tolerant storage,
- Best weighted average response time (fast response to frequent queries, slower response to rarer ones),
- Low memory overhead, *or*
- any other application specific attributes.

```
import java.beans.*;
import java.lang.reflect.*;
import java.util.*;

public class LUT {
   /**
      This is the list of known providers of the LUT implementations.
      Associated with each is a list of attributes which may be used
      in selecting the appropriate implementation for a particular user.
   */
   private static String[][] providers = {
      /* Implementation Class,                Attributes...         */
      { "ciips.testbed.introspect.SimpleLUT", "Default", "SmallData" },
      { "ciips.testbed.introspect.BinaryLUT", "Fast",    "LargeData" },
      { "ciips.testbed.introspect.RemoteLUT", "Remote" }
   };

   private Object instance = null;

   private Method findMethod = null, addMethod = null, deleteMethod = null;

   /**
      Searches through the list of implemenation providers to find a
      provider which supports the specified attribute flag.  If a match
      is found the appropriate implementation will be tested for its
      interface and used if appropriate.  Otherwise an exception will
      be thrown.
   */
   public LUT(String attrib) throws Exception {
      for (int i = 0;  i < providers.length; i++) {
         String implementor = providers[i][0];

         // Search through the providers list in order until a match
         // for attrib is found.
         for (int j = 1; j < providers[i].length; j++) {
            if (providers[i][j].equals(attrib)) {
               Class clazz = Class.forName(implementor);
               System.out.println("LUT: chose " + clazz + " from " + attrib);
               testInterface(clazz.newInstance());
               return;
            }
         }
      }
      throw new IllegalArgumentException(
         "No implementation available for " + attrib);
   }

}
```

Fig. 9. An LUT class which selects an LUT from a number of candidates on a remote source. A single string is used by the user to identify a suitable candidate. Only the constructor is shown, the remainder of the code is the same as that in figure 8.

In figure 9, we see an example design in which the LUT client queries several LUT implementations to determine whether one meets a desired criterion - specified by passing an appropriate descriptor string `attrib` to the constructor in the example. This example shows one way to query a class for information about its performance - a table is held in the "selector" class which has sufficient information about the candidate classes for the client to make a choice from. In figure 9, the table has been hard-coded into the selector class, but in a more realistic application it could be loaded from a file (either local or via some URL) - allowing installers of additional candidate components to simply update this file. There are several other techniques that could be used:

1. add an identifying method to the candidate classes, *e.g.* a hash table variant of our LUT needs a hash function, so it should have a method to set the hash function, *e.g.* `void setHashFunction( Method m );`
2. add an identifying field to the class - a hash table might have a `table_size` attribute. A `Field[] getFields()` method allows the fields of a class to be discovered in exactly the same way that the `Method[] getMethods()` method provides a list of methods in figure 8.
3. a `BeanInfo` object can be associated with each candidate class: this object provides a number of descriptors which can be searched for the required capability. This technique is probably the best one if large numbers of properties are involved and the client object is prepared to sift through all the properties and apply some heuristics to determine which class is the most suitable for the current application.
4. add a method which can be invoked once the class is loaded which provides performance information - this method

requires each candidate class to be loaded first and would presumably be used only when a `BeanInfo` object cannot supply suitable data easily.

It can be seen that reflection capabilities provide us with *adaptable* software systems. Our LUT client could determine some parameters of the LUT, *e.g.* maximum size - or whether a maximum size can be confidently predicted - before determining which LUT to use. Software systems can exploit reflection capabilities to build themselves as they determine the actual requirements of the current application. Since a class can be removed and a new one loaded to take its place, systems using reflection can even change their underlying implementations as they run to meet the changing needs of a persistent system.

In CORBA-based systems, the *Naming Service* provides the basic capability - allowing one object to locate another that will provide the required service. An additional service, the *Object Trader*, allows objects to provide and query a more comprehensive representation of their attributes and operations and facilitates the trading of objects as transactions between a vendor and a client.

```
public static void main(String[] args) {
  LUT lookup;
  static final int min_entries = 100;

  try {
    // Constructor for original LUT
    // lookup = new LUT( min_entries );
    // Constructor for new LUT which allows an attribute string to
    // specify desired characteristics of the LUT
    lookup = new LUT( min_entries, "SmallData" );
    lookup.Add(new Integer(13));
    lookup.Add(new Integer(3));
    lookup.Add(new Integer(9));
    lookup.Add(new Integer(7));
    Integer r = (Integer) lookup.Find(new Integer(3));
    System.out.println(r);
    lookup.Delete(new Integer(7));
    }
  catch (Exception e) {
    System.err.println("Test failed...");
    }

  System.out.println(lookup);
  }
```

Fig. 10. A simple example of a program which uses the Introspector class to set up proxies to call a target class - after having ensured that the chosen target does have the required interface. Note the trivial change to the constructor (original commented out) from the version of this program which interfaces to all the previous examples, illustrating the power of component substitutability to enhance capability without losing investment in software development.

Finally we observe that component software serving particular purposes often appears under special names reflecting that purpose. Some of these groups are described briefly in the following sections.

G. Plug-ins and Extensions

Plug-ins are components which extend the capability of a 'host' software system. They are most commonly encountered with Web browsers, which use plug-in components to interpret different types of content which may be part of a Web page. The ability to down-load plug-ins and link them into a host browser as Web pages requiring them are encountered has the effect of turning Web browsers into client-side operating systems whose capabilities may be extended as long as the host machine can provide resources such as memory and disc space[2]. They are thus dynamically loadable components which provide some reflection capabilities to allow the host to ensure that they have the necessary interface. The Macintosh operating system makes extensive use of *extensions* - mainly as device drivers. Again they are dynamically loaded as the operating system configures itself, but substitutions cannot be made without restarting the system. Many other operating systems also allow device drivers to be dynamically loaded - either as the system starts up and is configuring itself *or* at later stages as the need for a particular device becomes apparent. Since they must provide a pre-determined interface, they are also components providing reflection capabilities in some form, albeit it simple (*e.g.* an operating system might require a device driver to supply a table with pointers to required methods. An empty or NULL slot in this table would indicate the absence of a capability (*e.g.* a tape drive which cannot be rewound, so a `seek_back` method has no meaning). Too many empty slots would disqualify a driver and prevent it from loading.

H. Middleware

Middleware components are used to 'glue' systems together: they transfer service requests and responses from one system to another. Object Request Brokers (ORBs) are the most important group of middleware components: when one component invokes a method on another, they locate the target object, transform the request if necessary (*e.g.* if gluing CORBA and DCOM systems together), transmit it to the target object and ensure that the response, if any, is delivered to the requesting object.

```
ls | grep "abc" | wc -l
```

Fig. 11. A Unix 'program' that counts the number of files in a directory whose names contain "abc". `ls`, `grep` and `wc` are some of the earliest reusable, substitutable components.

Basically, they are dynamically linked components with a name indicating their purpose - to sit in the 'middle' and link other components together. Substitution is generally possibly in a running system.

I. Filters

Programs themselves may be classed as components. For example, Unix programs designed to be assembled together into pipelines are possibly the first practical examples of software components. *Filters* is an appropriate label for these pipeline components, because they operate on data flowing through them. Large numbers of Unix scripts have been written which assemble members of a relatively small set of programs, *e.g.* `grep`, `find`, `wc`, *etc.*, into larger and more complex programs.
The individual filter programs are not bound to the scripts until they are invoked by running the script, so that improved versions of the filter components may be substituted at any time.

III. Engineering Components

Components may be built following well-accepted SE models, *e.g.* a 'standard' development lifecycle may be employed:

- Determine Requirements
- Build a software specification
- Validate the specification against the requirements
- Implement
- Verify
- Maintain

However, the basic requirements for components introduce different considerations into each phase: in the following paragraphs, we outline some of the key differences. In the following section, tools supporting CBSE are briefly surveyed.

A. Requirements for Components

Successful components must be designed as components at the outset, so that a re-use requirement needs to be clearly enunciated along with other requirements. Except in the rare occasions when a system's life expectancy is known to be short, a system may need to be adapted to a variety of hardware and software architectures during its lifetime. Forward-looking users will build adaptability requirements into their system requirements. Professional developers or systems analysts will ensure that suitable adaptability is specified when users are sufficiently naive about technological change to ignore (or avoid) this issue. Components that interact with the environment can be specified to ensure that core operations of a program are 'insulated' from environmental considerations. These components are replaced as the environment - either hardware or software - changes allowing the core operation modules (which will generally contain the major unique or innovative parts of any system, and thus represent the largest investment of the organisation using the system) to continue to be used without alteration.

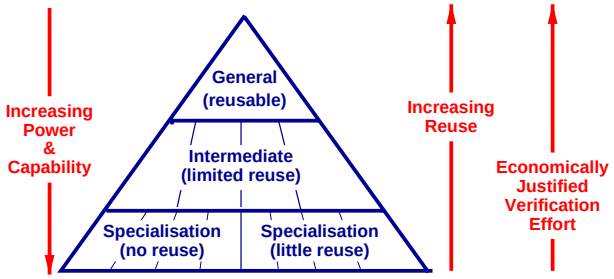

Fig. 12. Capability of components *vs* reuse

Design for reuse poses some special challenges: a component which achieves a high degree of reuse will certainly be more economic for its owner. Reuse justifies more effort in validation and verification and enhances the reliability of systems. However, a component which is very specialised is unlikely to be reused. On the other hand, 'lean' components with lesser capabilities are likely to be usable in more systems. Fortunately, languages which support *class inheritance*

provide a simple solution: a hierarchy of classes can be designed. At the top of this hierarchy, one specifies general classes with limited capabilities, but high reuse potential. Below these general classes lie specialised classes with additional capabilities needed by only small numbers of systems and thus limited reuse potential. The 'core' classes at the top of the hierarchy are naturally the focus of intense verification efforts (justified by their wide use): their defect levels will be very low after a short period.

B. Software specification

To ensure that a component can be re-used, it is essential that a formal modelling approach be taken. For component models, the pre-eminent modelling environment is the Unified Modelling Language (UML). UML provides a framework in which component specifications, use cases, activity diagrams, scenarios, timing diagram, *etc.*, can be constructed and linked to component deployment diagrams, to clearly show the internal architecture of components and their interfaces. Some CASE products, such as Rational Rose[13], can be directly linked to Integrated Development Environments (IDEs), so that a model expressed in UML can be converted to code and compiled. For example, Rational Rose is a CASE tool which supports the mapping of UML to component code in a variety of languages (e.g. Java, C++, Visual Basic) and the reverse engineering of component code back to UML. In this way the IDE can be used to debug the code and changes made will also be reflected in the design model for the component.
To verify individual components it is essential that the test case set encompasses all aspects of component execution. This can be guided by the development of appropriate models in the specification and design process for the component. At the highest level domain and product-family type models can be used to group like capabilities for components together. Thus design knowledge, or the patterns of component design, can be reused as much as possible.

C. Validation

Validation - the matching of a software model with the real world model - is assisted in the component approach by a formal modelling process: mapping real world objects and their behaviours to software components. CBSE approaches perform this process incrementally: components are designed to model parts of real objects and these components, once validated, are used to design more and more complex objects. The key aim is that a complex object will be 'correct by construction' - it will correctly model the large object's behaviour by faithfully modelling it as an interaction of components which are themselves correct. CBSE enables the re-use of components whose behaviour is well understood (*aliter* whose semantics are well-defined) and known to meet a previously stated set of requirements. Requirements engineering is an expensive process in itself - requiring many iterations between users and design teams before they are correct. Note that re-use of a component implies that its previously developed requirements may also be re-used! From an economic point of view, this means that the iteractions between users and design teams can start with refined requirements for available components and can reduce to determining whether the existing component meets the current requirements, or whether some specialization of it will be needed.

D. Implementation

Formal models for component architectures simplify the implementation phase by providing patterns or templates for code to be written. By removing some of the freedom allowed to the writers of code, their ability to write in 'innovative' (read 'dangerous') styles is reduced. Automatic generation of code (or at least skeletons of code) from modelling languages (*e.g.* UML) further assists in the production of uniform, easily maintained code. However the component based approach to software construction also provides opportunities for visual editors to be used for the rapid production of new systems from components. The 'bean box' which may be used to construct systems from Java beans[14] is one example. The locally developed VICON system (*cf.* section III-D.1) is another in which visual tools have been developed to overcome limitations that conventional approaches to integration of large systems.

D.1 Development Environments

"Integrated" Development Environments
Existing so-called 'Integrated Development Environments' (IDEs) generally integrate syntax-sensitive editors, compilers, linkers and debuggers fairly well: there are a host of commercial IDEs which can be used to build individual components and thus component-based systems. However, CBSE depends heavily on coupling components through defined, published interfaces and development environments which support formal models can provide a significant productivity boost.
Large systems may contain several thousand components and thus tools which provide good support for the definition and cataloguing of the many small components from which a large system will be constructed are important. Without such tools, developers of large systems are likely to needlessly replicate much development work in producing more than one component to model a group of similar objects in a system.
Visual Development Environments

The VICON system targets the largely neglected problem of systems integration: the task of integrating or connecting the various parts of an application to form a system as a whole. Surprisingly, for much of the past forty years, this aspect of system design has been paid little attention: it has been seen as an afterthought or chore that must be performed once a system has been developed. It is now obvious that as systems increase in size and complexity it is no longer possible to leave integration concerns to chance at the end of the development process.

Current Limitations

Systems integration is limited in scope by its textual representation and lack of high-level data-oriented support.

Firstly, the use of scripting and programming languages as a means of expressing systems integration greatly restricts the integration's scope and flexibility. Rather than maintaining the spatial aspects of integration it is necessary to sequentially lay out the integration concerns using a textual representation.

Wiring standards and Architectural Design Languages (ADLs) attempt systems integration through mediation protocols, while binary integration mechanisms rely on complex and tightly coupled programming and operating system environments. Both approaches attempt to marry the integration back into the implementation environment, again restricting the scope and flexibility of systems integration. To overcome these difficulties the systems integration must remain separate from the implementation concerns if we are going to deliver better data services.

Furthermore, current systems integration methods fail to take into account the fundamental currency of exchange - the data. Rather than focusing on the data and data delivery, most integration standards have strengthened the relationship between systems integration and the actual development infrastructure. This limits the integration scope, as it does not employ its own computational model. It also forces the integration back to a textual representation.

Two steps need to be taken to overcome current systems integration problems:

- Firstly, the integration concerns need to be separated from the implementation concerns of an application. Not just at the design and specification stages as suggested by MILs but throughout the design, implementation and deployment of a system. Once systems integration is removed from the functional concerns of an application it is then possible to introduce a separate computational model capable of delivering data oriented facilities, such as complex data flow and architectural support, to the system's components.
- Secondly, rather than persisting with a restrictive textual representation, separation allows us to introduce a visual notation. A data-oriented computational model can then support the visual design. Unlike current computational models, which focus on functionality rather than integration, a data-oriented computational model could support visual architectural modeling and data flow management. Such an approach can maintain the spatial nature of systems integration and the high-level design of the system.

The VICON Approach

In the VICON model, a component is by definition a self contained, self-sufficient application capable of running in isolation: the only visible aspects of a component are its external data requirements or inputs and outputs, its public interface. Instead of concentrating on a binary integration standard, the VICON system focusses on data delivery and separation of concerns. Rather than the individual components having to communicate directly with each other via a coding representation, VICON places a *Data Exchange* or a 'thin data service layer' between the individual components making up the application as a whole. This thin data service layer handles the data delivery and execution order and hence separates the systems integration concerns from computational concerns. We can visualize this difference in approach as follows:

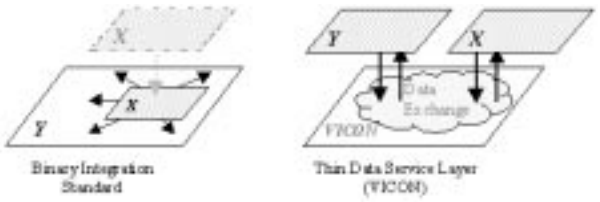

Fig. 13. The VICON approach

VICON allows a system integrator to edit a diagram which clearly shows the complex relationships that may exist between components. Maintenance of VICON 'programs' is simple, fast and less prone to error because all the relationships between objects involved in a change are clearly shown and the system is able to verify that many necessary details of the change have been correctly treated.

E. Verification

Verification - the process of determing that a component has been written to correctly reproduce the behaviours in its specification - is a labour-intensive (and thus expensive) process which commercial pressures to provide solutions to customers usually ensure is only ever half-complete. As with validation, the breakdown of complex components into loosely-coupled components which can be tested independently has a major impact on verification effort. This will occur in two ways:

- Independent or loosely-coupled components may be independently tested. The number of test cases which must be dealt with increases multiplicatively as system complexity increases. If we have a system composed of two components, one requiring testing with m sets of inputs and a second requiring n sets of inputs, then to test the system completely, we must perform mn tests. Usually the required number of tests increases rapidly to the point where comprehensive testing at the system level is practically impossible due to the sheer number of tests required. Thus as systems become more complex, the ability to test individual components thoroughly becomes more and more vital - at least we might have confidence that the 'sub-assemblies' are correct and therefore it's possible that the composite system will also be correct. (Although correct functioning of the composite system is clearly not guaranteed, the converse proposition - that a system composed of unreliable components will be unreliable - is invariably true!)
- Components which are designed for re-use clearly justify more verification effort than those which will be used in a single system. Re-used components will also see additional testing associated with the new systems in which they are embedded. If previously undetected errors are discovered, the substitutablity property of a component ensures that all systems may readily see the corrected component.
- Once repositories of reliable components become common, system integrators will be able to devote more effort to integration testing.

Testing Effort

The style in which a component is written has a major effect on its testability. Stateless or pure dataflow components are the easiest to test. Having no persistent state to influence the result of a second application of the same inputs means that each set of inputs needs to be applied to the function once only. This dramatically reduces the number of tests required and therefore the effort and cost of verification.

As components acquire persistent state, then the current state of any instance of the component has to be considered as part of the input value set. This generally contributes to the multiplicative explosion of the number of tests and hence the cost of verification.

This factor was one of the key reasons that we included simple dataflow functions as components in our taxonomy of components. System developers will only benefit from CBSE approaches if the components are reliable. This implies that component developers understand how to build components that are easily verified. Reduction of the amount of persistent state that must be considered when testing to the barest minimum - compatible with producing a component with some useful capabilities! - is vital to our chances of achieving any satisfactory level of verification and reliability.

E.1 Ceritification

One of the benefits of the component approach is the ability to use commercial off-the-shelf (COTS) components to build systems faster. However, the reliability of such components is a significant concern: Voas[15] notes that 'it is never clear how rigorous that testing (by the supplier) was and what generic profile was used.' He then describes a certification procedure in which the first step is 'black-box testing'. Software Engineering Australia is supporting a project to establish a library of COTS components *that will be accompanied by test certificates*[16]. These test certificates will enable a user of a component to (a) rapidly determine that a component operates according to specification in the target environment and (b) how rigorously the component has been tested.

F. Maintenance

One of the long-standing problems with maintenance of existing software is poor, incomplete or non-existent documentation. However, if one is going to use a component obtained as a 'black box' from a third party, then it is reasonable to presume that sufficient documentation will have been available at the time one decided to use the third-party component to justify the decision to use it.

As noted in the discussion on testing (section III-E), re-use implies more continual testing, increasing the probability that errors will surface *before* the code becomes legacy code that no-one understands! An error discovered early in a system's life is likely to have a lower economic impact. Replacing faulty components which are designed to be substitutable is also easier and less costly than in tightly coupled systems where substitutions of new modules may have far-reaching consequences.

F.1 Substitution capabilites: threat or opportunity?

The ability to substitute one component for another has major economic ramifications in the maintenance phase. On the one hand, a system may be adapted to a new environment without costly and delay-inducing re-writes by the subsitution of new modules which manage the system's interface to the new environment. This can result in enormous benefits to users, who are no longer constrained to continue to use legacy hardware or software because the cost to update an extant large system would be prohibitive. Thus investment in large and complex software systems may be amortized over much longer periods and maintenance costs associated with keeping abreast with technological change can be minimised. However, the substitution of a new module has important implications for the reliability of the new system.

- Will the new module function *exactly* as its predecessor did?
- Will it introduce new problems?
- To avoid this possibility, do we need to re-verify the whole system (at considerable expense)?
- Will the new system have lower reliability (by any measure) than its predecessor?
- Did we substitute an uncorrupted copy of the new component[17]?

If the new component has received the same thorough independent verification as the one it replaces, then we may expect that, because the interface has necessarily been well-defined (to allow substitutability in the first place) that a new component, independently tested to the same standard as the one it replaces, will perform identically (if it simply involves computation) or equivalently (if, for instance, it involves interaction with an external peripheral device with equivalent, but not identical, characteristics to the original one.) This emphasises the need to have access to the test certificate for a component so that it may be verified before it is incorporated in a new system - a primary aim of the Software Component Laboratory's library[16] will be to package test certificates with each component. Zhong and Edwards[18] describe an alternative 'sandbox' approach to testing new components: they build a harness (the sandbox) to assess the risk of substituting the new component.

IV. Standards?

It is clear that distributed component technologies require inter-operation standards: although we currently have a pitched battle between competing standards (CORBA and DCOM), bridges exist and it is unlikely that any new standard will gain enough support to further complicate this picture.

It is in the area of the components themselves that the industry would see most benefit from standardisation. Imagine a world in which a standard interface to a printer was described in, say CORBA's IDL, and all printer manufacturers supplied a conforming driver. Programs using printers in this world would be trivially ported from one system to another with no changes.

Java has led the way in this respect by defining a (daily growing!) list of API's for performing an enormous number of tasks. However the difficulty in specifying a sufficiently powerful, yet sufficiently re-usable interface has been amply demonstrated in the decision to replace Java's original windowing toolkit (`awt`) with the Swing interface after a very short time in the field.

V. Future Directions

Development environments will continue to expand in their ability to assemble powerful systems rapidly and easily. Component software will, of necessity, play a key role: without well-defined interfaces and capabilities, component assembly systems cannot produce useful systems. Visual development tools like the Java Bean Box and VICON will certainly become more and more common. Most high-level system designs already use diagrams in order to facilitate rapid comprehension of complex interactions: visual tools will simply make the textual duals of these diagrams redundant - at least in terms of the design effort to produce them: the tools will do that for the designer.

One exciting possibility is a merger of some of the ideas in intelligent agent technology with the substitutability capabilities of components to produce software systems that have greater adaption capabilities than we have indicated here - software systems that continually evolve with new components being continually selected to optimally handle changing environments.

VI. Conclusions

There are some clear and obvious benefits for a software developer who adopts a component-based approach:

- Reuse leads to faster production cycles
- Design tools (*e.g.* VICON) can work with a rapidly expanding library of components from a plethora of sources to expand their domain of operation
- Reuse of verified components leads to more robust products.

In addition, there are some more subtle benefits:

- Substitutable components allow systems to configure themselves dynamically,
- Dynamically configured systems require less maintenance effort as environments change and system requirements expand: new components which use new software or hardware to provide additional or improved capabilities may be automatically loaded,
- Different models for the way in which software is purchased become possible: *e.g.* 'pay-per-use' becomes a practical possibility, allowing older less-efficient components to be immediately discarded in favour of more effective ones with no financial penalty.

As a final word to counter the doubters, we quote from Szyperski[2]:
".. from a purely formal point of view, there is nothing that could be done with components that could not be done without them. The differences are concepts such as reuse, time to market, quality and viability."
Thus we can survive without component technology, but at enormous cost - firstly, in recreation and regeneration of similar tools over and over again and secondly, in verifying all these tools each and every time one is created or, much much worse, relying on inadequately tested software.

VII. Glossary

As with all new computer technologies, the rapid development pace and fierce competition between rival companies prevents effective standardisation until the new technology has matured and all the competing players find that the only way to increase the technology's impact is to standardise. This leads naturally to a sometimes bewildering forest of new terms - many of which are synonomous - or nearly so. The following glossary attempts to shine some light into this forest.

The demise of CI Labs has meant that many of these terms are now of mainly historical interest, but they have been included since they do occur extensively in early literature.

ADL - see Architectural Description Language

Architectural Description Languages (ADL) - a language which rigorously defines a software system architecture - or the way in which software components interact.

Binary Format - A component which has been processed partially (*e.g.* compiled to Java bytecode) or fully (*e.g.* an executable application or shared library) is said to be in binary format as distinct from a human-readable text format.

Binary Level Interoperability - Components need not be part of the same program and may interoperate as binaries which may have been compiled for different architectures from different high level languages, *e.g.* a component written in C and compiled by a C compiler on a Unix machine may interact with one written in Visual Basic and compiled on a Windows machine.

CBSE - see Component Based Software Engineering

CI Labs - an independent, non-profit consortium which includes Apple, IBM, Taligent, Novell and SunSoft. CI Labs maintained the OpenDoc standard (*qv*) until it was disbanded in 1997.

COM *see* Component Object Model.

CORBA *see* Common Object Request Broker

COSS *see* Common Object Services Specification.

Categories - sets of components that supply a required set of interfaces and thus may be substituted for each other. Microsoft's COM IDL has recently been extended to support such categories.

Component Based Software Engineering (CBSE) - this term incorporates everything that the original software engineering discipline implies but with an emphasis on the use of existing components (possibly from third parties) rather than building all parts of a system from scratch. A number of texts and paper collections describe the scope of the term more fully[19], [20] (and other papers in the same conference), *etc.*.

Component Object Model (COM) - a set of specifications which allow compiled components to interact (binary level interoperability), developed by Microsoft. *See also* System Object Model.

Common Object Services Specification (COSS) - part of the Object Management Reference Architecture defined by the Object Management Group - the Common Object Services are a collection of services that support basic functions for using and implementing objects. For example, conventions for creating, deleting, copying and moving objects are defined in the Life Cycle Service, one of fifteen defined services within COSS[21].

ComponentGlue - a collection of tools which allow OpenDoc components to be placed in OLE documents and vice versa.

DCOM *see* Distributed COM.

DLL *see* Dynamically Linked Library.

DSOM *see* Distributed SOM.

Despatch Tables *see* Method Despatch Tables.

Distributed COM (DCOM) - an extension of Microsoft's COM which supports distributed objects.

Distributed SOM - a variant of IBM's System Object Model (*qv*), which manages distributed objects.

Dynamically Linked Library (DLL) - the term used in Microsoft Operating Systems for libraries which are linked into a program when it executes rather than when the program's executable code is initially built.

EJB *see* Enterprise Java Beans.

Enterprise Java Beans (EJB) - a set of specialised Java Beans (*qv*) designed for use in server and middleware environments. They are essentially just a collection of Java Beans designed for commercial transaction processing applications.

Event Adapter - used in Java (and Java Beans) to provide a convenient and simple method of implementing an event handler which only needs to handle a subset of possible events generated by an object.

Fragile Base Class - The inability to modify a class without recompiling clients and classes derived from that class is called the Fragile Base Class problem. Use of method despatch tables (*qv*) can alleviate the syntactic version of this problem.

IDL *see* Interface Definition Language.

IIOP *see* Internet Inter-ORB protocol.

Inclusion Polymorphism - Polymorphism (or methods which have the same name but different uses) which arises from inheritance: all sub-types (or specialisations) of a class inherit all the methods of the parent. The inherited methods may be applied to objects of the sub-type as well as objects of the original class.

Independent Extensibility - a system is independently extensible if independently developed extensions can be combined, *e.g.* an operating system can be extended with independently developed applications, a browser can be extended with plug-ins.

Inheritance - a key capability of object-oriented systems: a class may *inherit* all the attributes and methods of a parent class. The inheriting class is called a *specialisation* of the parent or *super* class.

Interface Definition Language (IDL) - a formal language for describing a component's interface in a way which is independent of the programming language used for the component itself. ORBs rely on IDL interface descriptions of objects to transmit messages between objects.

Internet Inter-ORB protocol (IIOP) - defines data formats (the Common Data Representation or CDR) and message types used when ORBs exchange messages. IIOP is part of CORBA and is transparent to CORBA users[22].

Interoperable Object Reference (IOR) - a reference or handle for an object passed between ORBs using the IIOP.

Interface Definition Language -

Introspection - refers to ability of a component to supply information about its capabilities, in particular the interface that it provides to other components.

JVM *see* Java Virtual Machine

Java Virtual Machine - All Java programs are compiled as if they are going to run on an abstract machine, the Java Virtual Machine. A program on the actual machine reads instructions for the JVM and 'executes' them for that machine.

Late Binding - components may be incorporated into (or *bound to*) programs which use them at run time, rather than statically bound to a program by a linker. Late binding allows substitution of components which are incorporated in programs without re-compiling or linking the original program.

Method Despatch Tables - tables generated by the compiler or loader which enable a method call to be despatched to the correct super-class. Run-time initialisation of despatch tables is the key to late binding (*qv*) and dynamic substitutability.

Microkernel - Minimal operating system (or kernel) which provides only basic services and leaves application level servers to provide most of the required capabilities. Microkernels provide the potential to build efficient custom systems from components.

Middleware - any program that serves to connect other programs. Object Request Brokers are the most important examples of middleware in this context.

Module Safety - a component must specify explicitly which services (from the system or other components) which it needs to access in order to be a safe module.

OLE *see* Object Linking and Embedding

OMG *see* Object Management Group

ORB *see* object request broker.

Object Linking and Embedding (OLE) - a standard for compound documents defined by Microsoft - it is a collection of COM interfaces.

Object Management Group - a consortium of over 400 software companies whose charter includes the establishment of industry guidelines and specifications to provide a common framework for application development. OMG is responsible for the definition of CORBA (*qv*).

Object Request Broker (ORB) - a 'middleware' program that enables objects to interact; it receives requests from objects for information about or services from other objects with which they may interact.

Open Scripting Architecture - the means by which parts of documents interoperate in OpenDoc (*qv*).

OpenDoc - a set of standards for compound documents maintained by CI Labs (*qv*).

out of band - a term applied to messages that are sent on a logically separate channel linking two objects. Messages on this channel may arrive ahead of messages despatched at early times on the normal channel: thus out-of-band messages can be used for alert and error messages.

Parametric Polymorphism - is the polymorphism which results when a method works in the same way on a range of types. The "legal" types will normally have to have some common sub-structure and the compiler (or, *in extremis*, the programmer) will need to be able to tell the method what type is actually being passed.

Polymorphism - there are two main types of polymorphism, Parametric Polymorphism (*qv*) *and* Inclusion Polymorphism (*qv*).

RMI *see* Remote Method Invocation

RPC *see* Remote Procedure Call

Reflection interface - an interface which allows a user to discover characteristics of a component at run-time.

Remote Method Invocation (RMI) - A Java object uses RMI to invoke methods on objects residing on different hosts. An example of Java's RMI is found in figures 4 and 5 of this paper.

Remote Procedure Call (RPC) - a mechanism by which one component invokes a specific procedure or handler on another component rather than simply sending a message and using the state of the receiver to determine how to process the message.

Serialization - in order to transmit an object to another processor copies of all the objects which are attributes of the object to be transmitted must be made so that the object received by the remote processor is complete, *i.e.* can obtain values for all its attributes locally.

SOM *see* System Object Model.

System Object Model - a language-independent CORBA-compliant object request broker developer by IBM. *See also* Common Object Model.

Work-flow Computing refers to the control of task sequences across disparate objects when operations are subject to synchronisation constraints [23].

References

[1] Alan W Brown and Kurt C Wallnau, "The current state of CBSE," *IEEE Software*, vol. 15, no. 5, pp. 37–46, sept/oct 1998.

[2] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[3] Nilesh Sampat, *Components and Component-Ware Development; A collection of component definitions*, http://www.cms.dmu.ac.uk/ nmsampat/research/subject/reuse/components/index.html, 1998.

[4] Phillippe Krutchen, *Software Component Definition*, *cited in* [1], 1998.

[5] Gartner Group, *Software Component Definition*, *cited in* [1], 1998.

[6] Ashley Aitken, *Component-Based Software: A Business Perspective*, Software Engineering Australia (WA), 1999.

[7] Tim Hopkins and Chris Phillips, *Numerical methods in practice: using the NAG Library*, Addison-Wesley, 1988.

[8] John Morris, *Objects First; A First Course in Program Design using C*, http://ciips.ee.uwa.edu.au/ morris/Year1/CLP110, CIIPS, University of Western Australia, 1999.

[9] SunSoft, *Java 2 SDK, Standard Edition*, java.sun.com/products/jdk/1.3/, 1998.

[10] Scott M. Lewandowski, "Frameworks for component-based client/server computing," *ACM Computing Surveys*, vol. 30, no. 1, pp. 3–27, 1998.

[11] Object Management Group, Inc, *CORBA/IIOP 2.2 Specification*, http://www.omg.org/library/corbaiiop.html, 1998.

[12] David Krieger and Richard M Adler, "The emergence of distributed component platforms," *IEEE Computer*, vol. 31, no. 3, pp. 43–53, 1998.

[13] Rational, *Rational Rose*, http://www.rational.com/products/rose/index.jtmpl, 1999.

[14] Sun Microsystems Inc, *JavaBeans*, http://java.sun.com/beans/index.html, 1999.

[15] Jeffrey M Voas, "Certifying off-the-shelf software components," *IEEE Computer*, vol. 31, no. 6, pp. 53–59, 1998.

[16] John Morris, Gary Bundell, Peng Lam, and Gareth Lee, *Component Test Bench*, Work in progress, SEA(WA) Software Component Laboratory, CIIPS, University of Western Australia, 1999.

[17] Ulf Lindqvist and Erland Jonsson, "A map of security risks associated with using COTS," *IEEE Computer*, vol. 31, no. 6, pp. 60–66, 1998.

[18] Qun Zhong and Nigel Edwards, "Security control for COTS components," *IEEE Computer*, vol. 31, no. 6, pp. 67–73, 1998.

[19] Alan W. Brown, *Component-Based Software Engineering, Selected Papers from the SEI*, IEEE Computer Society, Los Alamitos, 1 edition, 1996.

[20] Klaus Bergner, Andreas Rausch, and Marc Sihling, "Componentware – the big picture," in *Proceedings of the International Workshop on Component-Based Software Engineering*. Apr 1998, IEEE.

[21] Object Management Group, *CORBAServices: Common Object Services Specification*, "http://www.omg.org/library/csindx.html", 1998.

[22] David Curtis, Christopher Stone, and Mike Bradley, *IIOP: OMG's Internet Inter-ORB Protocol; A Brief Description*, "http://www.omg.org/library/iiop4.html", 1999.

[23] Richard M Adler, "Emerging standards for component software," *IEEE Computer*, vol. 28, no. 3, pp. 68–77, Mar 1995.