SUPPORT FOR COMPONENT BASED SYSTEMS: CAN CONTEMPORARY TECHNOLOGY COPE?

I Coutts and J Edwards
MSI Research Institute
Department of Manufacturing Engineering
Loughborough University, Loughborough
Leicestershire LE11 3TU, UK
Tel: +44 1509 228250 Fax: +44 1509 267725
Email j.m.edwards@lboro.ac.uk or i.a.coutts@lboro.ac.uk

Abstract

Information Technology (IT) systems can generally be described through subdivision into components whose level of abstraction allows the people who are involved with their creation and maintenance to better understand the system. However, this component sub division seldom exists beyond the conceptual system description. Implementation of the system based on this component level decomposition offers many advantages in terms of re-use, system maintenance and general support for change.

The work described in this paper has investigated the level of support for component systems that is provided by available distributed object technology. The work covers the lifecycle phases from conceptual design to implementation using technologies which include Smalltalk, IDL, C++, and CORBA.

The work concludes that important features for enabling component implementation are not supported by the object paradigm and are still missing from available distributed object support products. As such it is necessary for the system creator to design in features which support the component paradigm. Without this additional infrastructure it is highly likely that system implementations which claim to be component based will still be monolithic and subject to all the problems of today's legacy systems.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35390-6_58

L. M. Camarinha-Matos et al. (eds.), *Intelligent Systems for Manufacturing*
© IFIP International Federation for Information Processing 1998

1. MANUFACTURING SYSTEM SOFTWARE

Software systems created to support the control of a manufacturing business can generally be regarded as being Information Systems. Jackson states that [Jackson 1995]:

"In its simplest form, an Information System provides information, in response to requests about some real world domain of interest"

Figure 1 illustrates this, highlighting the fact that the IT system models the real world but is not directly coupled to it. Indeed, most manufacturing control software does not control the manufacturing process, it provides information so that a person or machine can control it. To enable an IT system to provide relevant information we must produce a model of the real world and embody that model in the system. In effect, the system becomes a simulation of the real world, and derives its information directly from its model, and only indirectly from the real world itself.

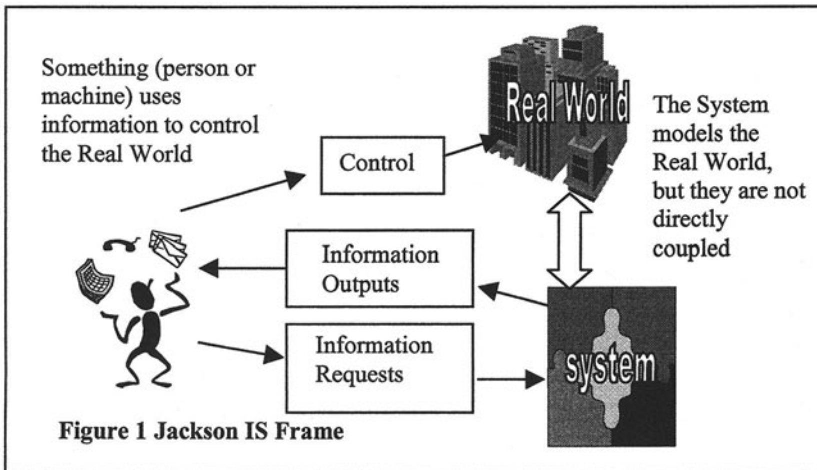

Figure 1 Jackson IS Frame

The real world for manufacturing industry has dramatically changed over the past two decades, from a very stable world of established and slow changing practices to the current position of extreme turbulence. Pressures from globalisation, decentralisation, customisation and the acceleration of the rate of business change itself, driven by technical, social and organisational factors all demand a requirement for flexibility from the business [Taylor 1992]. If the IT systems which underpin contemporary business processes are simulations of these real world businesses they must in turn support a high degree of flexibility. Unfortunately recent experience has shown [Ganti 1995] that the inherent flexibility of software has led to the creation of highly complex monolithic bespoke solutions which turn out to be one of the least flexible elements within a business.

2. MONOLITHIC SYSTEMS

Manufacturing Information Systems are commonly created using a data base and associated management system structured using a 4 GL database language. Here a high level language can be used to define transaction processes linking data fields within the system. In this way, a working system specific to a particular domain or an individual companies requirements can be rapidly created. However, the

complexity of the system soon becomes very high as it mirrors the complexity of the real world. Although complex, the real world is comprehensible as it is made up of identifiable component parts combined through structured relationships. In manufacturing these include such things as people (in sales, design, engineering etc), orders, products, parts etc linked through defined relationships such as order processes or product designs. In the equally complex IT system implementation all that is generally visible is a collection of programs which reflect the computational abstractions imposed by the architecture of the underlying computer hardware.

The conceptual descriptions of these monolithic systems as portrayed in sales literature are often modular in nature as they match the natural component breakdown of the real world they model. The user is offered the advantage of being able to chose specific modules generally based around some core system such as a general ledger facility. However this vision of modularity seldom extends beyond the conceptual system description with hard links between modules introduced in design and implementation which make it impossible to reduce complexity through sub division in these phases of the system development lifecycle.

The solution proposed by many in the software industry [Cox 1987, Edwards 1996] is to carry the component philosophy through design to implementation, using the real world and its component parts as a model. Distributed object technology has been championed [Orfali 1996] as the medium capable of supporting this notion. The experiment described in this paper tests the hypothesis that contemporary distributed object technology is capable of supporting component system implementations.

3. COMPONENT BASED SYSTEMS

The creation of modular component parts that provide a service but encapsulate their internal complexity, and the use of these components to build tangible products is well established in modern industry. The computer hardware industry being a classic example. The architectures that enable this are seen as key to enabling cost and time reduction. In modern software development however compliance to rules is often seen as a constraint, and the pace of change in the industry has provided some justification to the belief that it can stifle innovation. However the emergence of standard distribution technology from low level communications protocols [Lefffler 1989] to higher level object request broking services [OMG 1994] has provided the opportunity for the industry to take advantage of a component based approach. If this opportunity is taken up it can offer the same benefits as componentisation has provided in other industries i.e.:

- Reduction in time to market through re-use;
- Lowering of costs through multiple use of standard components;
- Improved quality through the use of tried and tested components;
- Simplification of the system delivery process, and;
- Outsourcing of specialist skills.

The key requirements for useful components are no different from the two long standing principles of software engineering, a low degree of coupling and a high degree of cohesion [Chidamber 1994, Hitz 1996] which are often used to design objects when using an Object Oriented approach. Cohesion, essentially a measure of how well the internal functions of the component are related, demands a logical grouping of functionality. This logical grouping can be regarded as a design issue internal to the component, the only external issue here is the access to this functionality which demands the building of sensible well defined interfaces to the services provided by and encapsulated within the component. Coupling however, is a measure of the external dependencies of the component defined by the number of

links a component has to other components within a system. A low degree of coupling demands minimised interaction with other components in the system. Coupling then involves issues external to the component. As an external issue coupling demands support from the environment in which the components exist and to maximise the degree of flexibility achieved by component systems this support should provide for loose coupling. Loose coupling demands that the components in a system are not linked directly to form a complex network of interactions, but are linked in such a way that they remain as separate abstractions, as identifiable as their real world counterparts.

Ultimately loose coupling should provide the ability for developers to create components in complete isolation. System builders would then bring these components together to form an integrated system where each component in the system has no prior knowledge of the other components in the system. Whilst this is easily stated as a system requirement it is not easily realised within a working system, and as such this level of support for loose coupling is currently the subject of a number of research initiatives including those at MSI [Edwards 1996]. In the work described in this paper a general solution is not proposed, the solution path is to create a system design that addresses the provision of loose component coupling. In general each phase of the system creation lifecycle places requirements on the subsequent phase. Loose coupling defined as a requirement at the conceptual design level has generated a requirement which is satisfied through the provision of "mediation" facilities devised during system design and carried through to implementation. Throughout the system construction life-cycle requirements "trickle down" as each phase defines requirements which must be satisfied by the subsequent phase.

4. REQUIREMENTS FOR A COMPONENT BASED SYSTEM

In order to provide the loose coupling required between the components which comprise a system, a number of issues must be addressed:

- Each component should posses a well defined role or cohesive set of roles realised through the provision of services, such that any peer component can make use of these services;
- These services should be easily accessed, without the requirement for detailed knowledge of underlying computational mechanisms;
- The definition of the component coupling required to produce a system should not reside within the components themselves, as these are system issues external to the component.

As introduced in the previous section, a correlation exists between the object oriented approach to software construction and a component approach to system construction. An object is an encapsulated entity that possesses an identity and an interface to its functionality. Typically an object may manage a resource, or just its own data, which can only be accessed using the object's published interface. Other objects can either invoke the services it provides through this interface or reuse the complete object by inheriting from it [Booch 1994]. Using these means, objects which invoke services on each other can be combined to create systems.

Hence, objects can be used to meet the requirement of providing a well defined role or cohesive set of roles. In addition, as object services are usually invoked by a method (or function) call or by receipt of a message, a suitable abstraction exists for hiding any underlying mechanisms required for object interaction. This abstraction allows the composition of well defined interfaces to the services an object encapsulates.

With the emergence of distributed object technology, systems can be constructed as a set of interacting objects without any concern for details such as:

- on which host computer the objects reside;
- the operating system resident on a host computer;
- the programming language used to implement objects.

When using distributed object technology, components use remote procedure and function calls in order to interact with each other. A C++ program (for example) can use distributed object technology to invoke functions on a object executing on another host within a distributed system. This achieves a high degree of transparency because a C++ programmer can make a standard C++ function call to invoke a remote object method. This allows a familiar programming paradigm and syntax to be used to create distributed programs.

To enable the implementation of different objects using different programming languages, an abstract representation of their interface must be produced. The Interface Definition Language (IDL) is a standard language defined by the OMG, for defining distributed object interfaces. Having used IDL to define an object's interface, a programmer is then free to implement the object using any suitable programming language. Correspondingly, a programmer who wishes to use the services of an object, can employ any programming language to issue remote method calls to the object.

Realising system components as objects may provide the cohesion of services we require, however, relying on the object paradigm alone to implement the proliferation of interactions that are required to define an object based system leaves the degree of flexibility achieved entirely on the hands of the system programmer. When a large number of object interdependencies exist within a system, the system tends to become monolithic, due to the difficulty in isolating the role of any one object. Object systems based on this approach offer little advantage over conventional legacy software systems. So the object oriented approach in itself does not fulfil the third of our system component requirements, this must be tackled by some other means. The approach adopted by the authors which provides additional infrastructural software elements to support the systems programmer in creating loosely coupled component based systems is discussed in the following sections.

5. SYSTEM DESIGN

The system considered in this paper is a manufacturing information system as introduced in section 1. Such a system must model the natural component breakdown of the real world. For example, central to many manufacturing organisations is the concept of parts, together with the concept of buying and holding parts as stock, the construction of more complex parts or products from a collection of parts described within a design and the storage and sale of these products. A component based system which models some of these concepts can be described as in Figure 2, where Sales Order Processing deals with the buying and selling operations, Inventory holds details of stock levels and the Bill of Materials relates parts to the more complex products. Each of these information system entities must interact to form the IT system.

Having established that the principles embodied by the object oriented approach could be used to help realise this component based design, the authors required an appropriate object oriented notation and support tool in order to enunciate and evaluate the design. The Smalltalk programming language was chosen, primarily for the following reasons:

- its adherence to the Object Oriented paradigm

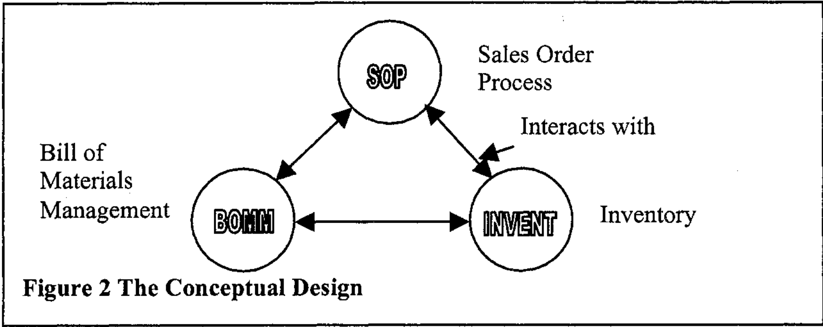

Figure 2 The Conceptual Design

- its compact notation, and
- its ability to simulate particular scenarios within the design

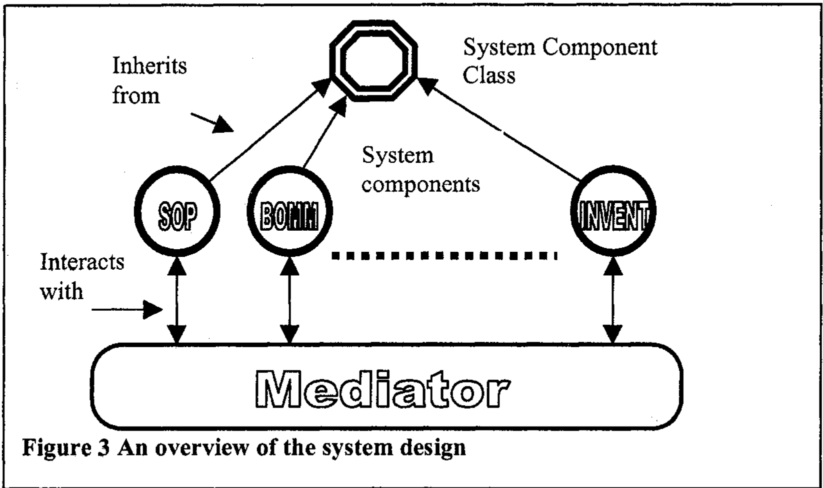

Figure 3 An overview of the system design

The requirement for loose coupling of components, specified in the conceptual design, is not inherent within the object paradigm and its supporting language, nor is it yet supported by the general purpose products that are emerging to underpin distributed objects [OMG 1994]. Loose coupling is therefore a design requirement and mechanisms to support it must be devised by those responsible for systems design where these mechanisms must generate a clear set of requirements for system implementation. An overview of the structure of the solution proposed by the authors is depicted in Figure 3. The approach is known as mediation [Gamma 94], the intent of which is to:

"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently"

A system component mediator should be responsible for controlling and co-ordinating the interactions of a group of components by serving as an intermediary. Hence components are not known to each other, they are only known to the mediator. This approach reduces the number of interconnections and controls the build up of complexity in the system which is responsible for the creation of monolithic systems.

To implement this approach the authors have introduced two infrastructural elements to the component system namely: a mediation facility, and; a system component class.

6. THE MEDIATION FACILITY

The mediation facility performs two main roles, it holds a definition of the system component coupling, and it brokers component message requests through its knowledge of the component coupling within the system.

Hence all interactions between system components are configured within the mediator. Figure 4 provides an example of the operation of the mediation facility. Here a system interaction comprising a request from the SOP component for a service provided by the INVENT component is brokered via the Mediator.

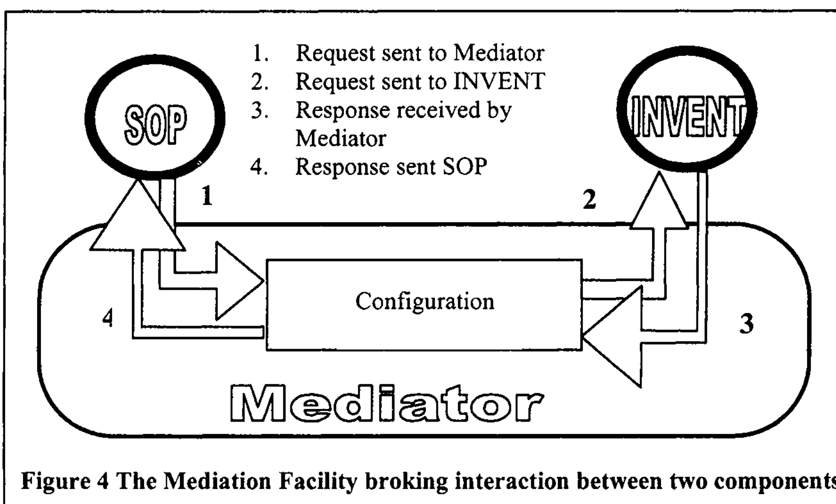

Figure 4 The Mediation Facility broking interaction between two components

Although the example in Figure 4 may seem to add a lot of operational overhead to what could be a simple system call from the SOP component direct to the INVENT component, it is important to remember the motivation for such an approach. The sales and inventory components do not directly interact, indeed the message signature brokered by the mediator need not be the same for both components. This allows for the mediator to broker service requests at a semantic level, and not just at a syntactic level. The configuration facility in the Mediator provides a semantic level understanding of the services within the system. For example a service request issued by the SOP component to obtain the level of stock of a particular part such as

```
stocklevel: flange
```

may be issued to the INVENT component by the mediator as

```
partRequestLevel: flange
```

The mapping of one syntax for the service to another, being achieved within the configuration utility in the Mediator. The service response may also be modified in a similar manner. This de-couples the two components to an extent where they must only share the semantics of their interactions and a common means to interact. The latter being the role of the system component class.

This level of configuration is not currently supported within contemporary Distributed Object Technology and so must be built into a mediation facility by the system implementor. However, the OMG's Trading service specification has recently been approved and is aimed at supporting the concept of system wide mediation service provision.

The second role performed by a mediation facility, the brokering of component interactions within a distributed system, is well served by Object Request Brokers as described in the previous section. The implementation of a mediation facility requires a layer of configuration functionality to be added to the services provided by an Object Request Broker. This has been achieved within a prototype solution using the following methods

```
add: method forObject: anObjRef supplier: aSupplier method: aMethod
args: anArgMap block: aBlock
"This method stores configuration information within a local
database"
```

This allows configuration information to be added to the mediator as follows

```
mediator add: #stocklevel: forObject: sop supplier: invent method:
#PartRequestLevel: args: #(1) block: [:ret|]
```

During system execution, when a message such as

```
stocklevel: flange
```

is sent to the mediator a method

```
aMessage:fromObject:
```

accesses the configuration information and performs the following:

- identifies the appropriate destination system component for the message, i.e. INVENT;
- transforms the message syntax to that required by the destination component i.e. partRequestLevel: flange;
- sends the message via the ORB to INVENT;
- transforms the response message into the required syntax of the message originator, in this case no transformation is required;
- sends the transformed response message back to the component SOP.

This facility enables any system component to interact with other system components without it containing details of other component interfaces.

7. THE SYSTEM COMPONENT CLASS

Using the solution proposed by the authors all components within the system require access to the mediation facility in order to interact with other system components. To achieve this a single system component class was devised. The intention here is to use the object oriented property of inheritance as a means of providing all system components with the capability of using the mediation facility Inheritance is the mechanism that allows the designer to create new child classes from existing parent classes. Child classes inherit their parent services and data structures. The designer can then add new services or override inherited services so the child class becomes a specialisation of the parent class.

Hence within the system solution proposed in this paper all components are a specialisation of the system component class where this class directs all service requests to the mediation.

During system implementation the construction of a system component class is required, this class encapsulates the means of interacting with the system mediator. The provision for constructing object classes and implementing object inheritance is well catered for by object oriented programming languages, only the means of interaction with the mediator has to be implemented. This has been achieved within a prototype solution by creating a systemComponent class which supports two instance variables 'myName' and 'myMediator' and implements the Smalltalk method 'doesNotUnderstand:' in order to divert messages to a mediation facility.

The class supports system component creation by providing the method

```
new: instanceName mediator: aMediator
       ^(super new) name: instanceName; mediator: aMediator
```

which identifies the name of the component and its mediator. Within Smalltalk when an inappropriate message is sent to an object the message is encoded in an instance of the class Message and sent back to the same receiver as the parameter to 'doesNotUnderstand:'. This method is implemented by the system component class as follows.

```
doesNotUnderstand: aMessage
       ^self mediator perform: #aMessage:fromObject: withArguments:
(Array with: aMessage with: self name)
```

The effect is to send the message and the component's identity to the system mediator. Therefore, to interact with another system component, a component only has to send a message to itself, the message will then be sent to the mediator by the 'doesNotUnderstand:' method inherited from the class systemComponent. Using the previous example the component SOP would execute the following statement to obtain the stock level of the part flange from the component INVENT.

```
self stocklevel: flange
```

8. CONCLUSIONS

The work described in this paper supports the notion that the software crisis, primarily caused by creating systems of extreme complexity as software monoliths, will only be curtailed when the component system paradigm moves out of conceptual design and can be clearly recognised in system implementations. The work has demonstrated that loose coupling of components is an important requirement for component systems. This requirement is as yet unsupported by general purpose distributed object technology products and is the responsibility of the system creator to build in at the design stage and carry through to implementation.

If the system creator abdicates his responsibility and does not provide the additional infrastructural software elements required to support loose coupling it is quite possible, if not highly likely that his distributed object system will be just as monolithic as the legacy systems which exist today.

REFERENCES

Jackson M., (1995) "Software Requirements and Specification", Addison Wesley.

Taylor D., (1992) "Object Oriented Information Systems: planning and Implementation", New York, John Wiley & Sons.

Ganti N and Brayman W, (1995) "The Transition of Legacy Systems to a Distributed Architecture", New York, John Wiley & Sons.

Chidamber, Shyam R,and CF Kemerer., (1994) “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, Vol. 20,No. 6.

Hitz, Martin, and B Montazeri, (1996) “Measuring Coupling in Object Oriented Systems”, Bject Currents, Vol. 1, No. 4.

Gamma E, Helm R, Johnson R and Vlissides J, (1994) “Design Patterns”, Addison-Wesley.

Leffler, S., McKusick, M., Karels, M. and Quartermain J., (1989) The design and Implementation of the 4.3 BSD UNIX Operating System. Reading MA: Addison-Wesley. 1989.

OMG (1994) "The Common Object Request Broker: Architecture and Specification", Object Management Group Inc., 492 Old Connecticut Path, Framingham, MA., USA.

Edwards J,Clements P, Gascoigne J, and Coutts I, (1997) “Component Based Systems: the basis of future manufacturing Systems”, Component Users Conference CUC96, Munich July 96, SIGS Books.

Cox BJ,(1987) “Object Oriented Programming, An Evolutionary Approach”, Addison Wesley.

Booch G., (1994) “Object Oriented Analysis and Design”, Bejamin/Cummings Co. Inc.

BIOGRAPHY

Ian Coutts spent two years at Marconi Research as a research scientist, working on industrial assembly automation and robotics projects. He has spent the last eleven years at Loughborough University, and currently works in the MSI Research Institute at Loughborough. Particular responsibilities include work on infrastructure and facilities for enabling model execution.

John Edwards gained his PhD from Loughborough University in 1994. Having spent 13 years in industry, being involved in the creation of computer control an information systems, he joined Loughborough University in 1987. During his 11 years at Loughborough he has been involved with Systems Integration and is now a member of the MSI Research Institute where his role is as principal investigator on a number of UK government funded research Initiatives.