A Reference Architecture for Component Based Development

Keywords: Architectural layering, OO design, component based design, UML, packages, re-use.

Mark Collins-Cope (markcc@ratio.co.uk) and Hubert Matthews (hubert@ratio.co.uk)
Ratio Group Ltd.

1. Abstract

This paper proposes a reference architecture for object-oriented/component based systems consisting of five layers. Our purpose is to show how this model helps us to understand the overall structure of a system, how layering helps to clarify our thoughts, and how it encourages the separation of concerns such as the technical v. the problem domain, policy v. mechanism, and the buy-or-build decision.

Assuming an application is made up of a number of components, the layering we propose is based on how specific to the particular requirements of an application each component is. More specific (and therefore less reusable) components are placed in the higher layers, and the more general, reusable components are in the lower layers. Since general non-application components are less likely to change than application specific ones, this leads to a stable system as all dependencies are downward in the direction of stability, and so changes tend not to propagate across the system as a whole.

As well as presenting the reference model, this paper also discusses and clarifies in concrete terms the *meaning* of one architectural layer being above another. Perhaps surprisingly, our background research has shown that the meaning of the layering metaphor is the subject of some confusion. Specific examples of this are given in the paper.

The model presented contains five layers, which are as follows: the interface layer; the application layer; the domain layer; the infrastructure layer; and the platform layer.

2. Introduction

Architectural layering is a visual metaphor whereby the software that makes up a system is divided into bands (layers) in an architectural diagram. Many such diagrams have been used, and by way of introduction we show two of these.

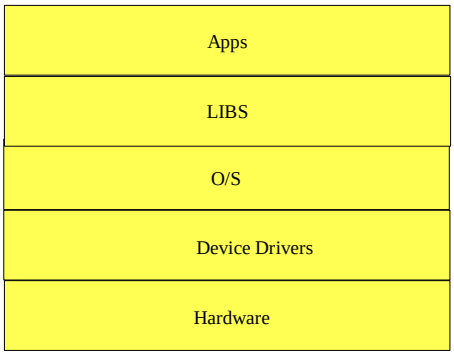

Figure 1 – Layered architecture – Example from Szyperski

Szyperski [Szperski98] presents a view of a strictly layered architecture as can be seen in figure 1. Note that this model has the device drivers below the operating system - a topic we will return to discuss later in this paper.

Figure 2 shows a type of ad-hoc architecture diagram [Carlson99] that is not uncommon in modern technical documentation. The example shown describes the architecture of the IBM San Fransisco product.

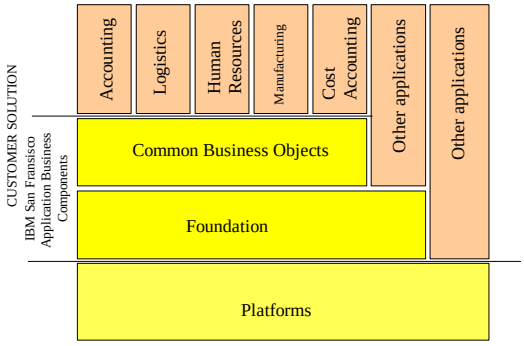

Figure 2 - Typical 'ad-hoc' architectural layering diagram

Some common themes run through these diagrams:

- that it is possible to identify a number of layers in the construction of pieces of software,
- that some layers sit on top of others (although there may be some question as to what one layer being above another actually means – see section 3.4), and
- that one may broadly categorise layers as being either horizontal (applicable across many, if not all business domains), and vertical (applicable across a subset or only one domain);

Turning to UML class diagrams (a younger notation), we notice that common convention usually place subclasses, which are more specialised, below their parents, which are more general purpose. This convention is the exact opposite of the architectural convention *highest is most specific*, and the cause of a undoubtedly confusing visual metaphor mismatch which we discuss further in our article *The Topsy Turvy World of UML* [Collins-Cope+00].

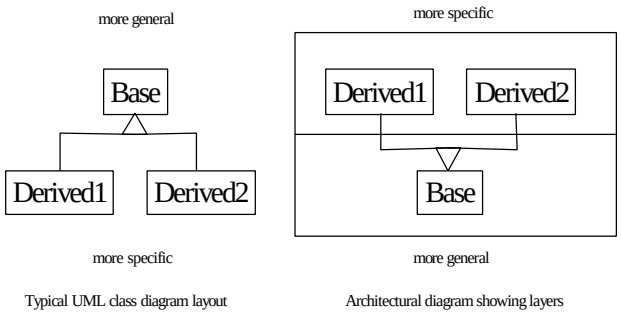

Figure 3 - Class diagrams and architectural views

This paper takes a revised look at application layering, with a particular focus on clarifying the unstated assumptions in such diagrams, and proposes a five layer architectural reference model for component based OO applications that can be used to assist in the design of component based systems.

3. Proposed model

3.1. Motivation

The objectives behind the architectural reference model presented in this paper are as follows:

- to provide a framework for decision making during the design of components,

- to support and re-enforce the appropriate application of good OO design principles, in particular those concerned with stability and dependency management,
- to provide an architectural framework to encourage re-use,
- to encourage re-use of business specific (not just technology) components,
- to position components as the unifying concept that tie together different architectural views of a system, and
- to provide clarification on the meaning of layering in a component context.

We come back to these motivations in the conclusions section of this paper.

3.2. Reference Model

We define the architecture of a system as *the structural relationship between the individual components that together create an application as a whole*[1]. We define a component as *an (object-oriented) software development deliverable implementing a well defined interface that is released at the binary (or equivalent) level*[2]*, which may have a number of well-defined extension points to enable it to be customised.*

Examples of components conforming to this definition might include '.o' or '.a' files on a Unix system, '.obj', '.dll' or '.ocx' files on a Windows based system. Components developed within a COM or EJB type environment are, of course, equally within this definition[3]. Note, however, that in most of the following discussion we consider the *design view* of components, which we represent as packages in UML notation.

Figure 4 shows our proposed reference model. Figure 5 shows the same model populated with a number of classes, components and relationships between them, taken from an example banking application. Two external *actors* [Jacobson+94] interact with the system: a bank clerk (using a debit dialog box), and an external banking system (using an intermediate file format).

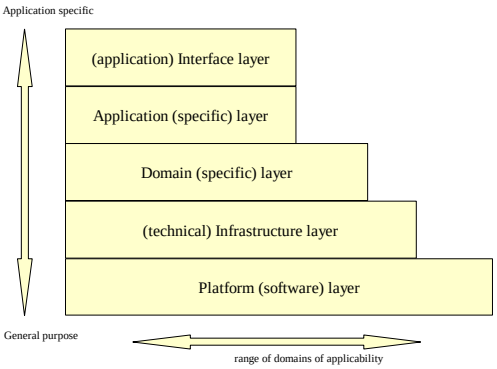

Figure 4 - Layered architecture reference model

1 We accept there are other defintions!

2 Three points here:
a) Framework is a term for a software deliverable which is generally assumed to 'drive' an application, leaving hooks for customisation. As can be seen in figure 6 - in which certain components 'drive' higher layers, whilst also being 'driven' by them, we do not see a clear distinction between components and frameworks, but rather view them as a continuum. We use the term 'component' generically across this continuum.
b) For the purposes of this paper, we discuss components as an extension of object technology, but note that we accept that components can be written in non-OO languages.
c) This definition presented does not preclude source parameterised components: in this case one would generate the binary file having instantiated the parameters in order to fit it into the framework described.

3 We see technologies such as CORBA, COM and Java Beans as *component inter-operability support technologies*, which may or may not be present in a 'component based system.' In the example shown in figure 6, they are not present.

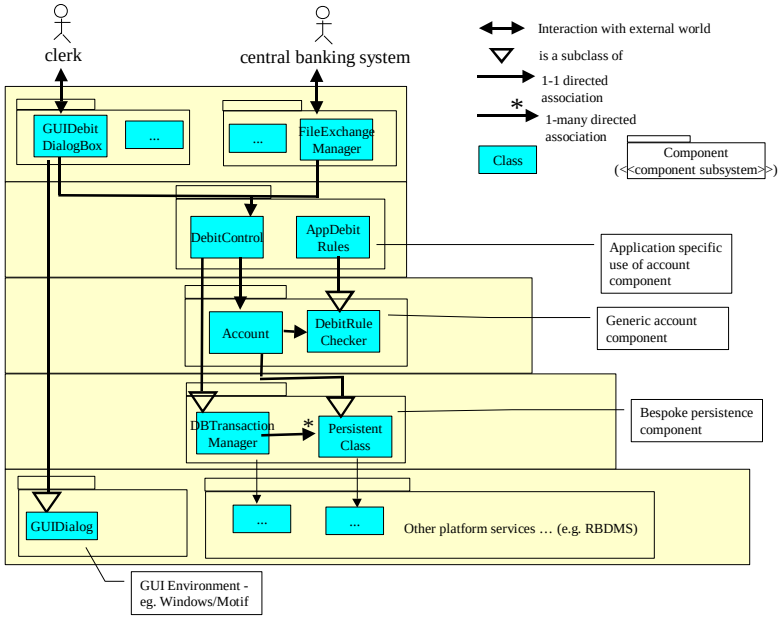

Figure 5 - Layered architecture populated with an example

The layers presented in this model may be summarised as follows:

- Highest (and most specific) in the layering is the application *interface*[4] layer. This layer is responsible for managing the interaction between the 'outside world' and the lower layers within the application. It typically consists of components providing GUI interface functionality - managing the interface to human users, and/or components providing external system interfaces - managing the interface to external systems. This layer often contains what Jacobson et al. call *boundary* classes [Jacobson+94].

  In the example application shown in figure 6, we can see two classes packaged within this layer. The *GUIDebitDialogBox* class implements an application specific dialog (to enter a debit). The *FileExchangeManager* class reads an external file format. Both classes use the application specific *DebitControl* class to process the information they receive from the outside world.

- Below this is the *application* specific layer. This layer is comprised of objects and components that encapsulate the major business processes and associated business rules automated within an application. Typically it will contain many objects akin to Jacobson's (use case) 'control' objects [Jacobson+94]; and often also acts as the 'knowledge' layer in Fowler's operational/knowledge split [Fowler98] (another description of this is separating policy from mechanism.) It may also contain specialised subclasses implementing interfaces left 'open' (as in Open Closed Principle [Meyer97][Martin96]) by the more general purpose components in the layer below, and typically does not contain persistent business classes. Most importantly, this layer contains the "glue" to tie together components within the domain layer below.

  In the example application shown in figure 6, we can see two classes packaged within an application specific debit component. The *DebitControl* class takes over application control when asked to do so by one of the higher level interface classes. It then drives the domain level *account* class to implement its functionality (which may involve several method calls on *account*). Note that the *DebitControl* class is derived from a database transaction management class defined in the bespoke persistence component in the

[4] We chose the term interface rather than 'presentation', as some application interfaces are to external systems, and the term presentation tends to imply a user-interface.

infrastructure layer[5]. The other class - *App. debit rules* - implements the debit-rule checker interface (derived from the lower level *account* component) to customise the debit checking rules as required by this application.

- Next is the business *domain* specific layer. This layer is comprised of components which encapsulate the interface to one or more business classes, which are specific to the domain (area of business) of the application, and are generally used from multiple places within the application. They might also be used by a family of related applications - a software product line. This layer typically contains the 'entity' classes discussed by Jacobson et al in [Jacobson+94].

  The example application shown in figure 6 shows an *account* component in this layer. The *account* class is driven by higher level components to undertake account related activities such as debiting and crediting of monies. As part of this, it uses a *DebitRuleChecker* interface (abstract class) to enable individual applications to customise the particular debit checking rules that may be applied (e.g. can go overdrawn, can't go overdrawn, etc.) This is an example of the open/closed principle [Meyer97][Martin96] being used to implement an operational/knowledge split [Fowler98]. Note also that, being persistent, the *account* class is derived from the *persistence* class in the infrastructure layer[6].

- Then comes the technical *infrastructure* layer. This layer is made up of *bespoke* components that are potentially re-usable across any domain, providing general purpose 'technical' services such as persistence infrastructure, general programming infrastructure (e.g. lists, collections).

  The example application shown in figure 6 shows a general purpose persistence component being present in this layer. In this component, a *DBTransactionManager* class keeps tabs on a number of *PersistentClasses*[7], which provide the hook by which higher level domain classes many be made persistent.

- Finally, most re-usable of all, is the *platform* software layer. This is comprised of standard or common-place pieces of software that are *brought in* to underpin the application (e.g. operating systems, distribution infrastructure (CORBA\COM), etc.) The example application shown in figure 6 shows a relational database and a GUI class library being used to build the application.

3.3. Associated rules

Some simple rules are associated with this model:

- there should be a clear and simple mapping between component structure and source code structure (the simplest being a 1-1 mapping), and between the component structure any other analysis and design artefacts produced during the development process (e.g. the design view of a component, as shown in a package diagram, or the use case view of a component, as shown using packages of use cases);
- the level of a component is the highest level of any of its constituent classes;
- components should not (and by the above definition, cannot) cross layers[8];

---

[5] This will implement database transaction begin and end commands, and manage the rollback of database changes if necessary.

[6] Since the original draft of this paper was written, Sun have released their EJB product. It is particularly interesting to note that the session bean/entity bean separation in EJBs mirrors the application/domain layer separation in this paper.

[7] Transactions encapsulate a group of related operations which business logic dictates must be either completed in their entirety, or not executed at all. The DBTransaction class is used to encapsulate this type of transaction. If the transaction succeeds, it can then automatically write all modified objects back to an underlying database – hence the need for tabs to be kept on all objects that may be modified.

[8] Here we ignore issues of 'convenience' packaging for customers (e.g. an account component that provides an GUI interface and some business logic may be packaged as a single component on release, following the RREP [Martin96-3]).

- the compile time dependencies between components within any particular layer should be to components in either the same or a lower layer;
- the application and domain layers should be technology free in the interface components within them present to the outside world;

3.4. Layering Semantics

Most layering diagrams omit to discuss the meaning of one layer appearing on top of another, or any description of the axis of the diagram. Earlier reviews of this paper, and the example shown in figure 1 have lead us to believe some further discussion of these aspects of the layering model presented here is desirable:

- *Vertical axis semantics*. The vertical axis of figure 5 indicates the *specificity* (how specific it is to a particular application/environment) of a component in the application. The higher it appears in the layering of the reference model, the more specific it is. The lower it appears, the more general purpose it is. This has lead us to coin the phrase *the centre of gravity of the application* - essentially a way of classifying the overall architectural feel of a system as being either 'high' (very application specific, difficult to extend without substantial modification to existing components), or 'low' (good layering applied, likely to have hooks for extension without any modification to existing components).
- *Layer ordering (highest to lowest) is based on component compile time dependencies.* In figure 1, Szyperski shows a layering model in which the device driver layer is shown below the O/S layer. Whilst this seems appropriate at first glance, a deeper examination reveals the ordering in Szyperski's example is not based on the same criteria as the layering presented in our model. In our model, layer ordering is based on the compile time dependencies between the components that reside within the layer. In the terms presented in this paper, the device driver interface of an operating system is an *extension point* to enable customisation of the operating system "component" to a piece of particular hardware. The operating system is *more* generic (general purpose) than the device drivers it uses (which are tied to particular hardware). The device drivers are also dependent *upon* the operating system for their definition – their interface must conform to the calling interface used by the operating system (they will use the function prototypes defined in an operating system header file) – not the other way round. For this reason, we would present the middle three layers of figure 1 in the following manner (with additional detail to show interface definitions and instantiation of interface)[9]:

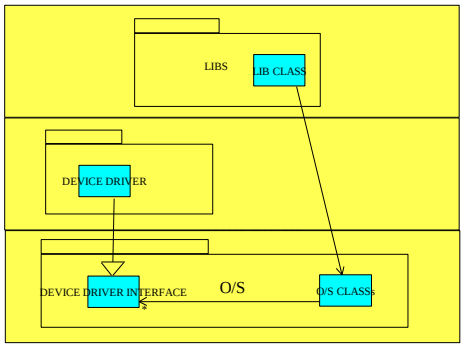

Figure 6 – Szyperski's example using our layering rules

Summarising, the layering semantics presented here tie together the concept of the specificity of a component (how much detail is filled in, how specific it is) with the notion of compile time

[9] Note, however, that using our classification scheme, the bottom two layers of figure 7 would both reside at the platform layer – see section 7 for further discussion of this. Note also, in a non-OO operating system, pointers to functions would be used instead of abstract interface instantiation. The dependency implications of this are, however, identical.

dependencies. The higher a component in the model, the more specific it is likely to be, and the more dependent it is likely to be on other components, and *vice versa*.

3.5. Further notes

A number of additional points are worthy of brief discussion:

- *A component oriented approach*. We have defined our view of architecture as one being based on the structural inter-relationships between the binary components that are used to make up a system. We adopt this focus because at the end of the software development process we would like to have a number of well-defined and well-structured, loosely coupled, internally cohesive binary components that we may, without modification, use again in extending the current application (or possibly another application).
- *GUI components.* All GUI components do *not* reside at the interface level. The interface level may contain application specific refinements of general purpose GUI components (e.g. an application specific dialog box), however the *general purpose* elements themselves (e.g. the generic dialog box from which the application specific one is derived) live at the platform level. The same is true for any general purpose GUI component without application specific customisation (e.g. a graph drawing widget).
- *Substitutability*. Many discussions of architectural layering focus on being able to replace one *whole layer* with another – they are effectively treating the whole layer as a single component. This is *not* the purpose of the model presented here, which is intended as a guide to determining the contents of a particular component by deciding upon within which layer it is appropriate it reside. Substitution would take place at the individual component level, not on a per layer basis.
- *Three-tier architecture.* It is interesting to see how the model presented here maps to the classical view of a three-tier architecture (presentation, business logic, database). The model presented here can be viewed at a conceptual level as being independent of detailed deployment issues. However, it can also be used for applications deployed across multiple machines/processes – for example as in the classic three-tier model. In this case:
  - the *presentation tier* would contain the interface layer, (possibly) the application layer (in a thick client configuration), and some components of the platform layer (e.g. CORBA client components, generic GUI components).
  - the *business logic tier* would (possibly) contain the application layer (in a thin client configuration), the domain layer, the infrastructure layer, and some components of the platform layer (e.g. distributed ODBC client components , CORBA server components), and
  - the *database* tier would contain the remainder of the platform components (e.g. the RDBMS, ODBC server components, etc.);

  the net result being that the higher layers are deployed in one particular machines/process, and that the lower two layers are often present on multiple processes/machines.

4. How the layering helps us understand design

To see how layers and particularly our visual metaphor help us, let's examine the Adapter pattern from the Gang of Four's book [Gamma+95].

Figure 7 shows an adapter being used to allow two components with incompatible interfaces to be used together (a common problem in component design). The billing adapter implements the Account component's outward billing interface and passes on any messages to the credit card billing component's inward billing interface, with possibly modified parameters. Since both of the components are reusable and not specific to this application they belong in the domain layer.

The adapter, on the other hand, is very specific to this particular configuration and so it is not in general reusable and so belongs in the application layer instead – it is acting as application-level "glue" for the other components. Note here that the two component dependencies point downwards – one being caused by an inheritance relationship, the other by a directed association.

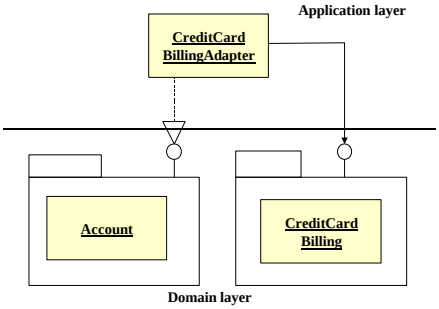

Figure 7 - Getting an account paid by credit card

For a second example, we show in figure 8 one of the refactoring patterns from Martin Fowler's book [Fowler+99]: Separate Domain from Presentation.

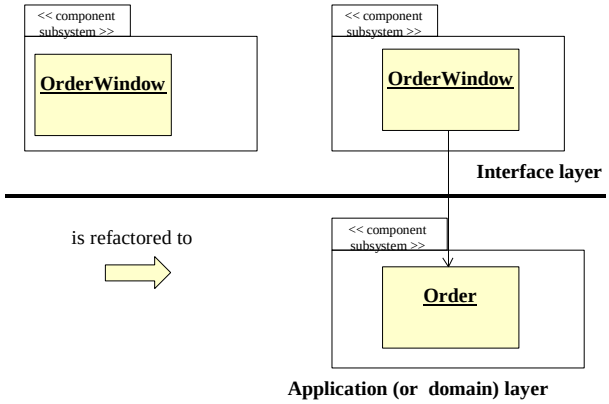

Figure 8 - Separate Domain from Presentation

Here we see a GUI dialog class containing business logic being split into two. Our architectural overlays add context to this, showing that in doing so what was previously an interface layer component has now been split into two components: one still in the interface layer, and one in the application specific (or possibly domain) layer. The refactoring visibly lowers the centre of gravity of the application. Two more benefits are that we have separated the usage of the Order component from its implementation (in other words we have separated policy from mechanism), and that we have separated the technology-free Order component from the inherently technology-based OrderWindow, thereby giving us more portable code and allowing us more freedom in deployment (for example in a three-tier system).

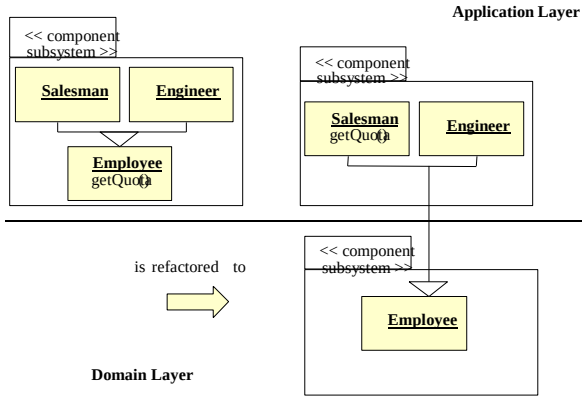

Figure 9 - Push Down Method (reworked by us as Pull Up Method!)

In figure 9, Fowler shows us an inheritance hierarchy with an inappropriately placed method, which is effectively *polluting* the component which contains it by forcing it up to an inappropriate level. The refactored version shows the component being split into two (over two levels), the method having being moved out the Employee class and into the Salesman class. This enables the Employee component to reside at a more general level in the hierarchy, and again visibly lowers the centre of gravity of the application.

5. Broader issues

- *No name, no discussion*
  Nowadays we talk about composites and flyweights [Gamma+95], but this would not have been possible ten years ago. Having a standard layering vocabulary would be of clear benefit in enabling developers to discuss the level at which components might lie.

- *No architecture, no re-use*
  An architectural framework is key to achieving re-use [Jacobson+97]. Re-use requires a clear understanding of what is specific and what is general in designing a set of related components. The reference model assists in understanding these separations, whilst also adding clarity to the layering semantics.

- *Architecture adds context*
  An architectural view adds additional context to many design patterns. Take the observer pattern [Gamma+95]. If packaged together in a single component, the base classes Subject and Observer can be seen as providing a low level (infrastructural) flexible component from which higher level (domain, application or interface) concrete observation mechanisms can be built. The derived ConcreteSubject and ConcreteObserver can be seen as application, domain or interface specific extensions to a general purpose mechanism.

- *Architecture aids (good) design*
  The layering presented also supports a number of OO design principles: the open closed principle [Meyer97][Martin96] – the idea being that lower layer components are closed against modification, and higher layer components extend the open aspects of them; the stable dependencies principle [Martin97] - the layers are organised based on expected stability of their contents – the lower layer components being more stable; and the acyclic dependency principle[Martin97] - the depend downwards rule supports this principle.

- *Invariants*

One important part of the design of component based systems is the identification and allocatiion of responsibility for maintaining invariants *across* (possibly bought in) components. It may be necessary for two components in the domain layer to maintain an invariant relationship - e.g. customer address must be maintained in both components. The responsibility for maintaining invaritants across components resides in a layer *above* that in which the components reside. So in the case of our customer address invarient, it would be the responsibility of a component within the application layer to ensure it was not violated (perhaps using a change event generated when the customer address was changed in either domain layer component).

6. A brief philosophical aside

Many classification systems are blurred around the edges, and our layer classification is no exception. In his excellent book *Darwin's Dangerous Idea* [Dennet96] Dennet describes how examining the characteristics of a particular species of gull, starting in Britain and moving west to east around the globe, yields a set of gradually evolving changes until, as the loop closes and the examiner returns to Britain, a *different* species of gull is finally found next to the original! Species clearly have blurred edges, so we're in good company!

7. Compromises

The model presented here is not perfect, but a compromise between simplicity and meeting the stated set of objectives. Some potential shortcomings of the model are as follows:

- It would have been superficially pleasing to have a rule that said: *you can only depend on the layer below you*. However, upon deeper consideration we believe that the reason for this is an emphasis in previous discussions of layered architectures on complete substitutability of layers. As discussed in section 3.5 (*substitutability*) this is not the motivation behind the model presented here.
- There are clearly times when there will be sub-layerings within the layers presented, and we could have made these explicit. However, we feel the price would have been too high in terms of additional complexity. Instead, we prefer to allow the option of discussing the 'lower application' layer to resolve such issues.
- We have chosen to separate the infrastructure and platform layers based on a buy versus build criteria. Architectural purists may object to this - why should the layer in which a particular component resides be dependent on whether you buy or build it? Our motivation is simple: we wanted to put the focus clearly on the *aspects of the application being developed*. For similar reasons we have been unconcerned with sub-layerings within the platform layer.

8. Conclusions

Summarising, in this paper we have proposed a simple five layered reference model and a number of associated rules to assist the software designer. We have noted that, by convention, *UML class diagrams are upside down*, at least when considered in parallel with architectural layering conventions, and that this is a block to visualising one aspect of what happens during refactoring. We have shown that once this is addressed, UML and the architectural model complement and re-enforce each other.

We have identified examples from Gamma et al's Design Patterns book, and Fowler's refactoring book that show the reference model adds context to well known design (and refactoring) paradigms.

We have discussed how the model supports good OO design principles, in particular those concerned with ensuring stable dependency management, and have emphasised and clarified the rules on which our layering model is based (specificity/generality and compile time dependency).

Coming back to the objectives detailed in section 3.1, we believe the architectural reference model presented here:

- *provides a framework for decision making during component design* by providing a number of layers within which the developer can position their components,
- *supports and re-enforces the appropriate application of good OO design principles*, by encouraging components to be extended (customised) by other components in higher layers, and by imposing a downwards only dependency rule,
- *encourages re-use* by providing a layering system and associated set of rules that puts the focus of design on the specificity/generality of components, encouraging components contents to be seperated on the basis of the layering provided,
- *provides clarification on the meaning of layering in a component context*, by putting the emphasis on compile time dependency management,
- *encourages re-use of business specific (not just technology) components* by presenting two layers within which business components may reside. This encourages more generic functionality to be in the domain layer, and application specific customisation/glue type functionity to be in the application layer, and
- *positions components as the unifying concept that tie together different architectural views of a system*, by stating that the package structure of the system (and associated specification, design or use case views) should be based on the target component structure.

9. References and credits

[Szperski98] Clemens Szyperski, Component Software: Beyond Object-Oriented Programming, January, Addison Wesley Longman, 1998.

[Carlson99] Brent Carlson, Design Patterns for Business Applications, the IBM SanFransisco Approach, ObjectiveView Issue 3, available at www.ratio.co.uk/ObjectiveView.htm, 1999.

[Collins-Cope+00] The Topsy Turvy World of UML, Hubert Matthews and Mark Colllins-Cope, ObjectiveView Issue 4, available at www.ratio.co.uk/ObjectiveView.htm, 2000.

[Jacobson+94] Ivar Jacobson, Magnus Christerson, Partrik Jonsson, Gunnar Övergaard, Object Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1994.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Fowler98] Martin Fowler, Analysis Patterns - Reusable Object Models, Addison Wesley, 1998.

[Jacobson+97] Ivar Jacobson, Martin Griss, Patrik Johsson, Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley Longman, 1997.

[Meyer97] Bertrand Meyer, Object Oriented Software Engineering (second edition) - Prentice Hall Professional Technical Reference, Published 1997.

[Martin96] Robert C. Martin, The Open Closed Principle, C++ Report, Jan 1996.

[Fowler+99] Martin Fowler with contributions from Kent Beck, John Brant, William Opdyke, and Don Roberts, Refactoring - Improving the Design of Existing Code, Addison Wesley, 1999.

[Martin97] Robert C. Martin, Stability, C++ Report, Feb 1997.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Dennet96] Daniel C. Dennet, Darwin's Dangerous Idea: Evolution and the Meanings of Life, Touchstone Books, 1996.

Particular thanks are due to Andy Vautier, Nigel Barnes and Keith Haviland of Andersen Consulting, upon whose 1 million line+ C++ project many of the underlying concepts presented in this paper were formulated. Further detailed technical discussion this project can be found at www.ratio.co.uk/techlibrary.html.