Published in the Proceedings of The 6th International Workshop on Organisational Semiotics: Virtual, Distributed and Flexible Organisations, 11-12 July 2003, Reading, UK, 317-330.

Facilitating Useful Object-Orientation for Virtual eBusiness Organisations Though Semiotic

Dr Simon Polovina
Senior Lecturer
Computing and Management Sciences
Sheffield Hallam University, UK
Email: S.Polovina@shu.ac.uk

Duncan Strang
Java Consultant
Email: dls@nearplanet.com

Abstract

Ultimately the virtual, distributed and flexible eBusiness organisation will be characterised by its participants (stakeholders) not having any direct human-human contact but simply relying on computer-mediated interaction for the organisation to function successfully. The computer system's representation must thus be valid (to avoid doubt and loss of faith) and unequivocal (so that the medium itself does not add ambiguities of its own). Regardless of any intelligent systems considerations it is now even more fundamental that human-computer and software engineering hurdles are overcome, hence the true advent of the object-oriented approach (epitomised by the rise of n-Tier enterprise applications). There are however two lingering issues with object-orientation. Firstly it is well known that object-orientation and relational databases do not sit easily together. Secondly the rewriting of an object's class can cause all the other object classes that it interacts with having to be rewritten in turn, with its consequential knock on effects throughout the application. These two issues cause the very purpose of the object-orientated approach to be defeated. For the data vs. objects issue we require a reconciling framework for these two views of the world around us. Peirce's pragmatism suggests that there is a logical, semiotic way that divergences can be reconciled. Accepting this, Peirce's philosophy could be applied to this fundamental object vs. data divide. For the application-wide knock-on-effect issue the object's evolution might be recorded by means of its 'palimpsest' i.e. embedded yet visible layers denoting that object's temporal development within one, humanly manageable, entity. The other objects can then interact with the appropriate previous version of that object thus avoiding a potential rewrite of the whole application. Moreover the object's evolution would be captured, thus offering a valuable experiential record for future information systems development. Through its unifying framework, semiotic offers the optimal way to so record the objects' evolution and, with reconciling the data vs. object divide, enable software development to model the eBusiness world at last.

1 Introduction

Computerisation, the Internet and the World Wide Web are increasingly affecting how we organise our personal and business lives. We regularly rely on networked computerised devices (e.g. Personal Desktop Computers, Laptop Computers, PDAs, Mobile Phones, or some other present or future computer-mediating device) to undertake even the simplest human-human interaction. This phenomenon is evidenced by the 'e' (for 'electronic') prefix attached to more and more of our daily activities (eMail, eDiscussions, eFriendships, eLearning, eCommerce, eBusiness, eEverything…)[1]. The ultimate realisation of this trend of course is that we no longer conduct our interactions other than via a computer-mediated device. Our meetings thus become 'virtual' rather than 'physical'.

While the e-phenomenon seems to have touched upon most sections of society in the developed world one of the most enthusiastic user groups has been the world wide business community [1]. It is certainly the case that the trend is for organisations to 'get connected' and use diverse networked devices to free them from physical geographic limits in terms of their operational activities and the markets they serve [2]. Thus, for example, an eBusiness engaged in producing specialist Scandinavian office furniture may be able to manufacture

[1] Indeed so much so that we end up losing the e prefix and refer back to the 'original' name (e.g. 'eBusiness' becomes 'Business' again) as we take the e phenomenon for granted.

its wares in Sweden whilst retaining its best designer who gets his inspiration by living in a remote part of Greenland (where its registered office is for cultural and tax reasons).

The organisation can promote and sell its furniture world-wide paying particular attention to South America as this market culturally identifies with these furniture designs (therefore making this business profitable!). It can maintain its distribution hub in the USA whilst allowing it's directors to live in Copenhagen, Stockholm, Oslo, New York, Tokyo and Buenos Aires. Without the e factor, this organisation would have to invest much more heavily in physical infrastructure (e.g. foreign offices and staff), and regularly fly the directors, the designer and other key staff to one physical meeting location (assuming, aside from disrupting the quality of their lives, they can agree dates when they are all free). Faced with these constraints the business would be uneconomic, and the products it provides lost to its customers. It would remain undeveloped with its consequential losses to the economies in which it operates and its stakeholders (e.g. customers, employees, shareholders, suppliers, government tax revenue and the state's economic development). It either exists as a virtual organisation or not at all.

Given that our example organisation (and others like it) would not exist without the ability to communicate in the ways described, there is an expanding need for computer systems to manage these communications. These eBusiness systems are by choice often designed and implemented using object-oriented techniques and programming languages [3]. Prior to the widespread utilisation of these languages, most business systems were restricted to the internal office environment and were based on the RDBMS (Relational Database Management System) paradigm. As this paper will highlight, these two paradigms are not immediately compatible. Indeed as we shall see the problem is not just technical but cultural: religious divides have sprung up, undermining progress on unifying these approaches so that eBusiness systems can make use of the undoubted benefits each approach provides.

A further obstacle to the development and maintenance of object-oriented eBusiness systems is what this paper will later explain as the problem of 'message litter'. Systems must continually adapt in the light of better understandings of business problems and changes in the business environment whilst having to interoperate with other systems new and old, good and bad (and more towards the latter) on which businesses are nonetheless obliged to rely upon [4].

It is therefore these two key obstacles that this paper will advocate may usefully be addressed by *semiotic*, the science of signs and meaning making, enabling the virtual eBusiness organisation to be realised. We begin by discussing certain background issues that lead to these obstacles.

2 Quality of the Computer-mediated Representation

As the idealised virtual, distributed and flexible eBusiness organisation is such that they will lack any human-human interface, its modus operendi is the *human-computer* interace. Clearly this places the onus on the quality of this human-computer interaction, as these organisations *must* rely on these computer-mediated representations amongst its stakeholders as being credible. They must thus be valid (to avoid doubt and loss of faith) and unequivocal (so that the medium itself does not add ambiguities of its own).

To achieve this aim much current work in engaged in developing intelligent systems for the eBusiness, and this is rightly so [5]. Its success will provide the eBusiness with the ability to capture, store and manipulate knowledge to achieve its objectives (nowadays

commonly referred to as the overarching term of its 'mission'). A system that from its computerised knowledge-base can present its users with the right, relevant information at the right time and place, and be able to answer adequately the fuzzy ('you know what I mean') querying we as humans take for granted would certainly be virtual nirvana. But we need to get the basics right first so that such systems can build upon them, so it is now even more fundamental that human-computer and software engineering hurdles are accordingly overcome. Addressing the problems of information systems (IS) failures through software engineering has lead to the true advent of object-oriented approaches and, as objects are human-centred ways of developing systems, addresses issues for human-computer interaction too.

3 Why OO, Why n-Tier

To understand why object-orientation (OO) has gone a long way to solving the problems of software engineering and how it might tackle human-computer interaction concerns, we need to revisit OO concepts. We also need to understand why modern enterprise-level[2] applications are horizontally tiered ('n-Tier') and can be further levered to our advantage (all of which will be explained later).

3.1 The four Software Engineering goals

Beginning with OO concepts, we return to the *four goals of software engineering*, namely:

1. **Abstraction**. In order to try to understand some problem in the world around us it is often useful to sift out immediately irrelevant details so that we might concentrate on the core of the problem. This technique is called abstraction. We say that we abstract the relevant parts from that problem so we might focus all our attention on them. In this way we incrementally add to our classification of knowledge. (We classify things to provide our predictive framework, like Chemists do with the Classification of the Elements – the 'Periodic Table', or Credit Agencies with Credit Ratings e.g. AA rating. After all, how else would be understand the world around us?) As a simple example we might realise that a payroll system needs to deal with our employees, consequently we might come up with an `Employee` object to represent our employees in the system. We can now think about how an `Employee` moves about the system without worrying about how an `Employee` might actually be implemented or even what data we might need to identify our employee. We have abstracted out the details of an `Employee` into a convenient representation so that we can avoid having to deal with all the clutter of detail associated with the concept every time we need to be aware of it.
2. **Cohesion**. At some level of abstraction what we actually end up with are potential objects in the solution space. We now need to be careful to maintain our abstraction when we come to focus on the detail. For example, what are the attributes of our Employee object? We might decide that our `Employee` contains items like `'Name'` `'Address'` and `'Payroll-Number'`. However, would we expect `'Heartbeat-Rate'` to be there? If we were developing some aspect of a medical system then quite possibly yes, but in any event it has nothing to do with an employee! What about `'Company-Car'`? Should our `Employee` have to concern itself with, say, working out the tax rates for a company car? No, but in trying to add program

[2] i.e. larger-scale applications that can cover all the activities of a whole enterprise (e.g. Shell, Amazon.com etc., as well as many SME – *s*mall to *m*edium sized *e*nterprises). This is as opposed to small, individual-scale applications such as the writing of some macro programming code in a spreadsheet application, or a personal user's or small office database written in Microsoft Access™.

code for employees who have company cars can too easily leads to code also being added to calculate 'seemingly related' items like tax rates, and so on and so on. (Maybe the heartbeat rate gets to be included after all??) This one small example highlights a major cause of software project (thus IS) failure: it is all too easy to fall into the "it's only one more line of code" state of mind that inevitably leads to a huge amorphous chunk of code that is impossible to modify and de-bug. From our abstraction we thus derive objects that should be richly coherent in themselves i.e. an object, simply put, 'does its one thing well'[3]. We therefore seek *high* cohesion.

3. **Coupling**. As we seek to design our objects so that each one 'does its one thing well' thereby being highly cohesive, we also require that the number of links between the components in a software system should be as few as possible i.e. *low* (or loose) coupling. Why? Because the more linkages (interfaces) there are between objects in a system the greater the interdependence between each of them. Consequently if one object's interface is altered this could easily have a knock on effect throughout the whole system – the 'ripple' effect. This once again leads to the same major cause of software project failure; one huge amorphous chunk of endless lines of incomprehensible, un-maintainable, code.
4. **Modularity**. This is simply the breaking down of the problem domain into manageable chunks – the 'divide and conquer' principle found in all human endeavours. Then we can concentrate on each of these elements. Modularity, if conducted effectively, inherently supports good abstraction, high cohesion and low coupling.

Miller's classic study from psychology shows that the human mind can handle only 7±2 concepts at a time thus truly underpinning the whole software engineering endeavour of modularity, low coupling, high cohesion, and abstraction [6]. Consider for instance the amount of couplings that an object can offer. Clearly if has many hundreds, aside from the evidently high coupling danger it would be well over human comprehension as to which one to select etc. as that's way above the '7' limit. The least amount could of course be 1, as 0 (zero) would mean that object would be unable to make itself aware to any other thus be useless! What's particularly significant about the Miller study is that it arises from psychology, not 'techie' computer science. This recognition on the part of software engineering fundamentally supports the human-centric nature of OO, which consequently came about to realise the four software engineering goals in enterprise-level software development.

3.2 The three OO concepts

Accordingly, contemporary OO programming languages such as Java and C# have built into their very fabric the practical constructs to enable this, as does the information systems' analysis and design modelling environment UML (Unified Modeling Language, http://www.uml.org/). How is this actually attained? That is where the *three OO concepts* come into play:

1. **Encapsulation**. This is the practice of including in an object everything it needs [7]. Everything an object needs includes data and operations on that data. Most importantly,

[3] To illustrate further, this is like the game played with very young children where we ask them, for example, what does the tiger do? Answer: Roar. What does the cat do? Meow. What does the snake do? Hiss. And so on. If the cat, say, barked too it would be not be cohesive with our expectations of such an animal, so it's at best unusual (maybe even undiscovered?) but, far more likely, wrong so we'd correct the child accordingly. Similarly we expect a Video recorder to play and record videos i.e. it should do its one thing well – video operations, not make toast.

the data is *only* accessible through those operations that are made visible to ‘outside’ users. The actual data (or state) of an object is hidden away inside the object. Encapsulation facilitates the change and modification of the internal workings of an object without affecting any current users of that object. The practice of hiding the information away behind a published interface is known as ‘information hiding’[4] and is essentially denoted by the keywords ‘private’ and ‘public’. These do not feature in pre-OO languages like C[5]. Those data or operations items that the software designer or developer prefixes with `private` cannot be accessed from outside the object, only from inside that object itself. This therefore reduces the amount of potential couplings – i.e. low coupling – and raises cohesion as it forces the rich complex activity to be dealt with away from its users (other objects).

An analogical example is the Play button on a video machine. The user need not need be concerned with the complex electronic and mechanical operations the machine has to undertake to play a tape; it is the machine’s job to know how to play a tape, not the user’s i.e. delegation. And it safeguards the user from making a mess of things by inadvertently enacting procedures (operations) that user doesn’t understand – the familiar ‘No user-serviceable parts inside’. From a software development point of view it dissuades ‘hacking’- one cannot so easily ‘just add one line of code’ without considering these issues as encapsulation is now ‘in the face’ of the designer/developer. (All this of course satisfies Miller too.)

Our `Employee` object might include an `Age` data item. We would want to keep this private as we would not want other objects ‘hacking’ this field inputting nonsense values like ‘`-10000`’. We would want it to be ‘managed’ so that its values lie, say, in between 16 to 65 (or 60 for women) reflecting current UK legislation. And of course it should allow for those employees who want to continue to work after retirement age. We would therefore only have a public `setAge(age-value)` operation (or method in OO parlance) available and it would in turn privately call the `Age` field, as well as the object’s other private inner workings to conduct its proper management as suggested. As before it’s delegated to the `Employee` object to deal with this (as it’s its job not the user’s job) so it does its one thing (Employee-type operations) well.

2. **Inheritance**. So far in this discussion we have simply referred to objects, and mentioned classification as the way we can categorise elements of our knowledge. In OO we distinguish between objects and classes. The class is rather like a pastry cutter; the objects are the actual pastries it creates. So when we refer to Employee this is really a class; an object would be a particular instance of this class. Accordingly “`Fred Smith`” would be an object of the `Employee` class. Classes are particularly significant for inheritance, in which each class is arranged in a *hierarchy* of classes, with the most general at the root (top) and its specialisations as its leaves (down from the root to the bottom). So for instance our most general class (superclass) might be `Employee`, and its specialisations (subclasses) might be `Sales-Employee`, `Senior-Manager`, etc. Similarly it is possible for, say, `Chief-Executive` to be a subclass of its superclass `Senior-Manager`, which in turn is a subclass of

[4] And/or ‘data hiding’.

[5] In this regard C++ is a strange beast in that it allows both OO and pre-OO software development (effectively C). As such it cannot enforce an OO approach e.g. by essentially requiring that all program code is contained within a class.

`Employee`. As class relationships are transitive, we can also state that `Chief-Executive` is a subclass of `Employee`.

Inheritance takes advantage of this 'is-a' relationship, which works as indicated above. Accordingly we can place data fields and operations in a superclass and all its subclasses will inherit (hence the term) these data and operations, as long as they are public of course[6], plus any of its own. So `Employee` might have a public `setName(…)`, `getName()`, `setAge(…)`, `getAge()` etc., which `Chief-Executive` will thus also have as well as adding its own e.g. `calculateBonus()`. Inheritance enhances modularity and abstraction as it adds the sub-dividing dimension of levels of generalised or specialised abstractions, and their interrelationships, to be considered. Inheritance improves cohesion as it focuses the purpose of each class, but it worsens coupling as interdependence has increased between classes (hence its objects) in the hierarchy. So, for example, if a change was made to the public interface of `Employee` e.g. an extra parameter was added to the setAge method like `setAge(age-value, yes-no-retired)` this would ripple down to all its subclasses, meaning any object calling this method would have to be modified, unravelling all the software engineering goals already discussed.

3. **Polymorphism**. Inheritance thus is an optimising trade off, and should thereby be used judiciously. However we can take advantage of the interdependency for the purpose of polymorphism (taken from Greek meaning 'many forms')[7]. In the context of this discussion, polymorphism allows any subclass to be substituted for its superclass, after Liskov's Substitutability Principle [8]. Accordingly `Chief-Executive` can be substituted for `Senior-Manager` or `Employee`, but `Senior-Manager` or `Employee` cannot be substituted for `Chief-Executive`. This makes sense[8].

   A superclass can be designed / programmed so that it actually has no implementation of its data or operations. It merely provides the names (or 'data/method signatures') by which these items are publicly referred to e.g. `setAge(…)`. These classes are known as *interfaces* (perhaps not surprisingly)[9]. As they have no implementation details, objects cannot be created from them – they are thus indeed abstract! Other objects may call an interface but one of its *concrete* subclasses (i.e. class that has implementation thus can create objects) must be substituted to run the implemented code. As the interface has no implementation the concrete subclass must implement *all* the interface's signatures. A simple example is `Shape`, which has a `calculateArea(…)` method. A shape's area cannot be calculated unless we know what kind of shape it is – Shape is truly abstract! So `Circle`, `Triangle`, `Rectangle`, `Square`, etc. objects would have to be substituted and have to match this method signature and give its particular implementation according to the shape in

---

[6] There are visibility modifiers other than `public` or `private` (e.g. `protected`) that for the sake of brevity are not included in this paper.

[7] Strictly speaking we are referring to parametric polymorphism, as there are other forms of polymorphism such as 'operator overloading' that are not discussed in this paper.

[8] Consider as an illustration that if you were to ask for an Employee you should be happy to see the Chief Executive, the Sales Employee, the Office Assistant or indeed the Janitor. These are, after all, all employees. On the other hand if you asked for a Senior Manager then (in our simple inheritance hierarchy) then you would be happy with the Chief Executive but angry if you saw the Janitor (no slur on this noble profession intended!)

[9] There is also the concept of the 'abstract class' that, again for the sake of simple clarity and brevity is not distinguished in this paper.

question. The `Employee` class could instead be an interface (as indeed could its subclass `Senior-Manager` to include a hierarchy of interfaces if so wished), so that all its objects would have to be one of its concrete subclasses e.g. `Chief Executive`. The concept is essentially plug-and-play; printer drives are an evident example as Microsoft for instance would not want to rewrite Windows™ every single second a new printer is produced!

Polymorphism allows so-called 'late binding' at runtime. Code does not have to be rewritten (with all the horrors that can bring as we have seen) for an updated situation e.g. a new printer driver or a new class of employee. We should therefore always follow the "principle of reusable object-oriented design: Program to an interface, not an implementation." [9]. As we are levering increased coupling to our advantage by focusing on the interface, this refines abstraction to its truest, abstract sense in achieving the optimum structure for the interfaces.

3.3 Layering, N-Tier

3.3.1 Content vs. Presentation

The world consists of two key elements. The tangible objects of which it is made up (e.g. Land, Sea, Ballet, Business, Employee, Bank-Account, …), and the intangible information flows that, indeed, informs us of these objects and their significance (e.g. 'That performance of Nutcracker did a brilliant service to ballet because…'). To convey these messages correctly therefore, information has to be represented truthfully, succinctly and unambiguously. This signification process is the domain of *semiotic*[10], the science of signs and meaning making. Information can thus be separated into two components or 'layers' accordingly:

- The actual content of that information
- The way that information is presented

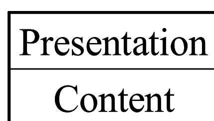

Figure 1

As shown in Figure 1 these layers are tiered horizontally, reflecting that they can cover the whole of an organisation's operations as opposed to vertically which focuses on a specific function or product-market (e.g. marketing, food retail.) That way we can divide-and-conquer (i.e. achieve modularity) and cohesively focus on what any item of information is about regardless of how it is presented to the user and vice versa. Once we have cleanly defined the interface between a content class and a presentation class (known as a *boundary* class) we can polymorphically 'plug-and-play' the different user interfaces according to the users needs and abilities, background knowledge, and culture. This defines the *presentation* layer, the *content* is the other layer (made up of *control* or *entity* classes as explained shortly), hence 'content vs. presentation', and these layers communicate via a simple (to maintain), well defined interface.

To appreciate how to make this separation, imagine how the *same* piece of information might be conveyed to:

- A blind user
- A deaf user

[10] Or semiotics or semiology. In its application (on which this paper focuses) the arguably detracting philosophical debate surrounding any differences in semiotic terminology is not entered into; they are treated interchangeably for the purposes of this paper.

For instance, take a favourite music track or artist, and consider how this would be explained to deaf or otherwise hearing impaired people so that they can fully experience it as if they had full hearing. Likewise take a favourite picture and convey this fully to blind or otherwise visually impaired people. Clearly this level of accessibility is an ambitious challenge[11], but the true seperation of presentation from content is what contemporary enterprise systems development seeks for the reasons already given. This is further evidenced in that one of the tenets of XML (eXtensible Markup Language, http://www.w3.org/XML/) is that information content is separated out, and XSL-FO (eXtensible Stylesheet Language – Formatting Objects, http://www.w3.org/Style/XSL/), a part of XML that is used to format (i.e. present) that information. The accessibility effort in this direction is reflected in the XML Accessibility Guidelines (http://www.w3.org/TR/xag). There are more modest but immediately practical attempts such as the use of XSL-FO to create printable documents [10]. There is accordingly an unmistakable call for semiotic to address this separation and to offer, from its inherently relevant theoretical basis, a meaningful way towards its solution. This is of course an intrinsic component of this paper's subject matter and opens up a specific thread of highly worthwhile research. Indeed work has begun on this very topic [11]. That work is thus presently left for elsewhere and we remain focused on our two primary concerns (i.e. 'database vs. OO', 'message litter'), whilst acknowledging that research's successful outcome would undoubtedly augment the material of this paper.

3.3.2 Business Logic vs. Persistence

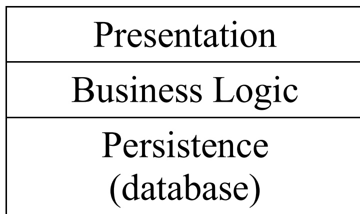

Figure 2

Information content can be further separated into two 2 intrinsic components. One of these is those classes (the *control* classes) that encode the dynamic business processes or operations i.e. the *business logic*. The other is the *c*reated, *r*etrieved, *u*pdated or *d*eleted ('crud') information that that business logic processes and, from this processing, stores (persists) accordingly. This is the *persistent store* (the *persistence* layer), and characterised by the term *entity* classes[12]. There are thus three layers, illustrated by Figure 2. As the persisted information is commonly handled by a Database (DB), with file handling managed by a Database Management System (DBMS) of which the Relational DBMS (RDBMS) is the prevalent approach the bottom tier in Figure 2 is illustrated as the database. Many systems have more layers essentially by breaking these three down further, giving rise to the term 'n-Tier', but the three as given by Figure 2 embody the essential tiers. This discussion therefore centres on this basic 3-Tier architecture.

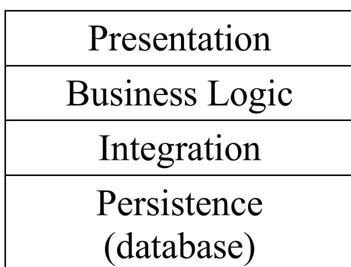

Figure 3

There is however also a layer in between the persistence and business logic known as *integration*, which addresses the actual mapping between the two thus giving a 4-tier architecture, illustrated by Figure 3. This does not detract from the concept of 3 basic layers as the integration layer essentially translates the way that the other two can communicate to one another, but as we shall see

[11] The W3C's Web Accessibility Initiative (http://www.w3.org/WAI/) has nonetheless been addressing this subject from the outset.
[12] The terms boundary, control and entity classes are defined in Jacobson et al. [26]. Bennett et al. discusses these terms as well as layering, persistence and data management [15].

from the following discussion the role of the integration layer may well be particularly significant.

4 Lingering Issues

As already indicated, there remain two lingering issues with object-orientation. Firstly it is well known that object-orientation and relational databases do not sit easily together [12]. Secondly the rewriting of an object's class can result in all the other object classes that it interacts with having to be rewritten in turn, with its consequential knock on effects throughout the application. These two issues cause the very purpose of the object-orientated approach to be defeated. Why is this so?

4.1 Data vs. Objects

When considering objects, and in particular encapsulation, we saw that a key aspect of their success was that data and information was hidden to support effective software engineering (and, hence, usability). If in 3rd tier the persisted data is to be handled by a RDBMS then the object to which is related has to have its data divorced from its operations. This is the 'nasty' way of separation: encapsulation is lost, cohesion is lost and coupling has to increase! Once again we risk too easily returning to IS failure due to inadequate software. We return to this issue later, after introducing the second issue.

4.2 Refactoring

As an object's operations or data can be hidden (encapsulated), internal improvements can be made to the design of that object's class to enhance the application without causing a knock on effect by other objects having to be rewritten. As far as those other objects (hence application) are concerned there's been no change, so we do not have to rewrite, re-bug and re-test the whole application with its consequential (over)massive overhead for potentially a small change, with its consequent software failure.

This is because the other objects are never aware to the change (thus cannot tinker with it); the enhanced object's public method signature remains the same. So, for example, the internal processing of the `setAge(…)` method of the `Employee` class could be enhanced so that it handles changes in retirement age law. The calling object could still be informed if the change is valid or not, it just happens to be assessed differently by the `Employee` object.

This technique is known as *refactoring* (http://www.refactoring.com/). Although it is now practical to refactor the internal, encapsulated components of a class, alterations to its external public interface remains unworkable as it *will* affect all the other objects it interacts with. This rewrite of that's object's method signatures thus does increase coupling as all the other objects it relates to have to reworked with its knock on effects throughout the application. The spectre of IS failure re-emerges.

The traditional approach to dealing with this issue has ended up creating 'object litter' by having to spawn numerous new (sub)classes that allow the original (imperfect) object to be used, but add to the information overload as the application's class library size goes out of all humanly manageable recognition well over Miller's 7±2 limit referred to earlier [6]. As these 'new' objects inevitably require new message signatures at their public interface, object litter in turn spawns even more 'message litter'. All in all modularity is thereby lost as indeed seen in practice (e.g. the latest, even huger Java Class Library itself which now

contains 3,151 classes quite aside from the increase in the number of new message signatures![13]).

5 Reconciling Data and Objects

All in all therefore the very purpose of the object orientated approach is thus defeated. If de-encapsulating the data of the 3rd tier so that it can be stored and manipulated in a DBMS, it might be argued that databases cannot figure in contemporary software development. Indeed alternatives have been developed, such as 'serializing' the objects as in Java to make them persistent [13]. But these approaches do not have the robust, scalable data management capability found in a RDBMS (nor might they ever catch up?) Moreover however RDBMS are so prevalent and well-known that it would be a very brave organisation that would risk its business on an 'unproven' alternative (and certainly an analyst/designer/developer would not want to risk his/her career accordingly!) Even mighty Microsoft in their latest .NET technology (http://www.microsoft.com/net/) have stepped back from this issue, as indicated by Fowler's discussion around his 'table module' vs. 'domain model' pattern [14][14].

To address this issue, object-oriented DBMS (OODBMS) such as Caché ("the post-relational database for e-applications", http://www.e-dbms.com/) have emerged. But as Bennett et al. (page 464) note, even these technologies do not re-encapsulate the operations [15] – there is thus no remarriage here! We can therefore take the view that applications that strongly database oriented are considered unsuitable for object-oriented development as Bennett et al. typify but, more realistically as we have seen and as they also accept, we 'have to' address how to map OO onto RDBMS [15]. For the data vs. objects issue we therefore require a reconciling framework for these two views of the world around us. What will the nature of this be?

5.1 Hybrid possibilities

In attempting to solve this obstacle in practice, hybrid-type solutions such as Enterprise Java Beans (EJB, also known as J2EE, http://java.sun.com/j2ee/), of which Raj provides an elegant illustration, have been developed [16]. Though these approaches do go some way to alleviating the difficulty the reconciliation they ultimately offer is more of a resignation to the apparent dichotomy. For instance EJB's 'entity beans' are arguably little more than shallow objects representing a connection to a database (albeit a rich one)[15]. There is thus no encapsulation; it's possible for private data changes to be made outside the control of the object. It's back to square one.

---

[13] This figure is arrived at by simply adding up all the classes shown in the left hand frame of http://java.sun.com/j2se/1.4.1/docs/api/index.html (2,723) plus all those at http://java.sun.com/j2ee/sdk_1.3/techdocs/api/index.html (428) shown similarly. Naturally many of these classes can be clumped together to be considered as part of a whole, as they are into 'packages' (see the links given). Nonetheless their sheer number, and their complexity (many classes have more than 7 public methods to begin with!) make the activity of working with them a mind boggling prospect.

[14] The topic of patterns is not addressed in this paper, but as patterns have a potentially highly relevant role they should certainly feature in future works. Although not explicated, reference has been made to the work on patterns in this paper [9, 14].

[15] Incidentally Fowler (e.g. pages 100-101) goes even further and highlights the potential dubiousness of using EJB / J2EE (and certain 'leading edge' aspects of .NET for that matter), when POJO (plain old Java objects) will suffice and that EJB can serve to hinder rather than help [14] .

5.2 The religious dimension

All in all therefore it's becoming evident that technology cannot conquer this problem. This is somewhat not surprising because the limit is not technological but philosophical. Objects are a human construction of the world, recognising that software development is a human endeavour, typified by the fact that earliest successes have been with human-computer interaction i.e. graphical user interface (GUI) objects epitomised by the Xerox Palo Alto work and the arrival of the Apple Macintosh computer from the 1980's [17]. The database has likewise evolved into a philosophy of the world. The RDBMS's relational algebra is a powerful mathematical model of optimising the relationship between entities, and has been realised in IS development practice by the 'normalisation' process. As already stated many individuals have made their careers, and organisations have placed their expertise, on this paradigm. The prevalent IS view of the world is a data centric one and, as we can see, with good justification.

What we are left with therefore is an elemental reconciliation between the object vs. data views of the world. We have seen that the object-oriented approach is fundamental in addressing IS failure. There is thus no going back. Due to the paradigm shift needed, database professionals cannot relate easily to objects and rightly fear that all the valuable knowledge that RDBMS offer will be lost. Object professionals see databases as an implementation hurdle that contradicts the software engineering goals hence should be discarded. A process of shared understanding must therefore take place to overcome the object vs. data philosophical divide. The task is not trivial. As Amber notes these conflicting views have become so heated that each side's arguments have resulted in a religious divide that exists between database designers and their object-oriented counterparts; the debate has become embedded in divergent cultures [12]. It's therefore manifestly clear that this is the challenge for semiotic, as its success in this key impedance mismatch would indeed be its vindication.

5.3 Semiotic approaches

There are a number of ways that semiotic might help. Peirce's pragmatism suggests that there is a logical, semiotic way that divergences can be reconciled without the need to accept compromises that ultimately satisfy no-one [18]. Accepting this we could apply Peirce's philosophy to the apparently very fundamental divide. Polovina has demonstrated that the semiotically inspired conceptual graphs (being based on Peirce's existential logic, http://www.cs.uah.edu/~delugach/CG/) can bring divergent disciplines like accounting and strategic planning into one unifying, non-compromising framework [19, 20].

The organisational semiotics work that Andersen, Liu, Stamper and others are developing could usefully be applied to this problem [21, 22]. Given this paper's acceptance and airing in that arena, contributions are thus highly welcomed. The practical-oriented Shared Meanings Design Framework (SMDF, http://www.smdf.org/), which focuses on semiotic in the IS requirements and development from a conflicting user/stakeholder perspective offers a particularly attractive route.

How any of these may be appropriated has yet to be determined, but demonstrates that there is a substantial base of semiotic to tackle the worthwhile problems that this paper raises. Mention was made earlier of the integration layer in the n-tier software architecture. It is evident that this is where the effort could be focused, as indicated by the earlier discussion on the shallow-ness illustrated in J2EE's entity beans. Instead of the integration layer representing a compromise 'just to get the thing working' with some nominal deference to n-tiered-ness, semiotic could be applied to this layer so that it recognises and

embodies the cultural divide. It can then be said that it truly offers integration, by each side of the object-data cultural divide engaging in shared meanings as they understand what each side can bring to achieving the ultimate goal of facilitating useful OO for eBusinesses.

6 Palimpsest the Message Litter

For the second, application-wide knock-on-effect issue the object's evolution might be recorded by means of its 'palimpsest' i.e. embedded yet visible layers denoting that object's temporal development within one, humanly manageable, entity. The other objects can then interact with the appropriate previous version of that object thus avoiding a potential rewrite of the whole application. Moreover the object's evolution would be captured, thus offering a valuable experiential record for future information systems development. Accordingly low coupling is retained as each object can interact with each other according to which palimpsest incarnation the objects are aware of; incidentally this would also result in the manifest of a palimpsest for the whole application – a valuable experiential record for future information systems developments! Through its unifying framework, semiotic offers the optimal way to record the objects' evolution [23]. The palimpsest metaphor has been applied in the context of architectural design [24]. Again the semiotic work identified for the object-data divide is pertinent here too.

7 Concluding Remarks

It is evident that for eBusiness to succeed now and in its ultimate sense by enabling completely computer-mediated communication, software engineering and human-computer interaction (HCI) hurdles must be overcome. In its course this paper agrees with Anderson and sees HCI as a facet of IS development [25]; nonetheless by its very nature semiotic does particularly offer a human-centric (thus HCI) perspective thereby further highlighting this key driver. Object-orientation, which is a human-centric interpretation of our world, has helped greatly in this regard, indeed to such a great extent that we need not turn away from this paradigm but build upon it. As this paper has revealed, semiotic can help overcome two fundamental hurdles: the object-data divide and the generation of object / message litter. If it cannot then it is difficult to see what role semiotic has in IS development. As the overwhelming evidence suggests otherwise this paper does not accept that alternative, and it is up to us in the semiotic communities to now demonstrate this. Let's enable software development to satisfy eBusiness at last.

8 References

[1] N. Bandyo-padhyay, *E-commerce : context, concepts and consequences ,* London: McGraw-Hill, 2002.

[2] M. Norris and S. West, *eBusiness Essentials: Technology and Network Requirements for Mobile and Online Markets,* John Wiley and Sons Ltd, 2001.

[3] Y. Shan and R.H. Earle, *Enterprise Computing with Objects: From Client/server Environments to the Internet,* Addison Wesley, 1998.

[4] G. Kunene, "Enterprise Application Integration: The Problem that Won't Go Away". Accessed: 2003, January 29, 2003. 2003. http://www.devx.com/enterprise/Article/10667

[5] D. Parkes, N. Sadeh and W.E. Walsh, "Agent-mediated Electronic Commerce IV. Designing Mechanisms and Systems: AAMAS 2002 Workshop on Agent Mediated Electronic Commerce, Bologna, Italy, July 16, 2002, Revised Papers" pp. 339, December 2002. 2002.

[6] G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review,* vol. 63, pp. 81-97, 1956.

[7] D. Parsons, *Object Oriented Programming with C++ (Second Edition),* Letts educational, 1997.

[8] B. Liskov and J. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems,* vol. 16, pp. 1811-1841, November, 1994. 1994.

[9] E. Gamma, R. Helm, R. Johnson and J. Vissides, *Design Patterns: Micro-architectures for Reusable Object-oriented Software,* Addison-Wesley, 1995.

[10] R.M. Raya, "Using XSL-FO to create printable documents". Accessed: 2003, November 2001. 2001. http://www.ibm.com/developerworks/library/x-xslfo/

[11] T. French and S. Polovina, "xCulture for the eEnterprise: an elementary 4-tier architecture for site localisation," *in The 2nd BCS HCI Group Workshop on "Culture and HCI: Bridging Cultural and Digital Divides",* 2003,

[12] S. Ambler, "Crossing the Object-Data Divide". *Software Development Online* March 2000 and April 2000. 2000. http://www.umlchina.com/CBD/crossing.htm

[13] B.T. Kurotsuchi, "The Wonders of Java Object Serialization". Accessed: 2003, 05 August 2002. 2002. http://www.acm.org/crossroads/xrds4-2/serial.html

[14] M. Fowler, *Patterns of Enterprise Application Architecture,* Addison Wesley, 2003.

[15] S. Bennett, S. McRobb and R. Farmer, *Object-oriented Systems Analysis and Design Using UML (2nd Edition),* McGraw-Hill, 2002.

[16] G. Suresh Raj, "EJB Containers". Accessed: 2003, January 11th 2000. 2000. http://www.idevresource.com/java/library/articles/ejbcontainers.asp

[17] J. Durham, "History-making components: Tracing the roots of components from OOP through WS". Accessed: 2003, April 2001. 2001. http://www-105.ibm.com/developerworks/papers.nsf/dw/java-papers-bytitle

[18] R. Kevelson, *Peirce's Pragmatism: The Medium as Method,* Peter Lang Publishing, 1998.

[19] S. Polovina, "Bridging Accounting and Business Strategic Planning Using Conceptual Graphs," in *Conceptual Structures: Theory and Implementation,* Lecture Notes in Artificial Intelligence ed., H.D. Pfeiffer and T.E. Nagle Eds. Berlin: Springer-Verlag, 1993, pp. 312-321.

[20] S. Polovina and V. Veneziano, "Adding Knowledge to Accounting Systems for Virtual Enterprises," in *Internet-Based Organizational Memory and Knowledge*

Management, D. Schwartz, M. Divitini and T. Brasethvik Eds. Idea Group Publishing, 2000, pp. 103-122.

[21] K. Liu, *Semiotics in Information Systems Engineering,* Cambridge: Cambridge University Press, 2000.

[22] K. Liu, R. Clarke, P. Anderson, R. Stamper and E. Abou-Zeid, "Organizational Semiotics: evolving a science of information systems, Proceedings of IFIP WG8.1 Working Conference," 2002.

[23] B.J. MacLennan, "Continuous Formal Systems: A Unifying Model in Language and Cognition," *in IEEE Workshop on Architectures for Semiotic Modeling and Situation Analysis in Large Complex Systems,* 1995, pp. 161-172. http://cogprints.ecs.soton.ac.uk/archive/00000541/

[24] P. Medway and B. Clark, "Imagining the building: architectural design as semiotic construction," *Des Stud,* vol. 24, pp. 255-273, 5. 2003.

[25] P.B. Andersen, "What semiotics can and cannot do for HCI". Accessed: 2003, 2003. http://www.cs.auc.dk/~pba/Preprints/WhatSemioticsCan.pdf

[26] I. Jacobson, G. Booch and J. Rumbaugh, *The Unified Software Development Process,* Addison Wesley, 1999.