TECHNICAL REFERENCE ARCHITECTURE
FOR
COMPONENT BASED DEVELOPMENT
AND
ENTERPRISE APPLICATION INTEGRATION

By Tim Barrett

COMCOR
ComCor I.T Solutions b.v.

Contents

1. Introduction 4
1.1 Technical Reference Architecture Scope 4
1.2 Component Definition 4
1.3 Component-Based Solutions 7
1.4 Technical Reference Architecture Users 7
1.5 ICT And Business Problems addressed 7

2. Technical Reference Architecture Focus 9
2.1 Technical Reference Architecture Focus 9

3. Technical Reference Architecture Positioning 11
3.1 Business Strategy 11
3.2 Logical Application Architecture 11
3.3 Technical Infrastructure 11
3.4 Technical Application Architecture 12

4. Technical Reference Architecture Overview 13
4.1 Business Component Layering 13
4.2 Business Components and State 14
4.3 Integration of Function Components with Existing Applications 16
4.4 Business Components and Workflow 17
4.5 Component Invocation Models and Middleware 18

5. Technical Reference Architecture Meta Models 24
5.1 Positioning 24
5.2 Target Implementation Technologies 24
5.3 Business Task Components 25
5.4 Function Components 29
5.5 The Complete Technical Reference Architecture & Security 33

6. Supporting Infrastructure 35
6.1 Component Technologies 35
6.2 Message Brokers 36
6.3 Application Wrappers 40

7. In Conclusion 41

8. Glossary of Terms 42

Figures

Figure 1: Component Typology 5
Figure 2: Middleware Abstraction Through Layering 6
Figure 3: Middleware Abstraction through Runtime Software Components and Containers 6
Figure 4: Architecture 11
Figure 5: Technical Reference Architecture Business Component Layering 13
Figure 6: Technical Reference Architecture and Workflow Management 18
Figure 7: MOM Based Service Bus 19
Figure 8:Total View of Middleware 23
Figure 9: Business Task Component Meta Model 24
Figure 10: Function Component Meta Model 29
Figure 11: Function Component Granularity 32
Figure 12: Business Component Reference Architecture 33
Figure 13: Non-invasive Message Brokers 37
Figure 14: Super API Based Message Brokers 37

Author:

Tim Barrett

ComCor I.T Solutions b.v.

tim.barrett@comcor.nl

T: +31 (0)24 6453345

F: +31 (0)24 6453347

1. Introduction

This paper presents ComCor's **Technical Reference Architecture**. This provides an architectural reference for the **construction** and **integration** of **Component-based Solutions.** It provides answers to crucial questions in the following areas:

- The need to provide **Business solutions** which are both scalable and flexible
- The need to integrate new and already existing business solutions in a consistent, efficient, productive manner
- The need to define optimal physical and logical layering strategies

1.1 Technical Reference Architecture Scope

The reference architecture should be used widely within an enterprise for optimum effectiveness. It should be applied throughout an enterprise, or throughout a division or sub-division. If the architecture is used widely, then there will be a consistency of approach to the design, construction and integration of business solutions. Concrete projects use this architecture as a **reference**. It provides a consistent foundation for these projects. Building on this foundation, it guides the projects at a tactical level. Reference alone is not enough, the chosen architecture must articulate the directions of technical change to be followed. This means that the technical reference architecture also has a strategic role within enterprises.

The Technical Reference Architecture does not provide detailed coding guidelines, or recommendations aimed at specific tools. It provides a starting point and reference for the creation of concrete, detailed designs. From these designs, component-based solutions are constructed, and integrated with existing solutions.

1.2 Component Definition

The term component is used somewhat loosely within the software industry. The term can be applied at three distinct levels. We consider it important to identify these levels, and apply a specific component definition to each level. These definitions are then used consistently throughout the rest of the document.

1.2.1 Component Types

- **Objects** are programming language constructions. Objects and components are related because the code in a COM or CORBA or Java component is usually a set of VB, C++ or Java OO objects.

- **Runtime software components** are packages of programs managed as a unit and accessed only through interfaces. Runtime software components can be distributed – in other words they run in a middleware infrastructure. Com+, Enterprise Java Beans (EJB), and CORC (Corba Component model) provide standard (and competing) component models for runtime software components.

- **Business components** are the logical design modules in a service-oriented architecture. These components may be developed on any platform using any technique. They provide services through an interface. Business components are not necessarily built using runtime software components, although this will tend to be the case in modern software development projects. There are two types of Business Component: **Business Task Components**, which

interact directly with users in order to complete a set of logically related business tasks, and **Function Components**, which provide transactional services related to a logical domain. These services are often closely concerned with the manipulation of data, and a Function component usually encapsulates persistent data. Component Based Solutions are therefore made up of one or more Business Task components that orchestrate the use of Function components. Function components serve many masters. Many different types of Business Task component can use the same type of Function component.

All three of these levels have one important feature in common – they all make services available via **interfaces**. This document is largely concerned with Business components, and the use of runtime software components to build business components. We are also concerned with the need to allow existing (legacy) applications to project themselves as function components through application wrapping techniques.

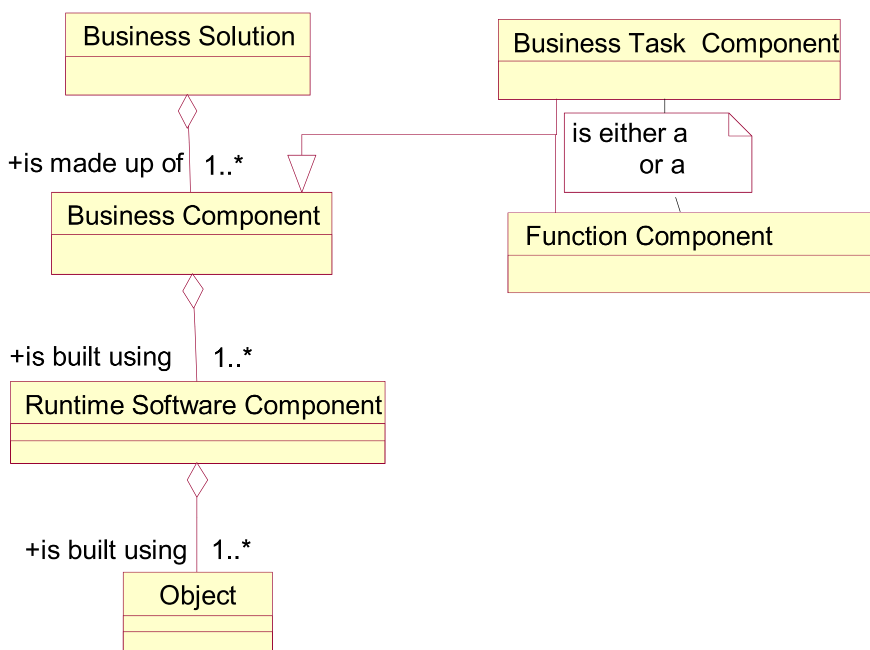

Figure 1: Component Typology

1.2.2 Runtime Software Component Containers

A runtime software component has to interact with its infrastructure (middleware) in order to solve the problem of its clients. This is a problem, which hinders reuse and can make the component unimplementable on other platforms. This problem is especially acute if the business logic which the component provides interacts directly with middleware.

To alleviate this problem, the component should adhere to the container model. The container model provides a clear separation between the container, which interacts with middleware, and the business solution provided by the component. A runtime software component developer provides the business logic. The vendor of the component technology provides the interaction of a component with middleware. The container model allows technical issues, such as transactional

behavior and security, to become attributes of the container. These attributes can then be configured dynamically, as opposed to arranging them 'somewhere' in the business logic. The component developer can concentrate fully on providing the business solution. This is a shift from the time-honored technique of building abstraction layers above a technical platform and middleware (see figure 2).

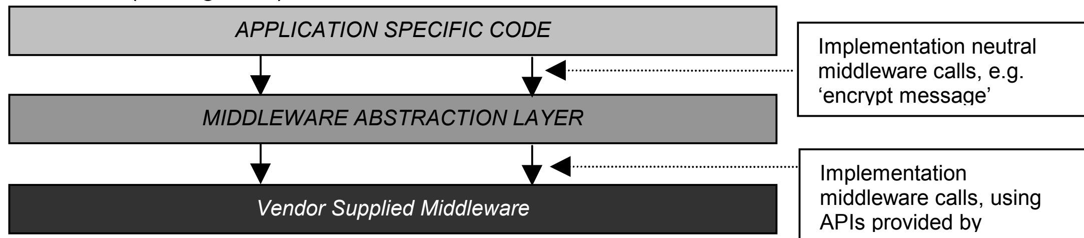

Figure 2: Middleware Abstraction Through Layering

Usually, the motivation for such layers is the perceived need to express middleware services, such as security services in terms of a vendor independent set of services. Application logic calls the abstract services as opposed to calling the middleware directly. The potential advantage is the perceived independence of the business logic from the specific vendor's middleware. This approach, however, can give rise to a serious maintenance burden. The abstraction layer needs to be maintained, supported, and kept up to date with the services supplied by product vendors. Often, extra services are provided above and beyond those supplied by the products upon which the layer is built. This exacerbates the maintenance burden. The abstraction layer is thus proprietary and does not solve a problem but instead can create a new maintenance problem.

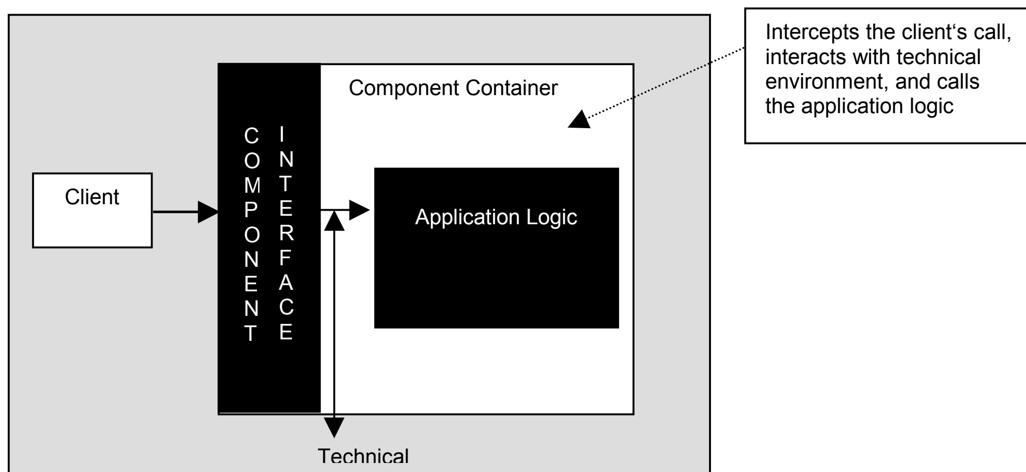

Figure 3: Middleware Abstraction through Runtime Software Components and Containers

Figure 3 depicts the component/container approach. The runtime software component's container intercepts client calls and manages behavior such as the transaction mode, locking regime, and security. This behavior is dynamic and can be configured at runtime. The programming model has altered quite radically. Instead of the abstraction layer calling the vendor specific middleware, the container *calls* the application code inside the component.

1.2.3 Runtime Software Component Technologies

At the time of writing two runtime software component technologies are prevalent. These are the Component Object Model (**COM+)** from Microsoft, and the Enterprise Java Beans (**EJB**) model from Sun et al. In addition to these two, the OMG is defining '**CORC**', the CORBA component model. We expect to see the first vendor implementations of the CORC model in early 2000.

1.2.4 Business Components and Runtime Software Components

Business Components can be built in such a way that the spirit of component based development – which is all about having applications provide interfaces – is lived up to, without necessarily using one of the industry standard runtime software component technologies to build the Business components. If a business application provides a clean set of callable services, which can be called by any client or peer application, then this application could be said to be a 'Business component', especially if these callable services are the only way in which the application's services can be accessed. This interface centric definition of the term Business Component is important from the point of view of integration with existing applications. It also allows us to design and develop in terms of components where large-scale mission critical applications are being constructed, for which the current runtime component technology products and standards such as COM, EJB and CORC do not yet provide a sufficiently robust and complete platform or sufficient performance.

1.3 Component-Based Solutions

Component-based solutions are characterized by:

- A purely interface focused approach to the design and development of components at all three levels (program objects, runtime software components and business components).
- The provision of business solutions through the creation, extension, deployment and assembly of **business components**.
- The ability to distribute business components in a heterogeneous environment.
- The use of various styles of communication between these components, including:
  - Synchronous method invocation
  - Deferred synchronous method invocation
  - Asynchronous communication (one way messages, and publish / subscribe messaging)
- The provision of access to business components through various 'channels' (e.g. internet, intranet)

1.4 Technical Reference Architecture Users

The Technical Reference Architecture has been written for those involved in the design and construction of new component-based solutions, as well as those involved in the integration of heterogeneous solutions of all types - new, purchased and already existing (legacy).

1.5 ICT And Business Problems addressed

The Technical Reference Architecture addresses the following strategic problem areas:

- The need for a standard, consistent approach to the **construction of component based solutions**
- The need for a standard, effective approach to **enterprise application integration**
- The need to **select a runtime software component technology**, and to correctly position runtime software component technology.
- The provision of technical architectural criteria to be applied to **package acquisition**
- The provision of a **migration strategy** providing a framework for the gradual introduction and integration of component based solutions, whether these are built in house or purchased

- Achieving **reuse and flexibility** in IT solutions

1.5.1 Potential Benefits

The potential benefits of a consistent approach to the creation and integration of component- based solutions are:

- Flexibility:
  - Components can be added without breaking the system
  - Component behavior can be changed without breaking the system
  - Component behavior can be changed quickly
- Presentation channel / Input device independence
- Backend ('BackOffice') application and database independence
- Productivity
- Business Solution Assembly - new business solutions are assembled from existing business components and new business components
- Invocational Reuse. Reuse in this model is not simply a matter of reusing code objects. It focuses on the provision of generally useful services which can be invoked by a variety of different business solutions in a variety of contexts.
- Existing applications can be rendered as reusable business components through the integration architecture
- Reuse of intellectual content of the architecture - no need to reacquire knowledge
- Reuse of technical standards to be applied for package acquisition
- Highly productive languages can be applied - e.g. Java, Delphi as well as proven languages such as COBOL.
- Developers focus on providing solutions to business problems, as opposed to having to deal extensively with low-level middleware problems (especially if modern, runtime software component technologies are used).
- Architectural approach to technology ensures that technology is a prime enabler of business innovation

2. Technical Reference Architecture Focus

We begin by describing the structure of a business solution as defined within the Technical Reference Architecture.

The reference architecture exhibits *service centricity,* that is the focus is on *distributed services*, with 'thin' clients and a positioning of databases *not* simplistically as the third tier in a three tier architecture, but as a method of resource management within a shared resource layer. Other resource management approaches such as message queuing and pathways to legacy applications are also recognized within this layer. Another major feature of the architecture is the attention paid to the automation of business tasks**.** We suggest that automated support of business tasks must be *highly dynamic* and *event driven*, because real life business tasks exhibit these attributes. Another focus area is that of application integration. Most enterprises today suffer from *application spaghetti* - that is a large amount of difficult to manage ad hoc interfaces between heterogeneous applications on heterogeneous platforms. We describe an application wrapping architecture, as well as vital tools (application wrappers, message brokers and supporting middleware) which if used well can significantly reduce this interface burden.

We then turn to the *Technical Infrastructure* services vital to any Technical Reference Architecture based solution. We focus on **component technologies**, **message brokers**, **application-wrappers** and **communications middleware.**

2.1 Technical Reference Architecture Focus

The Technical Reference Architecture focuses on the following key areas:

- The **Construction Architecture** to be used for the design of new component based solutions. This architecture focuses on the logical layering of a component based solution, providing guidelines which will help designers and developers to arrive at a clean separation of concerns, and to design components with the correct level of **granularity**.

- The **Integration Architecture** needed to provide a cohesive integration strategy and infrastructure for the cheap, fast and effective integration of new solutions, purchased package solutions and existing applications.

- The combination of these two brings about a **Migration Architecture**, which can be used to enable coexistence of existing applications and new component based solutions. If properly managed, migration from the older applications to the new solutions can be achieved at a pace which returns results without causing business disruption.

- The Technical Reference Architecture can be used as a source of criteria for **the evaluation of commercial packages**. This is particularly valuable when packages are being evaluated from the point of view of integration with other packages and existing applications. Thearchitecture is very strict about layering, as each layer provides indirection relative to the layer that uses it. This facilitates integration. We would therefore expect to justify and articulate the need for such layering schemes, using the Technical Reference Architecture as a reference, when evaluating commercial software packages.

- The **Technical Infrastructure** needed to support the architecture. Runtime Software Component Technologies and Enterprise Application Integration middleware are the key areas addressed.

- Recognizing that few commercial projects undertaken today can afford to ‘start from scratch’ in terms of creating development tools and frameworks brings us to another important function of this Technical Reference Architecture - it can be used to **evaluate the architectures embodied within commercial development frameworks.**

The architectural approaches being proposed within this document provide ground-rules for practitioners which support the creation and integration of solutions that are future proof, provide real flexibility, and possess enterprise strength.

3. Technical Reference Architecture Positioning

It is probably worth examining the broad term *architecture*, in order to clarify the architectural area that is addressed by the Technical Reference Architecture. Several Architectural layers exist that address distinct concerns; we need to clearly define what these layers represent.

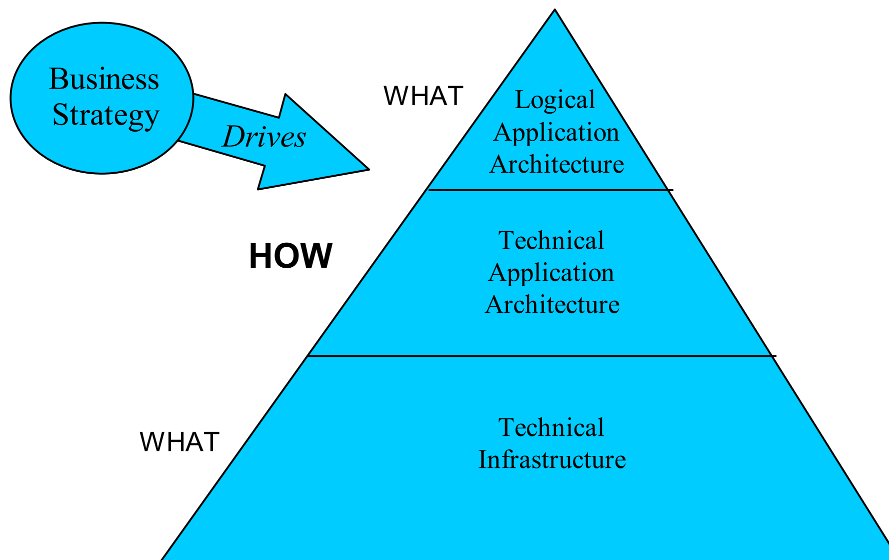

Figure 4: Architecture

3.1 Business Strategy

All enterprise activities, including the activity of application development, are conducted in support of an overriding **business strategy** which provides strategic direction. Examples of such strategic direction are imperatives to **design and deploy products rapidly and flexibly**, strategies aimed at achieving **customer intimacy**, strategies to introduce **Just In Time**, or **Total Quality Control** manufacturing practices etc.

3.2 Logical Application Architecture

The Logical Application Architecture essentially describes **what** a business solution does in terms of business data and process. Insurance policies and products, Manufacturing and Logistics products, components, their relationships to one another, functional roles and the processes that use these entities are examples of the artifacts to be identified and defined at this level. The functional definition of a Contract Management solution in terms of tasks and logical entities, and functionally related groupings of entities would be the subject of Logical Application Architecture.

3.3 Technical Infrastructure

The Technical Infrastructure describes ***what*** physical hardware and software components are used to support business solutions. These components include computer hardware, operating systems, database management systems, networks, transaction management systems, message queuing software, object request brokers, etc.

A technical infrastructure is put in place in order to support requirements such as flexibility, scalability, portability, interoperability, integrity, reliability, high performance and security which emanate from the logical and technical application architectures. It *supports* these architectures.

3.4 Technical Application Architecture

The upper (logical application architecture), and lower (technical infrastructure) layers are primarily concerned with '*whats*' - *what* business service, *what* business data, *what* physical infrastructure. The middle layer - the technical application architecture layer **-** is concerned with *hows***,** from the technical perspectives of construction and integration.

- *How* business solutions are structured in order to allow business solutions to take rapid advantage of advances in technology, and in order to enable the creation of flexible, scaleable business solutions.
- *How* business solutions interact with the Technical Infrastructure, without becoming directly entangled in the Technical Infrastructure. This requires a structure that insulates as much as possible the business content of a business solution from its interaction with middleware, database management systems, transaction processors etc.
- *How* business solutions are structured in such a way that integration with existing applications can be achieved through legacy wrapping techniques, which provide encapsulation of the details of the legacy environment, making both the using components and the legacy applications *reusable*.

The Technical Reference Architecture provides a reference for architects involved in the definition of Technical Application Architectures. It does not, in itself provide a complete Technical Application Architecture, as these need to be developed in 'concrete' – i.e. within one or more existing domains, using specific technologies, on specific platforms. The Technical Reference Architecture is intended to provide a substantial basis for the development of concrete Technical Application Architectures.

4. Technical Reference Architecture Overview

This section provides an overview of the technical application reference architecture at the Business Component level.

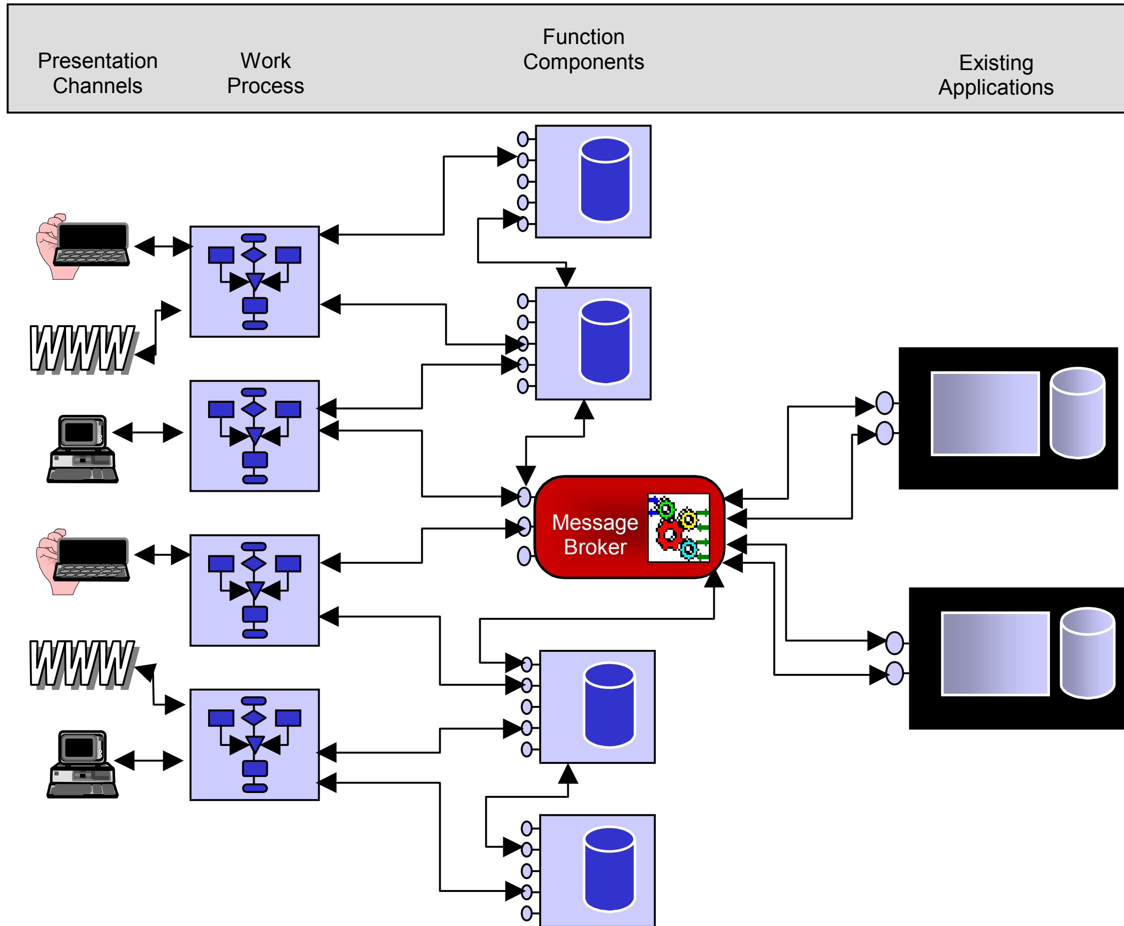

Figure 5: Technical Reference Architecture Business Component Layering

4.1 Business Component Layering

The architecture breaks service components down into 2 different **layers.** Each of these layers contains components of a different type: **Business Task Components** and **Function Components.**

4.1.1 Business Task Component Layer

Business Task Components interact directly with users in order to complete a set of logically related business tasks. A Business Task Component contains the business process logic needed to complete these business tasks. This will usually involve a high degree of interaction with the user. Presentation detail is not tangled up with business process logic. This allows Business Task Components to be server based, as well as allowing different display technologies to make use of the same Business Task Component. An example of a business task supported by a Business

Task Component is '*Add new relation to customer relations management system'*. A business task may have a duration of minutes, hours, or even days. In order to allow a user to stop and restart a task, or to allow another user to take over the same task, it is important that the state of a task instance can be saved to some form of persistent storage, and restored. Business Task Components do not contain the business services that they need to perform their tasks. The Business Task Component responsible for '*add new relation to customer relations management system*' invokes services which are provided by the Function Component layer, such as '*validateCustomer(customerName)*', and '*getPostcode(City, Street)*' – it does not implement or contain these services itself. It is also unlikely that it will contain a database, as the only state that it will maintain is conversational state (user preferences etc.), and state that it has obtained from Function Component calls, which it might cache during the execution of a task.

4.1.2 Function Component Layer

This layer provides services that will be used by many different Business Task Components. In addition to this, Function Component Services may invoke other Function Component Services. The services provided are therefore more *widely* useful and generic in nature than the services provided to users by Business Task Components. Business Task Components are dedicated to a specific business task or set of tasks, whereas Function Component Services are dedicated to many different types of task, as well as being invoked by other Function Component Services. A useful analogy here is the idea that a Business Task Component represents a meal cooked to the diner's taste. Function Component Services represent the ingredients that are used in the meal. These same ingredients could be used in many very different recipes. Function Component Services are inherently more stable (in terms of the amount of change that occurs to them) than Business Task Components. They can be used for many different Business Task purposes.

Logically related services are grouped together into Function Components**.** These components should provide services related to recognisable business or technical domains such as '*Customer Relations Management*', '*Contract Management*', '*Account Management*' and '*Naming'*. They provide services related to these business domains - such as '*getCustomerDetails*', '*extendContract*', '*transferFunds'* and '*locateService*'.

4.2 Business Components and State

Business Components can be either *conversational* or *non-conversational.* In general, Business Task Components are conversational, and Function Components are non-conversational. These terms are closely related to the management of *state*.

State can be broken down into three distinct types:

1. Conversational State
2. Function Component Service Execution State
3. Persistent State

- **Conversational State** involves the notion of a session. A client obtains a reference to an instance of a Business Task component. The client now 'engages in a conversation' with this component instance. This simply means that the client executes one or more services on the *same* component instance. As the 'conversation' is taking place, 'properties' (variables) within the component instance may be being set and reset as a result of the services being executed. No other client can obtain access to the same component instance. The state that is built up within this component exists for the lifetime of the conversation with the client. The term 'stateful' is often used in fact to mean that a component is conversational.

- **Service Execution State** refers to variables and properties that are set by a Function Component service whilst it is executing. This state is transient; that is it does not last beyond the invocation of a single service. Clearly, all but the most trivial types of service require this kind of state.

- **Persistent State** refers to data which is typically held in a database management system. This is data that can exist permanently, and is usually mutated transactionally. Most meaningful Function Components will own a persistent store of data.

The services provided by Function components should be non-conversational, because:

- It is difficult to maintain consistency of state in instances of these services with persistent state in the databases that they own.
- A TP monitor environment can 'pool' stateless (non-conversational) Function instances, sharing a limited number of instances or instances of components providing these services amongst significantly larger numbers of clients. This contributes significantly to scalability and performance. Products such as Microsoft Transaction Server, Tuxedo, CICS, IMS and most EJB servers provide this vital instance swapping and pooling capability for stateless Function components[1].

Many Function Component Services will involve the manipulation of persistent state. The details of the databases being manipulated must be entirely hidden by the Service however. This means that a Business Task component requesting the Function Component Service 'saveContract' should not be expressing this request in a manner which displays knowledge of the structure of the database that the Contract is held in. This would be the case if for example SQL were passed between the Business Task Component and the Function Component Service.

4.2.1 Business Resource Services

Function components often provide business services that often involve the manipulation of data. The precise location of this data can vary however. Customer name and address data for example may be held in an external legacy application, whereas opportunity related data might be held in a relational database dedicated to customer information. If the services provided by a customer information Function component (for example) are aware of these specific locations, then the internals of these services will:

- need to be changed if the data locations change
- will always need to be changed if the component is used in another environment
- will be polluted with code related to communicating with the legacy application, which may be far more complex than the logic of the Function Component service itself

In order to avoid this tight coupling between the Function component services and the resources that are being manipulated, the Business Resource Component has been devised. A Business Resource Component is an optional runtime software component class within a Function Component. Business Resource services provide a 'CRUD' (Create/Read/Update/Delete) interface for a given business entity – for example **Customer** or **Contract**. The implementations of these services *do* have knowledge of the specific locations of the business entities that they represent. This could involve straightforward access of a relational database. It could also involve access of a legacy application via wrappers. Function component services can now express data (shared business resource) access in terms of the interfaces projected by the Business Resource services.

[1] Note that the established TP monitor products mentioned (CICS, IMS, TUXEDO) do not explicitly run 'components'. They run 'programs' or 'transactions', which are stateless.

This means that the business logic is independent of the physical type and structure of data, and is therefore highly versatile and potentially reusable in a variety of environments.

Business Resource Services are *not* made publicly available. A Function Component Service Component *owns* one or more private Business Resource Service classes. This means that a given Function component can only make direct use of the business resource services that it owns. If it needs to access data owned by another Function Component, then it must access this data via the interface that the other Function Component makes publicly available. This ensures that Function components *own and encapsulate* data access, and changes made to data storage type, or access method, can be made without affecting other Function components.

Enterprise Java Beans Entity Beans are a good example of Shared Business Resource service providing components.

4.3 Integration of Function Components with Existing Applications

If a business service is already provided by an existing 'legacy' application, then the legacy application should be wrapped in such a way that its services can be made available as if these services were being provided by one or more Function Components. This is achieved through **Application Wrapping**. Wrapping applications boils down to providing the required business services, which are mapped onto one or more legacy applications. This can become complex if the service needs to access more than one legacy application in order to fulfil its task, or multiple interactions with the same legacy application are required within the context of a single service execution. Simplistic wrapping assumes that the applications being wrapped have straightforward APIs that simply need to be called. In the real world, many legacy applications do not have such APIs, as their code is monolithic, with single programs addressing presentation, business process control and data manipulation. Often, for such applications, the only way of accessing them from 'outside' is to resort to direct database access or screen scraping. Clearly *Function*s should not be presenting screen-scraping services, therefore **wrappers** are needed which translate the Function request into the necessary screen interactions via a screen scraper. There is a point at which the cost involved in hand-coding these transformations becomes prohibitively expensive. If this point is reached, then the use of a **message broker** should be considered. It is important to point out that the message broker itself would project its services as Function Component services, using whatever middleware/component model is being used by the 'conventional' Business Components.

4.3.1 Transaction Management and Integration with Existing Applications

Business Components can be assembled together to provide a Business Solution. If these Business Components have been built using technology which supports distributed transactions (and the TP monitor style products which have been mentioned so far do support distributed transactions), then Business Task components can request Function Component Services which may well be executed as a fully fledged, 2 phase commit style of 'ACID" transaction. This means that the service either executes successfully, or it will be aborted in such a way that all updates made to resource managers (databases. message queues etc.) are completely backed out. The resources are always left in a consistent state.

The concept of transaction co-ordination across multiple resource managers is vitally important to the subject of integration with existing applications. In the vast majority of situations involving integration with legacy applications this level of ACID transaction management will not be possible. This is because these applications either do not own resource managers which are capable of participating in a 2pc transaction as described above, or even if they did, they do not provide a layer of callable services which can be included in the overall transaction. Directly aaccessing the

database of a legacy application, even if that database is a fully fledged XA compliant resource manager is a very dangerous undertaking, as this will often bypass the business integrity rules that are implicitly built into the application's code base. Involving the resources of external applications in transactions which are originated by another application or component is usually only possible when both applications are based on a component based transactional architecture such as that being proposed in this paper. This general inability to access existing applications in a fully transactional sense is a simple fact of real life that must be recognized.

4.4 Business Components and Workflow

Business Task Components are *not* workflow components. A Business Task represents an automated single task, such as '**createNewCustomer**'. A task of this nature may involve several screens, but it is generally completed in one sitting, and does not cause other tasks to be initiated. The orchestration of tasks is essentially the domain of **workflow management**. Workflow management is also referred to as Business Process Control.

A Business Process, or Workflow differs from a Business Task in that:
- It may have a duration of days, weeks or even months
- Several tasks may need to be completed in order before the workflow as a whole is completed
- These tasks may be executed in parallel
- Some of these tasks may be purely manual, with no automated content

The Business Task createNewCustomer could be used in workflows such as createMortgage, or handleClaim.

Clearly the management of a workflow (or business process) involves a significant degree of **event management.** Events such as the completion of a particular business task need to be published, in order for the workflow system to determine what task(s) now need to be initiated in order for the workflow to progress.

From the point of view of the Technical Reference Architecture, Workflow (or Business Process Control), is a higher level layer which will interface with Business Task Components. This interfacing will largely take the form of:

- Requesting users to complete specific business tasks
- Receiving published events from Business Task components indicating that a particular task has been completed, together with any relevant status information.

Although workflow management is not an area which is addressed by this architecture, figure 5 positions it with respect to the Technical Reference Architecture. The key point being made here is that of **strict layering**. Workflow management does not interact directly with Function components – it interacts solely with Business Task components.

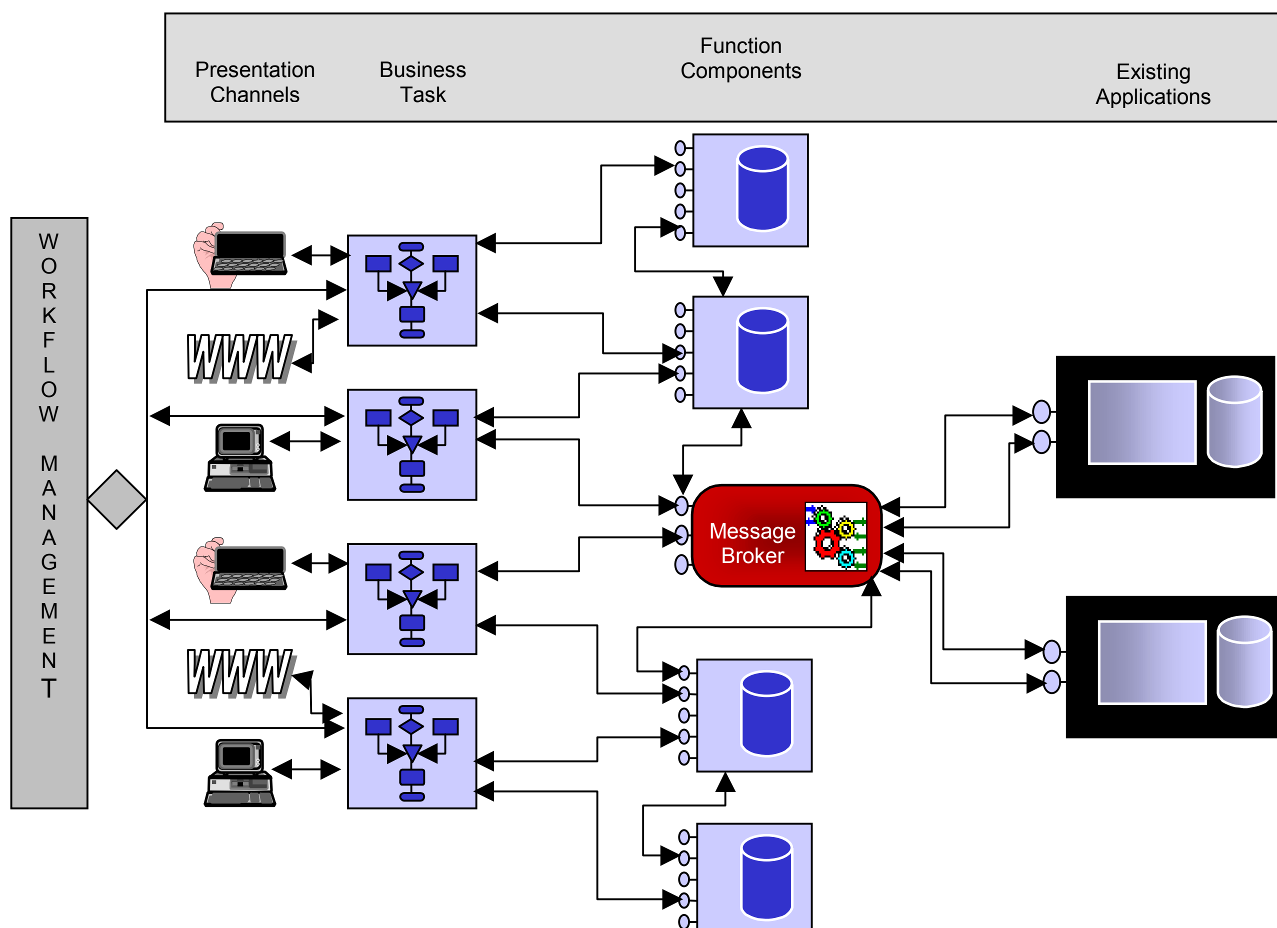

Figure 6: Technical Reference Architecture and Workflow Management

4.5 Component Invocation Models and Middleware

Figure 5, above, depicts an environment in which a great deal of inter-component communication takes place. This communication takes place at all of the component levels (objects, runtime software components and Business components) that have been identified. Confusion can arise when this architectural vision is mapped on to middleware which enables component communication, because there are several different *styles* of middleware available. These styles are largely related to what we term **component invocation models**.

4.5.1 Middleware Categories

The significant middleware categories, together with the component invocation model that they most naturally support are:

- **Message Oriented Middleware** (MOM), which is designed to support an asynchronous, point to point queued invocation model.
- **Synchronous Middleware**, which supports a synchronous invocation model. This category can be further sub-divided into:
  - Procedural Remote Procedure Call (RPC) synchronous middleware.
  - Distributed Component middleware which provides an object oriented invocation model

  Another vitally important characteristic of this style of middleware is its ability to support **distributed transactions**.
- **Publish and Subscribe Middleware**, which provides an asynchronous event driven communication style. Strictly speaking, there is no question of an *invocation model* here, as direct communication between publishers and subscribers never takes place. In this model, Business Task components would publish the completion of business tasks (or steps within these tasks) to an event manager. Other components (typically in the workflow level) would then receive notification of these task completions, because they have registered an interest in these events with the event manager. Another significant use for a Publish and Subscribe mechanism at a lower level would be in support of data synchronization between Function components. This is discussed in more detail in section 5.4.2.3.

It is essential that all of these middleware categories be recognized as being valuable, needed, and complementary. Unfortunately, they are often viewed as competing technologies, with proponents of one style suggesting that that the category in question is a panacea which can be used across the board.

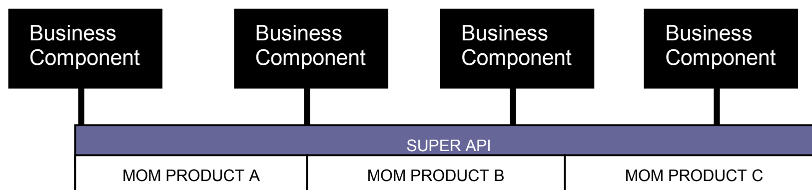

Figure 7: MOM Based Service Bus

A commonly held view is that a SERVICE BUS should be used, which abstracts middleware via a SUPER API, thus providing a level of insulation from concrete middleware products. This is depicted in figure 7[2]. More often than not, these Super APIs encapsulate Message Oriented Middleware. The logical conclusion of this would be that all communication between Business Components is restricted to the asynchronous, point to point queuing model. This is clearly unsatisfactory, because this style provides inadequate support for synchronous communication, cannot support distributed transactions because it is inherently time independent, and provides inadequate support for the Publish and Subscribe model. Clearly, the scope of the SUPER API could be widened to include full support for synchronous transactional communication as well as publish and subscribe capabilities, but this would involve a significant engineering effort. It would be particularly difficult to 'abstract' the standard component models, given the fact that they project sophisticated object and interface oriented programming models, make use of containers, and provide protocols for the transmission of transaction contexts, security etc. Also, the industry is already converging on a small number of standard component models, so abstracting these would

2 Note the subtlety in this picture – the second business component, residing at the boundary between MOM product A and MOM product B is providing the middleware switching capability typically found in message brokers.

result in a proprietary layer being added to a standard model! On the other hand, in the asynchronous messaging area there are several competing products, all of which provide much the same functionality, each of which comes with its own proprietary API. Furthermore, these APIs are often low level, and unnecessarily complex. This would suggest that abstracting these APIs, and providing support for several different MOM products under one uniform layer may well be worthwhile.

This leads us to the conclusion that it makes sense to adopt a 'horses for courses' approach, which would feature use of all of the middleware categories mentioned. The challenge here lies in recognizing which horses are suited to which courses. The following positioning is suggested.

4.5.2 Positioning of Middleware Categories and Component Invocation Models

We suggest that there are two major component invocation styles – in fact these styles relate directly to the type of service being provided. We refer to these as **Solution Level Services** and **Enterprise Level Services**.

4.5.2.1 Solution Level Services

Solution level Services have the following characteristics:

- *They are almost always invoked synchronously* (put another way – they are generally not invoked via time independent messaging).
- *They may be invoked using an object/CBD oriented programming model*. That is object references may be returned which have methods and properties. Methods are invoked on these references and these methods may themselves have object references as inputs and outputs. Also, transitive invocations may occur which involves invocation of methods of objects that are themselves properties of a root object. Finally, this model would introduce **polymorphism**, where a given object reference could be invoked 'as if' it were an instance of a more general class of object. Clearly, having stated that object references contain properties, we can also conclude that:
  - *These services may be invoked conversationally, and they maintain conversational state*.
  - *They often provide 'low level' services* - such as 'getAddressLine2'. (However, they will also provide services which as 'getCustomer', which might return a reference to an instance of a Customer class).
- *They can be called within a transaction context*. Any resource managers manipulated by this type of service will become part of a transaction, which is ended with a commit or rollback, when the root service (the service which initiated the transaction context) concludes. The non-initiating, called service has a 'veto right' which allows it to signal that the transaction in which it is participating should be aborted, if an error condition arises.
- *In general terms, the callers of these services are 'nearby'*. A distributed invocation mechanism (such as CORBA) may be used, but it can generally be expected that the caller and the service provider are physically in close proximity to one another. They will generally be found on the same LAN, and of course they may be found *within* the same process.
- *They use the same synchronous middleware*. In other words, these components invoke one another using a single middleware such as DCOM, CORBA etc.

We refer to these services as Solution Level services because they will tend to operate closely together within the context of a particular business solution – or set of business solutions. They make use of a common middleware (in other words a common runtime software component model).

4.5.2.2 Enterprise Level Services

What then could be the defining characteristics of Enterprise Level services?

- *They often involve 'pure' messaging.* This is where A sends a message to B and does not require any form of response, directly or indirectly. B can pick up the message from a queue.
- *They are usually invoked synchronously, but this may involve a high level of deferred synchronous invocation*. This requires explanation! Interestingly, this is arguably the optimum use of messaging. However, deferred synchronous invocation requires an underlying transport that is capable of sending messages that can be queued. Essentially, deferred synchronous invocation means that A sends a *service invocation message* to B. A does not wait for a response to B (although it is free to do so if necessary), instead it continues with other processing. Note that this processing may involve sending yet more service invocation requests to B or to other service suppliers. At a point to be determined by A, responses will be read. These responses could have arrived in any order, A's own application logic is responsible for matching responses to requests (when this is needed). A powerful variation on this theme is an even more loosely coupled scheme, where A sends messages to B and does not await the response. A is also a service, which receives requests. Some of the requests which A receives may in fact be messages from B, which were sent in response to the messages A sent to B in the first place. It is even possible that A receives requests from C, where C has been triggered by B which was originally triggered by A. Rather than try to make sense of the scenarios described here, suffice to say that the **service invocation mechanism is message based**. This type of invocation mechanism clearly does not lend itself well to invocation of 'setAddressLine2', and by the way it **cannot**, because of its inherent **time independence**, support an XA style of transaction, which is absolutely time dependent. This invocation mechanism *is* well suited to work flow style applications, where task requests are queued, may trigger other tasks, and acknowledgement of task completions may cause new tasks to be initiated within the requestor. Also, tasks may be long running which means that considerable time may elapse between service invocation and service response.
- *They are not invoked using an object/CBD oriented programming model*. These services are 'flat' – invocation is straightforward: Invoke namedService(inputmessage). The response is a message in its own right (see above). This has very little in common with the object/component oriented invocation model described earlier. Again, this leads us to the conclusion that object orientation and CBD technologies such as COM belong *within* a business solution, or closely related set of solutions. Enterprise level service invocations should be used for straightforward, high level services across business solutions, or sets of business solutions.
- *Enterprise Level services should always be non-conversational.*
- *They provide high level, business significant services* - such as 'processClaim', 'getExchangeRate', 'TransferFunds', or 'getCustomer'. Note that these examples exhibit some of the key characteristics that have been identified or implied so far:
  - They provide a straightforward, 'flat' service
  - Significant time may elapse between invocation and response (process claim is the prime example of this)
  - The 'get' services deal in significant business entities as opposed to attributes of these entities
  - They are non-conversational
  - They are not invoked within a transaction context
  - They could all sensibly be invoked in a deferred synchronous fashion
- *They cannot be called within a transaction context*. As has already been pointed out, the time independent service invocation model which is a cornerstone of this environment cannot possibly support distributed, XA style transactions. Of course some form of long lived business transaction compensation mechanism could be realized, which would require co-operation between application level software (supplying the identity and implementation of the

compensation services, and signaling the need to abort and compensate), and the middleware (recognizing the failure and invoking the compensation service).
- *Requesters and Providers of services are not necessarily in close proximity to one another*. In other words, Enterprise Level services could well be provided in one (or a small number) of places, and made available to callers emanating from a wide area (this may scale up to a service maintained in one country which is made available globally). Distribution to this extent also requires the sophisticated use of naming, directory and security services.
- *Enterprise Level services should make use of an enterprise wide naming & security scheme*. This is an important requirement. 'Invocational reuse' (which is what this is all about) will only be realized if the underlying naming and security infrastructure is consistent and effective.
- *They are made available in a highly heterogeneous environment*. Unlike the solution level services which reside in a homogenous environment (that is they all use CICS, TUXEDO, DCOM, CORBA or something similar), enterprise level services need to be made available in a heterogeneous environment. A heterogeneous environment features different styles of middleware, different middleware products and a substantial set of existing 'legacy' applications.

4.5.2.3 Workflow and Services

As we have already stated in section 4.4, workflow management is event driven. Workflow management is a higher level layer, which receives notification of business task completion, and triggers users to initiate business tasks in order to complete steps within the workflow. This type of interaction is clearly best served by a publish and subscribe style of communication.

Figure 8, below, provides a complete multidimensional middleware focused picture depicting the various service types and middleware communication styles in an enterprise environment. We refer to this as a **Total View of Middleware**.

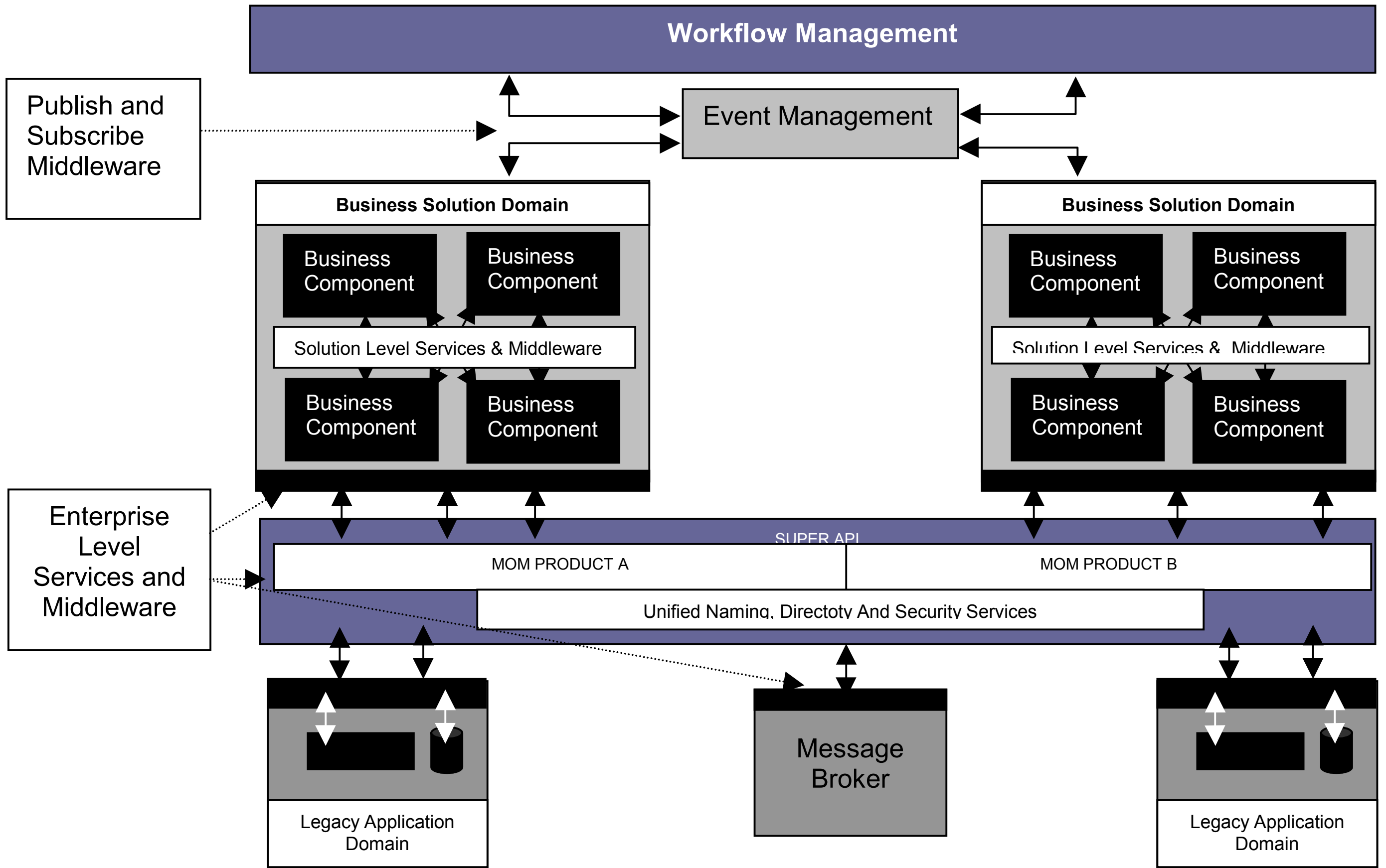

Figure 8:Total View of Middleware

5. Technical Reference Architecture Meta Models

This section provides meta-models, expressed in UML for:

1. Business Task Components
2. Function components

5.1 Positioning

These models should be used as a basis for a more detailed design, tied to a particular technology choice. A wide range of technologies is available to project teams when solutions are built. The technology chosen will require concrete decisions to be taken, which we cannot take at this reference level, as this would tie us to a particular technology. This is particularly relevant in the case of the Business Task meta model, where the decision to use web browser based presentation as opposed to GUI presentation, would result in significant differences of approach at the detailed construction level.

5.2 Target Implementation Technologies

Although some of the principles presented in the following section are applicable to a broad range of implementation technologies, this particular reference architecture focuses on the development of Business Software components that are built using **runtime software component technologies**. These, in turn, are built using object oriented development techniques, and accessed by thin clients. In other words, this architecture is aimed squarely at development in an environment characterized by the use of technologies such as:

- Web browser (and other thin client devices) front ends *generated* from a web server
- Java Servlets and Java Server Pages, Microsoft Active Server Pages
- Enterprise Java Beans(EJB) components /EJB Servers, Microsoft Transaction Server and COM+ components, CORBA component model (CORC) and CORBA based application servers.

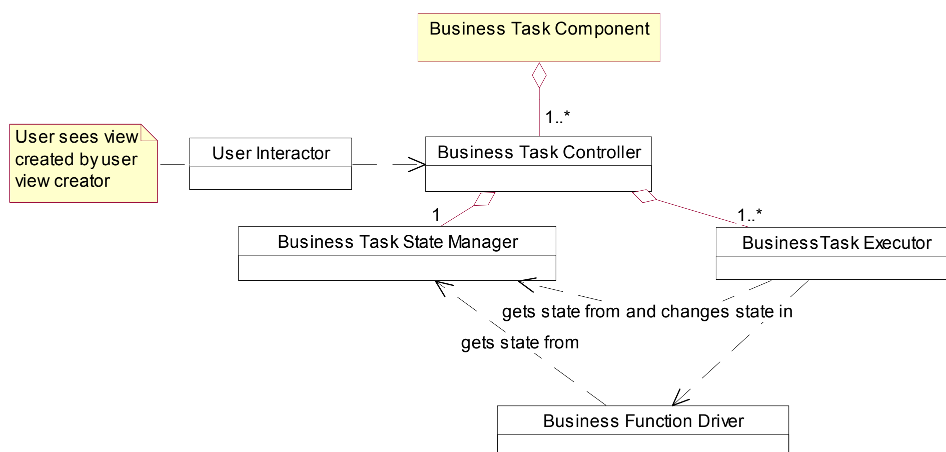

Figure 9: Business Task Component Meta Model

5.3 Business Task Components

Figure 9 depicts a Business Task Component together with its constituent runtime software component classes. These are described below.

5.3.1.1 User Interactor

This class communicates directly with the user. It is responsible for:

- Encapsulation of presentation detail. This means that it is aware of the type of presentation technology being used, and presents users with ‘views’, interacting directly with the appropriate presentation medium. It may use subordinate classes to achieve this.
- Initial creation of the Business Task Controller. Although the Business Task Controller is in charge of the management of the task, when a user first logs on, a Business Task Controller instance does not exist. The User Interactor does therefore have the *executive* responsibility of knowing which Business Task Controller to hand control to. This will almost certainly be linked to an initial menu choice made by a user.
- Communication between the user and the Business Task Controller. The Business Task Controller will request the presentation of views. The User Interactor presents those views, using the appropriate presentation technology. Users will react to these views by using controls contained upon them, which will cause events to be raised. Some of these events will be handled within the view itself (for example changing the contents of a textbox on a form). Other events will require ‘executive handling’. A *submit* button on a form is a clear example of this. An event requiring executive action will be passed to the User Interactor, together with any relevant data (often the whole form). The User Interactor will pass this data on to the Business Task Controller. In order to encapsulate fully the presentation technology being used, it is important that the User Interactor present the view data to the Business Task Controller in a neutral manner. The use of XML is strongly recommended.

In a sense, the User Interactor drives the business task, because it communicates directly with the user, and it calls the Business Task Controller. In fact it is fully subordinate to the Business Task Controller, because it is that class which decides what action should be taken next in order to complete the task. The Business task Controller is thus the executive class, and the User Interactor is an operational class. The User Interactor is fully responsible for the direct interaction with the user. It determines the number of screens required to achieve the desired result, and is fully in conversant with the particular user interfacing technology used. The Business Task Controller has no knowledge of these implementation details whatsoever.

5.3.1.2 Business Task Controller

The Business Task controller is the executive manager of a business process. This class will in fact be instantiated and called by the initial user interactor class, as described above. This class is responsible for executive management of the overall task. It does not execute task steps, that it is it does not execute business task logic. It is responsible for deciding which task actions should be taken, based upon the current state of the process (held in the business task state class), and *events* which could be:

- the User Interactor presenting data
- results returned by Business Task Executors
- business events which have been published by other business task controllers

The Business Task Controller is therefore a **state machine**, and should be modeled as such.

It is important to stress that the Business Task Controller is an executive class, implementing a state machine. It delegates actions to be taken in response to state machine state changes to other classes within the overall Business Task Component for the:

- storage of overall task state
- execution of business logic
- publication of and subscription to business events
- presentation of user views

5.3.1.3 Business Task Executor

The Business Task Controller calls a Business Task Executor, in order to execute a step within a process. The Business Task Controller will give a Business Task Executor a reference to the Business Task State instance. A Business Task Executor is allowed to change the state held in the Business Task State instance.

A Business Task Executor:

- Is called by the Business Task Controller in order to execute a piece of business task logic. The business task logic may involve:
  - The validation of user input, for example ‘confirm validity of data provided by user’
  - The processing of user input for example ‘update contract’
  - Interaction with the Function Components via Business Function Drivers, to cause updates to shared resources, or to obtain data from shared resources needed by the business task.
  - Publication of recognized business task events (task completion, significant step completion) to a publish/subscribe mechanism.

An advanced implementation, for tasks involving steps which could be executed in parallel, would use a deferred synchronous invocation to invoke several task executors which could execute simultaneously. The coordination of these concurrent tasks should be handled by a specialized Business Task Executor, not by the Business Task Controller itself.

5.3.1.4 Business Task State Manager

The Business Task State Manager class is responsible for the storage of all state related to the task for the lifetime of the task. This could include the management of state for a task which may require multiple ‘user sittings’, that is the task may not be completed in one user session. If that is the case, then the state must be serialized, written to some form of persistent storage, and revived when the task is restarted. Note that the ‘state’ of the Business Task Controller State Machine is also an item of task state which will be held within the Business Task State Manager.

The Business Task State manager should store state in a dynamic fashion. This means that the data stored should be keyed by a meaningful name, and should be retrievable by name.

The Business Task State manager should be available to all classes within the Business Task component, throughout the active user session. In a web server environment this would usually be achieved through the Http session object. Use of this should be encapsulated by a more generic state manager interface.

5.3.1.5 Business Function Driver

Business Function drivers are responsible for invoking transactions on Function Components. They provide a bridge between non-transactional business task components, and transactional Function components.

A Business task component is non-transactional for reasons of scalability. If possibly long running methods in task executors were to be implemented as transactional components, then they could possibly tie up shared resources (data) with transactional locks, for the duration of the long running method. The obvious answer would seem to be to ensure that only Function components provide transactional methods. This does not provide a complete answer to the problem however. Function Components provide services which could be used in a variety of different contexts by a variety of different tasks. A method to *set the expiry date of a contract* might be used in the context of a contract update task, as well as in the context of a contract creation task. Furthermore, within the overall contract creation task, setting the expiry date is a part of the overall transaction required, whereas in a pure update mode this may be all that the overall transaction entails. The important point being made here is that Function Component Services are stable, relatively low level transactions. Business Tasks however may need to compose several Function Component Services into one transaction, which is executed upon completion of the task itself, or upon completion of a significant step within the overall task. Predicting what these combinations are should not be the task of the Function components. Two Function components providing in total five Function methods provide 30 different combinations – and if the order in which the methods are called becomes significant, then the number of different combinations increases again.

The Business Function Driver is a class owned by the Business Task component. It has the behavior of a Function component however, in that it has a **transactional property** indicating that it requires a transaction. A transactional property is a property which is significant to the component container (see figure 2). This means that when a method on a Business Function driver class is called, a transaction will be started, unless a transaction already existed when the method was called. Business Function Drivers are thus composers of the specific transactions required by business task components. They should not be long running – they are called, execute their transaction and return control to the calling Business Task Executor. This ensures that the shared resources managed by the Function components are tied up for the shortest possible period.

The Business Function Driver also has a vital *abstraction* role. We envisage ‘hybrid’ implementations of the Technical Reference architecture, which feature Business Task Components being written using modern technologies (Java and Java Servlets, Visual Programming languages and ASP), and Function components written in traditional TP monitor environments, such as CICS. The Business Task components should not be directly aware of the fact that they are interacting with a CICS environment, especially if a migration to more modern technologies for the Function components is expected to take place over time. Clearly, consistent use of Business Function drivers, whose only role is the invocation of business service methods on behalf of the Business Task component, will provide this abstraction and enable migration.

5.3.2 Implementation Considerations

The following sections contain some brief points to be considered when implementing this architecture in a specific environment.

5.3.2.1 Session State

If this architecture is being implemented using thin, web browser based clients, then maintaining state throughout the session can best be achieved by storing a reference to the Business Task State Manager instance in the Http Session object. Care should also be taken to ensure that this instance is released when the task is completed, as the Http session object itself remains active until a timeout period is reached (typically 20 minutes).

5.3.2.2 MTS & EJB Implementation Classes

A Business Task Component is a grouping of runtime software components and objects of different types.

Class	Implemented as	Attributes
User Interactor	Servlet, ASP or JSP script, Conventional GUI EXE.	
Business Task Controller	Stateless EJB or COM class, or in Java environment simple Java class called by User Interactor (servlet)	Stateless
Business Task Executor	Stateless EJB or COM class, or in Java environment simple Java class called by User Interactor (servlet)	Stateless
Business Task State Manager	Stateful EJB or stateful COM class. Could also be simple stateful class if pseudo conversational state management is not required.	Stateful. Does not support transactions
Business Function Driver	Stateless EJB or COM class in MTS. This must be a transactional component class as it must initiate a transaction.	Stateless. Requires Transaction

5.3.2.3 GUI Implementations

This architecture can be implemented using straightforward GUI or applet technology. In this case the User Interactor must becapable of calling the Business task Controller.

5.3.2.4 Implementation in Traditional Environments

This architecture could also be implemented in a traditional environment, using technologies such as CICS and 3270 data streams. In such environments, component containers do not exist, so the Business Function Driver would need to explicitly start and commit (or rollback) transactions. Also, care would need to be taken if Business Function Drivers call other Business Function Drivers, to ensure that the transaction start, and commit or rollback is only executed in the ‘root’ Driver, that is the Business Function Driver which was first called and actually starts the transaction.

5.4 Function Components

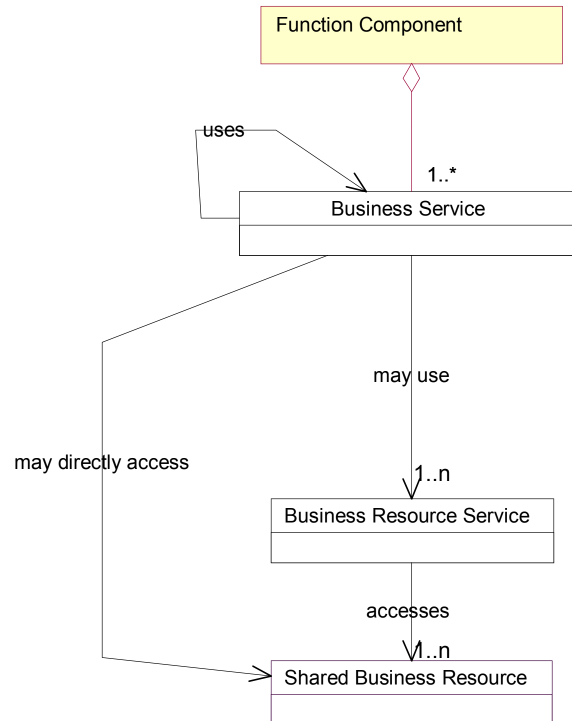

Figure 10: Function Component Meta Model

5.4.1 Constituent Classes

Figure 10 depicts the constituent classes of a Function Component. These are described below.

5.4.1.1 Business Service

A Business Service class is the main class within the component; the services that it provides are the services which are made publicly available to clients of the business service component.

Clients of a business service class could be:

- Business Task Components
- Business Service classes, which could be contained within the same function component, or within other function components.

5.4.1.1.1 Business Service State

Business services do not contain state that exists beyond the lifetime of the business service that is executing. Business Service methods are often said to be ‘stateless’, but in fact they may build up state which is used for the duration of the execution of the method. This state will disappear when the method has completed. This is important for two reasons:

1. Business Services often execute transactions, which cause mutations to persistent data managed by resource managers. If state is held in a component instance, which is supposed to reflect the actual state held in persistent storage, then it is essential that these two be kept in synch. Ensuring that state held within business service components can never exist beyond the lifetime of a transaction provides a foolproof solution to this problem.
2. Business Services are used by many different business task components. The function components are shared amongst business task components. If they are 'stateless', then the transaction server which houses the function components can **provide instance pooling**. This means that once an instance of a component has been created, it can be held in a pool of instances, and be used over and over again by different clients. A relatively small number of instances can be shared by a larger number of clients. This reduces the amount of system resource needed to manage these component instances and cuts down on the overhead of creating these instances. If instances contain state that exists between method calls, then each client would need a dedicated instance of a function component, which would limit scalability.

Business Service class methods should therefore be coded in such a way that they provide pure business services, which adhere to the 'GIGO' (get in get out) principle. They should not tie up shared business resources. They should provide data mutation type services, as well as calculations and algorithms. They should never interact directly with users, and they should never hold state that exists beyond the lifetime of the method being called.

5.4.1.2 Business Resource Service

Business Resource Services provide access to **resource managers**. A resource manager is a database or a message queue. Business Resource Service classes are private, and they are owned by, and fully contained within a Function component. Business Resource Service classes should contain functionality of the Create, Read, Update, Delete (CRUD) type. They should represent instances from the *logical*, as opposed to the physical model. We could encounter a situation where a logical entity has been modeled called 'Product', which contains general product description data as well as cost and pricing data. Our physical database may in fact contain two separate tables – *product master* and *product cost master*. In this situation, the Function component class *product* would contain and make use of one business resource service class, also called product. This ensures that the code contained within the Function component methods is compliant with a logical model, as opposed to the physical model. The business service method code is therefore more resilient, and could potentially be used in another environment where the two tables have been merged into one.

Create, Update, Read and Delete services should operate on single instances of the logical entities they represent. Read services may also provide:

- List type services which return information relating to multiple instances of the entity in question
- Explicit join type services which return information relating to (multiple) joined instances of different logical entities. This is a pragmatic, practical approach. If this were not permitted, then executing joins and list retrieval via single instance reads limited to single logical types would be disproportionately complex, and would provide poor performance.

5.4.1.3 Shared Business Resource

A shared business resource is not a component. In practice, the architecture assumes that the shared business resource is a database or possibly a message queue belonging to a message queuing product. Furthermore, the assumption is made that the shared business resource is a

resource manager that is capable of participating in 2 phased commit transactions. This means that a transactional method called on a business service which:
1. updates a database (via a business resource service), and
2. puts a message on a queue

… only actually *does* both of these things when the transaction is committed. In other words, the database will not be updated if the message transmission fails, and the message will not be sent if the database update fails.

Again, the vital point about shared business resources is that their type, structure, and access methods are fully encapsulated by business resource service classes.

5.4.2 Implementation Considerations

The following sections contain some brief points to be considered when implementing this architecture in a specific environment.

5.4.2.1 Use of XML

The use of XML should be considered for any Business Service which expects, or returns self describing lists of data which have dynamic length and structure. Standard XML parsers should be used to parse the data.

5.4.2.2 Use of Recordsets

Many business resource service methods, will result in lists of data being returned, or may require lists of data as input. If the component model being used supports the use of ‘recordsets’ then these should be used. In a Microsoft implementation ADO (Active X Data Objects) can be used.

5.4.2.3 Function Component Granularity

Perhaps the single most important implementation issue is that of Function component granularity.

In general, adoption of this reference architecture will result in the creation of several Function components, each of which ‘owns’ a database. This in contrast with the approach taken currently by ERP package vendors, where a large package makes use of one extremely large database. The Function component approach promises greater modularity, but it does introduce a problem of data dependencies between components.

The crucial question of granularity is all about the optimal size of Function components. If these components are small, then there will be a great deal of ‘inter component’ traffic. Tight relationships will exist between databases. *The problem being described here is all about databases*. These interdependencies can be eliminated by increasing the scope of the database owned by the Function component, but this can lead to enormous business components being built, which completely defeats the object of the component exercise.

Two rules of thumb which should be adhered to in deciding the granularity of Function components are:

1. Ensure that there are no list retrievals that involve fetching a list which contains data supplied by more than one Function component. Obtaining such lists *across* databases (i.e. across Function components) will prove to be complex, error-prone and expensive in terms of performance. Instead, this should be solved by allowing some basic data elements which are not owned by the Function component in question to be copied to, and held within the database of the Function component. A contract management Function component might, for example, contain a customer number inside each contract record, and may contain some basic

information about the customer (address for example). This would allow list retrievals to operate solely within the scope of the single database owned by the contract management database. This data copying does of course lead to the need to introduce a synchronization mechanism between the master Function component managing the customer database, and the various Function components which store subsets of its data. **Publish and Subscribe middleware** provides an ideal solution for this general problem.

2. Design Function components such that the majority of calls between Function components are asynchronous calls, which are not time dependent. These are invocations of the **enterprise level** service style identified in 4.5.2.2. Messages (event notifications) which do not require a response, or do not require an immediate response should constitute the majority of traffic between well designed Function components. Synchronous calls and transactions should be contained within a Function component as far as possible. Clearly, this goal can never be attained 100%, because the only way of achieving this would be to create enormous, monolithic components. It is, nevertheless, a goal which should be aimed for as it will lead to the creation of loosely coupled and manageable Function components.

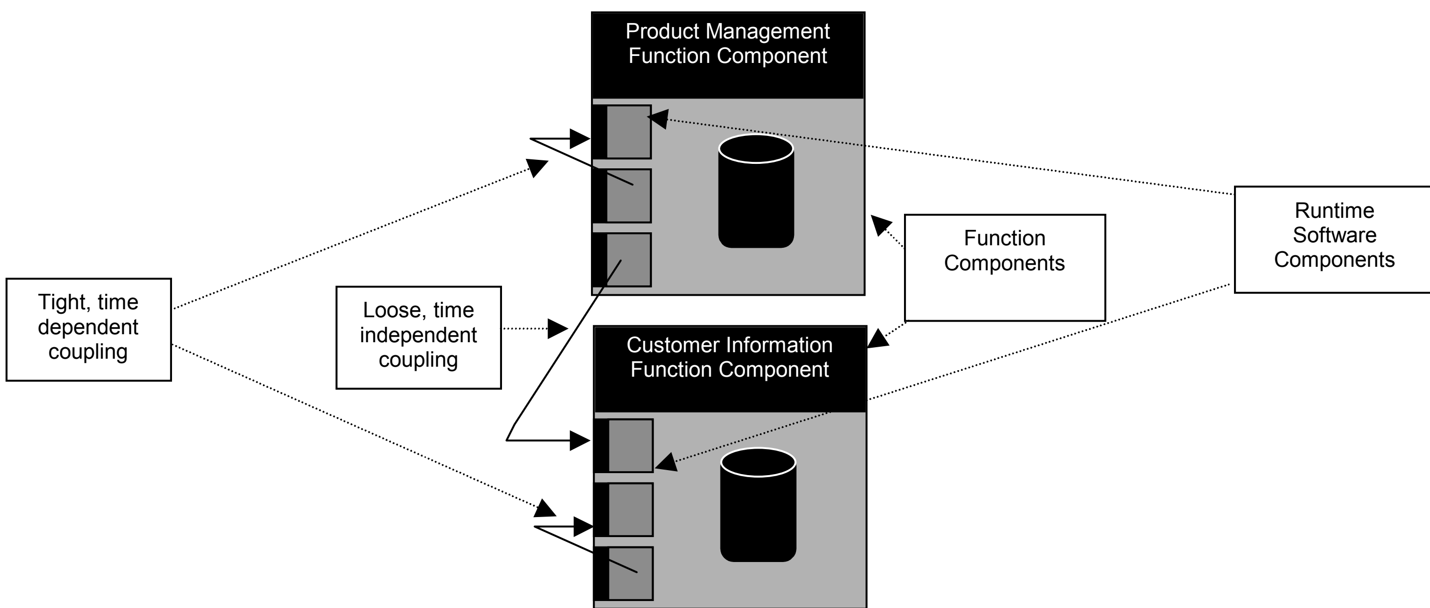

Figure 11: Function Component Granularity

5.4.2.4 MTS & EJB Server Attributes

If Microsoft Transaction Server or an EJB server is used, then the following properties apply:

Class	Transaction Property	Stateful/Stateless
Business Service	Supports	Stateless

5.5 The Complete Technical Reference Architecture & Security

The Technical reference architecture for business components is depicted below.

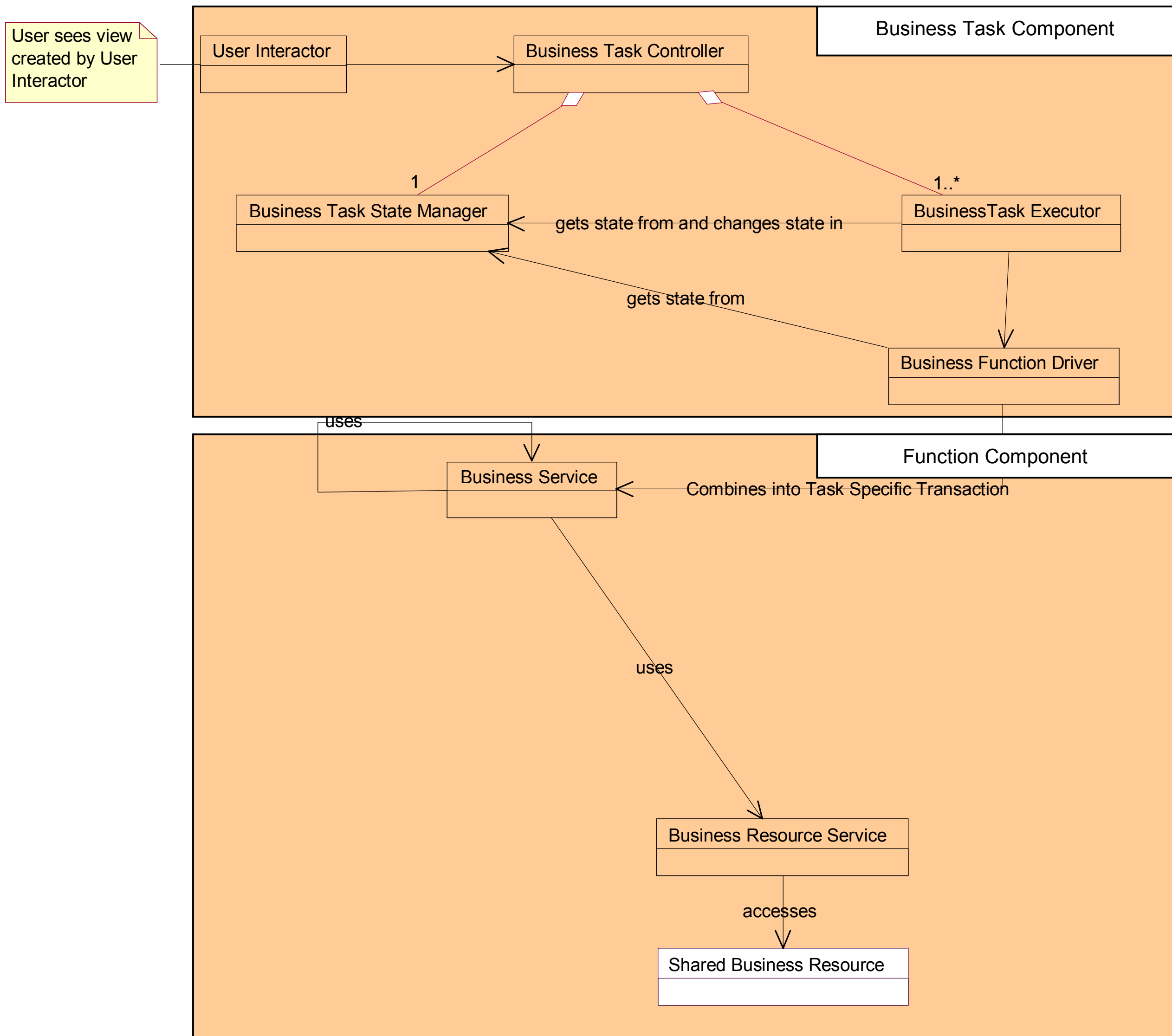

Figure 12: Business Component Reference Architecture

One overall aspect that has not yet been discussed is that of security. The rest of this section addresses implementation details regarding security.

5.5.1 Security

The Technical Reference architecture does not provide any specific structures which are designed to deal with security. In our view, component containers, component servers and web servers should handle security. In other words, standard products and approaches should be used.

5.5.1.1 Secure Socket Layer

Component servers (EJB servers, MTS etc.) and web servers should offer encrypted, authenticated connections via Secure Socket Layer (SSL).

5.5.1.2 Declarative Security

In the case of EJB and MTS implementations, component container security can be applied to each individual component interface, via declarative security, which allows lists of authorized users to be identified for each interface. Managing security through application code should be avoided. Although there is one exception to this – **role based security**. Certain services, or interfaces, cannot be managed from a security point of view solely through providing lists of authorized users. It may be the case that security restrictions become more or less stringent depending upon the values of variables involved in executing a particular method. A stock transfer would be an example of this – if the value of stock to be sold exceeds $1,000,000 then the person making the transfer must have the role of 'senior trader'. Whereas if the value is less than $1,000,000 the user making the transfer could have any role. At the same time, the interface itself would have a list of all users that are allowed to make use of its services, which would include both traders and senior traders. Role based security therefore provides an extra filter, with added flexibility, to a simple all or nothing list of authorized users.

5.5.1.3 Granularity Of Security

Care should be taken regarding the granularity at which component container security can be applied. Ideally, security (access control lists) should be applied at the individual method level, or better yet using an inheritance scheme whereby security is applied at the level of a component as a whole, which by default then applies to all interfaces and all methods within each interface. At the interface level, the default can be overridden, which provides a new default for each method in the interface. Again, the default established can be overridden at the level of each method in the interface.

With COM, lowest level at which security can be applied is the interface. The EJB model allows security to be applied at the method level.

In general, we would advise that a security solution should be found which involves the minimum of application code.

6. Supporting Infrastructure

This section describes the major characteristics of the Technical Infrastructure (TI) required to support implementations of the Technical Reference Architecture. We restrict our discussion to the most vital TI components within its context, namely:

- Transaction Servers
- Communications Middleware
- Message Brokers

6.1 Component Technologies

Up until recently, there were no standard component models. Object oriented languages have been around for a long time, and these were often used to build frameworks which included some sort of component model. Each component model was proprietary however. There were no standards which defined how components would define or project their interfaces, or how components would find one another, or how components would interact with underlying middleware. A **component model** was missing, and as a consequence of this the idea of reuse and off the shelf purchase of components never really came to fruition.

Things have changed, and it is widely accepted that there are now **two** standard component models:

- Enterprise Java Beans (EJB)
- Microsoft COM+

These component models, together with their associated transaction servers (EJB servers and Microsoft Transaction Server) can both be used to construct new component-based solutions which are built to the reference architecture. **Both component models are almost exactly the same**. Both of them are predicated on a container structure. The important differences between COM and EJB are:

- COM components can be written in many different languages (Visual Basic, C++, Java, Delphi) whereas EJBs can only be written in Java.
- COM components run on Windows platforms only, whereas EJBs and their associated EJB servers are portable, and therefore run on a wide range of platforms.

6.1.1 Products

On the COM+ front, the key product is **Microsoft Transaction Server**. This is a highly programmer friendly product, which provides distributed transaction coordination (DTC), as well as connection pooling and component instance management. Microsoft Visual Studio provides a highly COM aware development environment for COM/MTS components.

On the EJB front, several EJB server products are available. **IBM Websphere** and **BEA Weblogic Server** are probably the most significant EJB servers on the market. These products enable scalable transactional EJB solutions to be built, and are starting to introduce mission critical features such as load balancing, replication and fail-over. Both of these products also support **Java Servlets**. BEA Weblogic server is a fully functional web server.

6.1.2 Traditional Technologies

An implementation based on the Technical Reference Architecture can certainly be achieved based upon more traditional technologies, or a mixture of traditional technologies and more modern component technologies. Building Business Task Components in Java, using EJB, or COM+, together with Function components realized in a traditional TP monitor environment such as CICS or TUXEDO would certainly be feasible. Points to watch out for are:

- The container approach does not apply within traditional TP monitors. Separation of business logic and 'technical logic' (transaction coordination, security) would need to be addressed carefully.
- Distribution and well-defined naming services are less clean. Again, some form of abstraction would need to be defined.
- In the case of a hybrid implementation, the business task components should not be aware of the fact that they are calling Function components from another environment. This requires the consistent use of Business Function drivers, which encapsulate Function Component service access.

6.2 Message Brokers

A message broker fulfils the following key roles:

- **Translation -** of standard service requests into requests that application wrappers understand.
- **Flow control** - The capability of flowing a Function Component service request which might involve several messages being sent to one or more target applications, taking maximum advantage of parallelism.
- **Intelligent routing** - The contents of the incoming message allow the message broker to determine the destination to which the message is to be sent, and whether it is to be sent at all.
- **Message protocol translation** – The ability to act as a protocol switch between messages sent to recipients using more than one network/middleware protocol.
- **Interface Management** - Interface certification procedures, version control, regression testing, and runtime features such as replication and fail-over.

6.2.1 Message Broker Styles

Although all major message brokers more or less provide all of the features described above, we can classify message brokers into two distinct styles. The style chosen has a direct bearing on the quality of service, ease of maintenance, ease of development and cost of the message broker.

The two styles are **Non-Invasive** and **Super API Based** message brokers**.**

6.2.1.1 Non-invasive Message Brokers

This style of message broker is best explained using a simple example. In this style, application A constructs a message using the APIs provided with middleware product A – MQSeries perhaps. The message is sent to a queue which is owned by the message broker. The message broker process reading this MQSeries queue is a **middleware adapter**. Its function is to receive the incoming message, strip the specific protocol related information from the message, and pass the pure data carried in the message to the core message broker processes. The core message broker processes are now working with pure data – validating, formatting, establishing routing etc. At some point, one or more messages are created and sent to other applications – the Message Receiving Application B depicted in figure 10. Again, the message broker uses middleware adapters to package the data into the appropriate protocol used by the middleware that application B is 'listening to'. This could be an NT application reading Microsoft MSMQ queue for example. In a

more homogeneous environment, where only one protocol is being used, the same interactions between middleware adapters and the core message broker processes would nevertheless occur.

These message brokers are non-invasive, because they *intercept* messages that are already being sent by the sending application. The messages that it constructs and sends to the receiving application are messages that the application in question expects to receive. In other words, neither the sending nor the receiving applications are aware of the message broker.

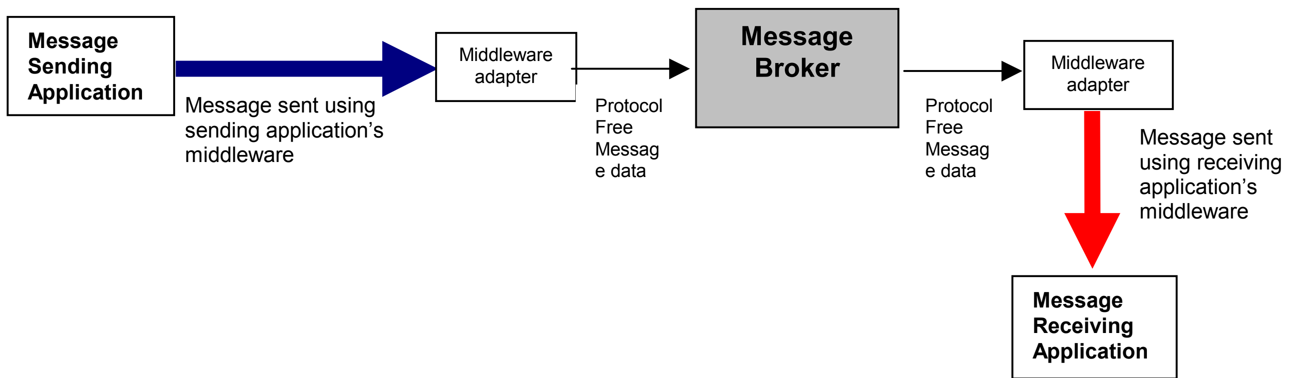

Figure 13: Non-invasive Message Brokers

6.2.1.2 Super API Based Message Brokers

Super API based message brokers make use of a generalized messaging API, which is layered above enterprise level middleware, (see Figure 8). This generalized messaging API provides the standard features common to all message queuing products, allowing messages to be created, populated, transmitted, read etc

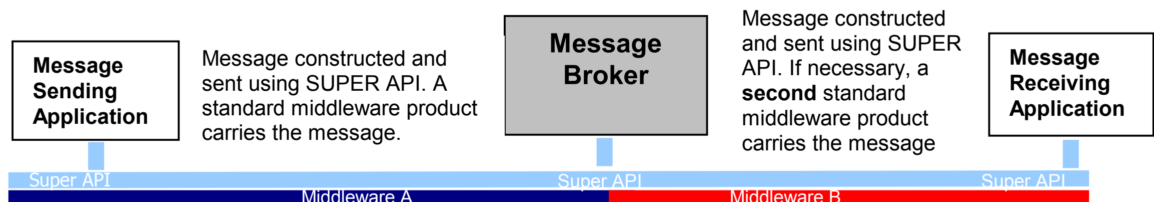

Figure 14: Super API Based Message Brokers

This form of message broker presents its enterprise level services in the same way that Function components provide their services. It should use the same naming service, and the same middleware. These message brokers are in fact **virtual Function components** in a component based environment as described by this architecture.

6.2.2 Self Describing Messaging

Both forms of message broker can make use of self-describing messaging. In the case of non-invasive message brokers, this would mean that the sending and receiving applications would already be constructing and receiving self describing messages. This may become more prevalent as XML is adopted, but in general, today's applications tend to send and receive 'buffers' of data. Certainly, newly constructed function components may well choose to articulate enterprise level service requests and responses using XML.

Advanced forms of Super-API based message brokers may in fact be ‘self-describing centric’. This means that their API is used to construct and read self-describing messages, and the message broker itself is configured in terms of manipulating the name/value pairs contained within the messages. This can prove to be less complex than the ‘buffer dissection’ exercise which is necessary in order to configure non-invasive message brokers.

Some key advantages of using self-describing messages as a means of message exchange between enterprise level services are described below.

6.2.2.1.1 Advantages of Self Describing Data

Interfaces become change resilient and flexible:
- Addition of new data elements (fields) does not “break” the interface
- Supports changes to data-types and value lengths
- Allows different information consumers (clients) to utilize the same interfaces
- Recognize that different applications are changed and released at different rates
- Static interfaces require coordinated upgrades
- Data elements (fields) can be extracted in any order simply by referencing the name.
- Extraneous data is simply discarded
- Easier to add elements to provide “context” of the message.

Multi-purpose service providers (servers)
- Easy to write servers that provide many services
- Fewer processes to manage and few programs to maintain
- Promotes dynamic wrappers

Simplifies interface contract
- Interface specification is self-documenting
- Need only to know the data element names
- Allows for message and data element interrogation

Easier to trouble-shoot and maintain
- Names provide first level of documentation
- Easy to display data in “human readable” format that is meaningful and supportable
- No more counting and adding field offsets
- The meaning of the data is passed with the value of the data

Easier to support repeating data sets
- Supports unknown message lengths resulting from varying numbers of repeating elements
- Easier to determine when last element has been read.
- Eliminates the need to map complex record structures.

6.2.2.1.2 Disadvantages Of Self Describing Data

- Larger messages are required
- May require additional translation – Name as well as value
- Lack of consistent naming can pose a problem
- Somewhat new programming technique and style
- Building self-describing messages vs. loading a static record structure results in performance penalties

6.2.3 Commercial Message Broker Products

ComCor has solid working partnerships with the vendors of the following products.

6.2.3.1 Muscato Engin

This is a non-invasive message broker. It is a robust product, which offers excellent performance. It is used widely by diverse companies across the world. This product is in its element when used as a non-invasive integration engine between existing applications.

Engin is capable of full participation in a Business Component oriented architecture. It provides:

- **JEngin** – a tool which generates a Java proxy class for any given message request/response pair provided by Engin. The generated code includes a class which represents the request message, and a class which represents the response message. An executor class is also included which handles the transmission of the request, and parsing of the response into the response class. The synchronous TCP/IP sockets protocol is used between the generated proxy and Engin's TCP/IP middleware adapter (this is called the TCP/IP transaction manager). The generated Java classes can be used standalone, or as is within Java applets, Java servlets, EJB components or Microsoft COM components.
- **A CORBA bridge**, by which CORBA becomes one of the middleware protocols supported by Engin. The advantage of this approach is that in a native CORBA environment, Engin is regarded as a CORBA server in its own right, in other words it is non-invasive.

For more details, visit: www.muscato.com

6.2.3.2 PRL I/O Exchange

This is a Super API together with a Super-API based message broker – the 'Interface Engine'. The product is agile, powerful, relatively easy to use and relatively inexpensive. PRL I/O Exchange can be obtained both directly from PRL Scotland ltd., or resold via Muscato. It provides full support for self-describing messaging.

The PRL I/O Exchange Interface Engine provides a proxy generator, which is capable of generating Visual Basic, C or Java code for any given Interface. Transmission of requests and responses is handled by the Super-API, which can be layered on:

- IBM MQSeries
- Microsoft MSMQ
- BEA MessageQue
- Netweave

The proxies can also be encapsulated within COM classes which can formulate their response value as an XML formatted string, which can then be parsed in web scripts etc. using industry standard XML parsers.

It is also important to note that the SUPER API provides a fully object oriented Java interface. Also, through a simple configuration switch, the product will format all messages which it creates using XML. This allows users to choose between the product's internal efficient formatting of self-describing messages, or formatting the same messages using XML. The API which manipulates these messages is not affected by the formatting choice made. This also means that

For more details, visit: www.prlscotland.com

6.3 Application Wrappers

In the case of real time messaging between applications, via message brokers, *wrappers* need to be created which *adapt* the applications in order to enable them to participate in the distributed environment. [3]

Specifically these wrappers ensure that 'wrapped' applications are capable of:

1. Producing requests to be sent to the message broker (call out)
2. Processing requests received from a message broker (by calling routines within the application, running screen-based programs, accessing the application's database etc.)

Wrappers should be **dynamic**. This means that a wrapper should not present a set of statically defined business services, rather it should provide a route through to the application articulated in the 'you tell me what to do and I'll do it' form. A wrapper which executes SQL requests against a database server, a wrapper which executes user functions via a screen scraper, or a SAP R/3 wrapper which executes named SAP functions are all good examples. None of these wrappers need to be changed when the application itself is changed. Static wrappers on the other hand simply add to the often already unbearable maintenance burden.

The interfaces projected by the legacy application domains in Figure 8 are wrapper interfaces.

[3] In the case of applications which already send and receive messages of some type, wrapping may not be necessary. In these cases, a non-invasive message broker provides an ideal solution.

7. In Conclusion

The Technical Reference Architecture has been described in terms of:

- Its architectural positioning
- High Level Guidelines and Principles
- Meta Models for the construction of new Function and business task components using modern runtime software component technology such as COM, EJB or CORBA components.
- Implementation considerations and candidate implementation technologies

We have stressed the point that the Technical Reference Architecture addresses two important aspects of information technology that are often addressed separately:

- Component Based Solution Construction
- Component Based Solution Integration

Rather than treat these as two separate items, we believe strongly that they are inseparable and should always be addressed together, for all but the most trivial green-field type of applications.

The Technical Reference Architecture strives to be general enough to be implementable using a variety of different technologies, whilst at the same time providing sufficient depth to be meaningful, and useful as a reference upon which implementations can be based.

The architecture does not provide an academic or novel approach to these problems. It is firmly based upon best practice, and approaches which are attainable, comprehendible and, hopefully, full of common sense.

8. Glossary of Terms

ACID Transaction	A Transaction which is: • **A**tomic – it is a single unit which succeeds or fails as a whole • **C**onsistent – a completed transaction whether it has succeeded or failed will always leave the resources that it has operated on in a consistent state • **I**solated – its effects are invisible to the rest of the system until it has completed, and it is unaware of the effects of other transactions running concurrently • **D**urable – the changes that it makes to the resources that it operates on are persistent
Asynchronous invocation	A client sends a message to a server. The client does not block and wait for a response, it carries on with other work. At some point a response (in the form of a message) may be sent. This 'response' could be handled by polling for responses at the client's convenience, it may be handled as a callback on the client, or it may simply be handled as an incoming message to the client, along with all other messages that the client receives. The key point is that an asynchronous invocation is not a standard synchronous call.
Business Component	**Business components** are the logical design modules in a service-oriented architecture. These components may be developed on any platform using any technique. They provide services through an interface. Business components are not necessarily built using runtime software components, although this will tend to be the case in modern software development projects.
Business Solution	A business solution is a set of business components (business task and Function) which when put together provide a solution to a business problem (contract management for example).
Business Task	A business task is a single task that may involve a series of interactions with a single user via some form of dialogue. It is usually accomplished in one sitting. It is not the same as a workflow process, which may involve several tasks, which are handled by several users, perhaps concurrently. A workflow is rarely completed within a single sitting. An example of a business task is 'calculate premium', an example of a workflow (aka business process) is 'handle insurance claim'.
Business Task Component	These interact directly with users in order to complete a set of logically related business tasks – see business task above. A business task component may be capable of handling several different (related) business tasks – premium calculations may for example contain several different premium calculation tasks for different kinds of insured object.
Component Based Solution	A component-based solution is a business solution which makes intensive use of both business components, and runtime software component technologies such as COM, CORBA or EJB as a means of implementing the business components.
Container	A container is the environment in which a runtime software component runs. The container is provided by the vendor of the component's runtime environment (i.e. the component server – MTS, an EJB server, a CORBA component server etc.). The

	container handles all low level interactions with the operating system on behalf of the runtime software component. This means that the business code implemented by the runtime software component is responsible for dealing with the business problem, and does not need to deal with transaction management, security, pseudo conversational behavior etc.
Deferred Synchronous Invocation	This is a form of invocation in which a client sends a request to a server. It may in fact send several requests to one or more servers concurrently. The client may then carry on with other work. At a point determined by the client, it explicitly waits for responses to the requests which it has issued. This is slightly different to asynchronous invocation where responses may not be required at all, are treated as requests the same as any other incoming requests, or are handled as callbacks on the client.
Enterprise Level Services	These are business significant services which are provided by business solutions, and are used by other business solutions. Ideally, they are largely asynchronous in nature. In the real world, they must be made available in a highly heterogeneous environment, therefore application wrapping techniques are usually required to enable these services.
Function Component	These provide transactional services related to a logical domain. These services are often closely concerned with the manipulation of data, and a Function component usually encapsulates persistent data.
Interface Engine	This term is almost synonymous with message broker. The term however is a better term in that it emphasizes the fact that a message broker should be capable of handling multiple concurrent invocations of backend systems, and must perform extremely well.
Message Broker	See interface engine above. A message broker provides translation of both middleware itself, and the content of messages. In a world of component based solutions its services in the same way as other function components in the environment provide their services. It is a vital component in a **migration strategy**.
Middleware	Middleware provides robust communication and runtime component services in a heterogeneous environment. Key middleware product groups are: • Remote Procedure Calls (RPC) • Message Oriented Middleware (MOM) • Object Oriented Middleware • Transaction Monitors • Database Access
Object	An object is a construction used within object oriented programming languages such as C++, Java or Smalltalk. They are programming level constructs, which may be used within runtime software components.
Runtime Software Component	Runtime software components are packages of programs managed as a unit and accessed only through interfaces. Runtime software components can be distributed – in other words they run in a middleware infrastructure. Com+, Enterprise Java Beans (EJB), and CORC (Corba Component model) provide standard (and competing) component models for runtime software components.

Self describing messages	These form the bedrock of adaptable, flexible business solutions. Self describing messaging provides a flexible way of invoking services and providing service responses in a dynamic environment. This as opposed to using static messages, and providing static services which makes systems highly sensitive to change. Self describing messaging is particularly powerful as an instrument to be used in legacy application wrapping. It is also highly appropriate as a means of formulating the request and response messages used within enterprise level services.
Solution Level Services	As opposed to enterprise level services, these provide lower level services within a business solution. They are usually invoked using a synchronous invocation model, and they often involve transactions. 'Write entry to Log', or 'getAddressLine2' are examples of solution level services. They usually operate within a homogeneous technology environment (e.g they are all COM services within a set of business components written using Microsoft COM technology).
Synchronous invocation	This is a form of invocation in which a client calls a service and blocks until it has received the result of that service. Synchronous invocation is needed within runtime software components, and between runtime software components as solution level services (see above).
Wrapper	A wrapper, used in the context of legacy application wrapping, is a server which provides access to the wrapped application in some way. A wrapper should be dynamic – i.e. it provides a generalized access to the application such as access to any named module within the application, the invocation of any named user process within the application via screen scraping, or the access of any named data within the application's database. In order to provide general dynamically invoked services of this nature, the wrapper should be invoked using self describing messages.