Naked objects

A thesis submitted to the

University of Dublin, Trinity College

for the degree of

Doctor of Philosophy

Richard Pawson,

Department of Computer Science,

Trinity College,

Dublin

June 2004

Foreword by Trygve Reenskaug

(Author's note: Prof. Reenskaug was the external examiner for this thesis. One of the pioneers of object-oriented programming, he is best known as the inventor of the Model-View-Controller pattern. After the thesis was accepted, he generously agreed to write a foreword to the electronically-published version.)

The world's shipbuilding industry went through a heavy modernization program through the nineteen fifties and sixties. My colleague Kjell Kveim invented a control unit for the numerical control of machine tools such as flame cutters. The first unit was installed at the Stord shipyard in 1960 and opened the field for integrated computer aided design and manufacturing. The matching design system, Autokon, was first deployed at the Stord Yard in 1963. Autokon was subsequently adopted by most major shipyards around the world and was still in use by the late nineties.

The purpose of Autokon was to empower the ship's designers. It had a central database holding important information about the shape of the hull, the position of frames and decks, and the shapes of the parts. There was a language that permitted the designers to specify parts in shipbuilding terminology. There was a designer programming facility so that they could specify families of similar parts.

Autokon was developed in close collaboration between shipbuilders and scientists. We considered ourselves as tool builders; the success criterion was that the tools should be handy, practicable, serviceable and useful. The success and longevity of Autokon was no doubt because it was human-centric, reflecting the nature of shipbuilding and the everyday life of the shipbuilder.

In another part of the world, Douglas Engelbart worked for his vision of using computers to augment the human intellect. This quote is from 1962:

> *By "augmenting human intellect" we mean increasing the capability of a man to approach a complex problem situation, to gain comprehension to suit his particular needs, and to derive solutions to problems. . . Increased capability in this respect is taken to mean a mixture of the following: more-rapid comprehension, better comprehension, the possibility of gaining a useful degree of comprehension in a situation that previously was too complex, speedier solutions, better solutions, and the possibility of finding solutions to problems that before seemed insoluble.*
>
> *(Douglas C. Engelbart: Augmenting Human Intellect: A Conceptual Framework. Stanford Research Institute, Menlo Park, Ca., October 1962)*

Much later, in the seventies, I worked with a system for the distributed planning and control of shipbuilding operations. The goal was to create a system that could be mastered by the users so that they could tailor it to suit their individual needs without compromising the goals of the enterprise as a whole. The key was that the users' mental models should correspond to the models built into the computer. One result of this endeavour was the

Model-View-Controller architecture that I developed as a visiting scientist at Xerox PARC in 1978/79. Its purpose was to bridge the gap between the user's mind and the computer-held data. The centre of this solution was the Model that was a representation of the user's domain information. The View was an editor that enabled the user to inspect and modify this information. The Controller coordinated the capabilities of several Views making it a comprehensive tool that the user applied in the performance of specific tasks.

The original MVC was later modified in Smalltalk-80 to become a technical solution that separated input, output and information. The most important participant in the original MVC architecture, the user's mind, was somehow forgotten.

The original version of MVC was never published. In my naïveté, I believed that everybody wanted to empower their users so that MVC was merely an obvious solution to a common problem. I was wrong. There are two traditions in the applications of computers; one is to employ the computer to empower its users, and the other is to apply the computer to control its users. I am sorry to say that the latter seems to be prevalent in mainstream computing today. I have been told that in many implementations of the "well known MVC paradigm", the "C" is implemented as a script controlling the user's actions.

I can only speculate why our industry fails to give users what the clearly need and want. There could be reasons related to organizational culture, or they could be related to certain software business models. A widespread myth is that current software is inherently complex; so complex that ordinary people cannot possibly understand it and that it is only reasonable to expect flaws.

Consider a forest with birds singing in the trees and flowers covering its floor. We can easily walk along its paths or you can be adventurous and make your own paths. We can select any aspect of its complex ecosystem and study it for your doctoral thesis. There is unlimited complexity, yet any human can master it to suit his or her purposes. There is no reason why a computer system should be more complex than a forest. I believe that the current complexity is man-made, and that we can resolve it by changing our approach to software development. We merely need to get our priorities right and create the appropriate tools. If we decide to build systems for people, then we will get systems that can be mastered by people.

In the quarter century since the inception of MVC, there has been little progress in empowering the users. This is where Pawson's work comes as a fresh contribution in an otherwise drab market. If the original MVC had been published at the time, Naked Objects would now appear as an important extension and implementation of its ideas. As it is, the original MVC was not published at the time and Richard Pawson's Naked Objects appear as an important and independent contribution.

The Naked Objects method and software give two important contributions to the evolution of information system technology:

- The first and foremost is that it augments the human mind in a way that conforms with Douglas Engelbart's vision of 1962. In the seventies, Alan Kay and his group extended the augment idea with Smalltalk. Smalltalk was (and still is) a personal information environment entirely consisting of objects. The main idea was that the objects should be meaningful to the user and appear concrete by presenting themselves in an appropriate way on a screen or in a loudspeaker so that the user could observe them and manipulate them. Pawson brings this idea a significant step forward. Where

Smalltalk focuses on individual objects, Pawson concentrates on the domain model as a structure of interrelated, behaviourally complete objects. User and programmer work together to ensure that the manifest model in the computer faithfully models the user's mental model of the domain. The first field study was a project with the DSFA. One of the premises was that "*The DSFA is committed to moving away from conventional assembly-line approach to claims processing, where each person performs a small step in the process, towards a model where more of its officers can handle a complete claim and appropriately-trained officers might in future handle all benefits for one customer.*" Naked Objects strongly support this augmentation goal.

- The second important contribution is that Pawson lets the objects present themselves to the user in a standardized way. One advantage is that the user interface software can be generated automatically. Another is that the user gets into direct contact with the model since the objects are shown without being cluttered or camouflaged by fancy graphics. The result is systems that the users can feel at home in and master since they reflect the users' professional knowledge directly without any unnecessary frills.

Neither Pawson nor I believe that the current Naked Objects represent the end of the road. On the contrary, Naked Objects represent a new beginning pointing towards a novel generation of human-centred information systems.

Oslo, June 2004

Trygve Reenskaug

Acknowledgements

The author would like to thank the following for their help in connection with this thesis:

The management team at the Department of Social and Family Affairs, for their willingness to be the first organisation to attempt to build a real business system using the naked objects approach, and for providing unlimited access to evaluate the results.

The Java Services team at Safeway, again for their willingness to experiment with a new approach, and for providing invaluable feedback that resulted in refinements to the approach

Robert Matthews, who as the developer of the *Naked Objects* framework, provided an opportunity to explore many detailed implications of the naked objects concept using a purpose-designed tool.

Dan Haywood for undertaking all the programming associated with the *CarServ* case study.

My thesis supervisor, Vincent Wade, and Simon Dobson for his additional input.

My family for their encouragement and support throughout this project.

CHAPTER 1 INTRODUCTION

1.1 Motivation

In February 2002 the Association of Computing Machinery (ACM) presented its annual A.M. Turing award to Ole-Johan Dahl and Kristen Nygaard of Norway

> *'for their role in the invention of object-oriented programming, the most widely used programming model today. Their work has led to a fundamental change in how software systems are designed and programmed, resulting in reusable, reliable, scalable applications that have streamlined the process of writing software code and facilitated software programming.'* [4]

The inventors of object-oriented programming conceived 'objects' as representations of the entities that model a chosen domain, with each object encapsulating the state of that entity (i.e. its attributes, including any relationships to other objects) together with the behaviours associated with that entity [29]. In other words, objects were originally conceived as being 'behaviourally-complete'. That term is not meant to imply that an object provides every conceivable behaviour that could be required of it in any context. Rather, the term simply implies that within the context of a given application, all the functionality associated with a given entity is encapsulated in that entity, rather than being provided in the form of external functional procedures that act upon the entities.

Today, however, most object-oriented designs, and especially object-oriented designs for business systems, do not match this ideal of behavioural-completeness. Whether by deliberate design or not, the objects representing the business entities (such as Customer, Product, and Order) are often behaviourally-weak [96]. They typically have methods for updating the object's attributes and relationships to other objects; possibly they have methods for implementing constraints on those attributes or relationships; and perhaps they have methods for implementing a few simple processing functions on the object. However, much of the business functionality required for the application is typically implemented in procedures or 'controllers' that sit on top of these entity objects. Firesmith describes this pattern as:

> *'dumb entity objects controlled by a number of controller objects'* [39].

It can be seen in most custom-built object-oriented business applications, as well as in publicly-available business object frameworks such as IBM's San Francisco framework [13].

The controllers or procedures may be written in a conventional programming language, or in a high-level scripting language that supports workflow or business process modelling, or in an object-oriented programming language. The procedures may even be first-class objects in their

own right (i.e. it is possible to hold a reference to one in a variable). But this does not disguise the fact that in essence the data has been separated from the behaviour.

Firesmith suggests that this separation of procedure and data in object-oriented designs results in

> *'excessive coupling, and an inadequate distribution of the intelligence of the application between the classes'* [39].

Excessive coupling tends to hamper the 'agility' of a system - that is, the ease with which the system can be modified to accommodate unforeseen future changes to the requirements. This assertion is difficult to validate empirically because there is seldom a chance to develop the same business system using more than one approach. However, two documented attempts at such controlled experiments ([103] and [32]) both indicate that where the core business entities are more behaviourally-complete there is less coupling between the objects, and in consequence the systems are easier to extend or modify.

The question then arises: if there are known benefits to aiming for behavioural-completeness in object designs, why is it not widely practiced? Are there other factors that tend to encourage the separation of procedure data at the design stage, and, if so, could they be overcome? These questions led to the research described in this thesis.

1.2 Objectives

The objectives for this research were set as follows:

1. To identify factors that cause, or reinforce, the tendency to separate procedure and data in the design of systems, even where those systems are intended to use object-oriented approaches.

2. To identify and specify an approach to the design of object-oriented business systems that would help overcome those factors.

3. To evaluate the use of this approach for the design of real business systems, and thereby to test its effectiveness in achieving the goal of behaviourally-complete objects.

4. To test whether the use of this approach does ultimately lead to more agile systems, and whether there are any other advantages to be gained from it, as well as any disadvantages or limitations.

5. To identify types of business system, or types of project, that would potentially benefit most from applying this approach.

1.3 Contribution

The contribution of this research is the development of the 'naked objects' approach to designing business systems, and the demonstration that the adoption of this approach yields significant benefits both to the developed system and to the development process.

Using the naked objects approach to designing a business system, the domain objects (such as Customer, Product and Order) are exposed explicitly, and automatically, to the user, such that all user actions consist of viewing objects, and invoking behaviours that are encapsulated in those objects. It has been demonstrated that this approach yields four benefits:

- The resulting systems are more agile, meaning that they can more easily be modified to accommodate unforeseen future business requirements. This is primarily because the naked objects are forced to be behaviourally-complete.
- The resulting systems provide the user with a more empowering style of interaction.
- The development cycle is significantly shortened, because the presentation layer is generated automatically from the domain object definitions, and because the overall design is simplified.
- The naked objects provide a common language between application developers and users, which facilitates the early stages of requirements gathering and domain modelling.

This work has led directly to the creation of two independent frameworks to support the naked objects approach: the proprietary *Naked Object Architecture* (commissioned by a department of the Irish government) and the open-source *Naked Objects* framework. Controlled experiments and real business case studies have shown that with the aid of such a framework, all of the benefits listed above can be realised. The emergence of a substantial development community around the *Naked Objects* framework is further evidence of the genericity of the approach.

1.4 Technical approach

The research comprised six main phases:

- Researching the factors that tend to encourage the separation of procedure and data even in object-oriented designs.
- Designing a new approach to overcome this tendency.
- Testing the approach through controlled experiments and real business case studies, evaluating the results of each.

- Identifying the potential benefits and/or limitations of this approach.
- Performing a comparative analysis with other approaches that may have overlapping objectives or characteristics.
- Drawing conclusions.

1.4.1 Researching the factors that tend to encourage the separation of procedure and data even in object-oriented designs

This was primarily achieved by searching and analysing the existing literature on object-oriented techniques, their advantages and problems. Particular attention was paid to the origins of the object-oriented concept, how the thinking evolved over time, and how the idea was combined with other areas of work.

This research, which is presented in Chapter 2, suggests that one of the major factors tending to encourage the separation of data and procedure is the 'use-case controller' [41], which is effectively a functional procedure. The use-case controller is inherent in the '4-layer architecture' which has become the dominant design for contemporary business systems development, and is directly encouraged by many object methodologies including the Unified Process [57]. There is also evidence (e.g. [98]) to suggest that the use-case controller pattern is often an accidental by-product of the widely adopted Model-View-Controller (MVC) pattern [66] - even though that was not the intent behind the invention of MVC [92].

1.4.2 Designing a new approach to overcome this tendency

The new approach, described in Chapter 3, is to specify an application solely in terms of the Model (i.e. the domain entity) objects. These Model objects are then rendered directly visible to the user by means of a generic presentation layer, which embodies the View and Controller roles from MVC. The user undertakes all tasks by directly invoking methods on those Model (entity) objects. This approach has been dubbed 'naked objects', because as far as the user is concerned he or she is viewing and manipulating the 'naked' business domain objects.

This new approach is intended to facilitate the design of behaviourally-complete objects (and thereby gaining the benefit of improved agility). In addition, three other benefits accruing from the use of naked objects are predicted.

- Improved usability - because the system automatically has a truly object-oriented user interface.
- A faster development cycle - because the developers do not have to write their own Views, Controllers or anything to do with the presentation layer.

- Improved communication between developers and users during requirements analysis, because the naked objects provide a common language.

1.4.3 Testing the approach through controlled experiments and real business case studies, evaluating the results of each

A timely opportunity arose to test the application of naked objects at the Irish Department of Social and Familiar Affairs (DSFA). At the start of 1999, the DSFA was beginning to think about the design of a new business architecture that would eventually replace all of its current core systems, and was attracted to the claims (at that stage hypothetical) made for naked objects. Chapter 4 tells the story of the early exploratory work, leading to the development of a brand new *Naked Object Architecture*. The first application to be built on top of this architecture - a replacement for the existing Child Benefit Administration system - went live in November 2002, and work is under way for a second and larger application to handle the administration of state pensions. Evaluation of the case study, which included formal interviews with IT managers, business managers and user, was conducted in February 2003. The results of the evaluation provide strong evidence for two of the four benefits predicted in Chapter 3, and some support for a third.

Chapter 7 describes a controlled experiment to implement a simple business application using two different approaches: one was a conventional 4-layer approach, the other used naked objects. Formal metrics were used to compare the two resulting systems. The metrics show that the system built using naked objects required significantly less development effort, and had a simpler design. The two implementations were then subjected to a series of business change scenarios. The amount of effort required to modify each system was recorded, and the results validate the prediction that naked objects would improve the agility of the resulting system.

1.4.4 Identifying the potential benefits and/or limitations of this approach

Having confirmed, through the experiments and case studies, that the benefits proposed for naked objects could be realized in practice, the next task was to identify its possible liabilities or limitations. The combination of benefits and liabilities was used to generate further guidelines on where and how best to deploy the naked objects approach. The findings from this exercise are presented in Chapter 5.

Chapter 6 describes two short projects undertaken at Safeway Stores (the fourth largest supermarket chain in the UK), where these guidelines were explicitly applied. The evaluation shows that that both developers and users involved in these two projects felt that adopting naked objects together with the guidelines for their use provided an effective and appealing approach to developing business systems. The evaluation also confirms the predicted benefits concerning speed of development and communication between developers and users.

1.4.5 Performing a comparative analysis with other approaches that may have overlapping objectives or characteristics

Chapter 8 discusses related work including object-oriented user interfaces, other techniques for surfacing objects to the user automatically, approaches to designing empowering user interfaces, and agile development methodologies (or approaches).

1.4.6 Drawing conclusions

In Chapter 9 the research is reviewed against the original objectives (as set out above). This is followed by an analysis of the contribution of the research, including its genericity. The chapter concludes with a statement of possible directions for further research in this area.

CHAPTER 2 THE EVOLUTION OF OBJECT-ORIENTED DESIGN

The first research objective was to identify the principal reasons why the design of modern business systems continues to separate data and procedure even where there is an intent to use object-oriented principles and techniques. The approach taken was essentially historical: examining the evolution of object-oriented design concepts and their application to business systems design. That evolution, presented in this chapter, reveals a number of key developments, which, it is argued, have tended to reinforce the continued practice of separating data and procedure.

Simula and the birth of object-orientation

The idea of object-oriented software originated in Norway in the mid 1960s with Simula, an extension to the Algol programming language. Simula was designed to make it easier to write programs that simulated real-world phenomena such as industrial processes, engineering problems, and disease epidemics [29].

Prior programming languages and techniques explicitly separated software into procedure and data. The assumption underlying this separation was that a computer system repeatedly applies the same procedure to different data. Simulation software challenges that assumption. Sometimes the data is fixed and the programmer manipulates the functional characteristics of the system until the output meets the required criteria. For example, the data might represent the roughness of a typical road and the programmer might alter the design of a simulated truck suspension system until the desired quality of ride is achieved. Sometimes it is difficult to tell data and functionality apart. When another axle is added to a simulated truck, for example, does that constitute changing the data (the number of wheels) or the functionality (the way in which the truck translates road bumps into ride quality)?

The inventors of Simula had the idea of building systems out of 'objects'. Each software object not only knows the properties or attributes of the real-world entity that it represents, but also knows how to model the behaviour of that entity. Thus each Wheel object not only knows the dimensions and mass of a wheel, but also knows how to turn, to bounce, to model friction, and to pass forces on to the Axle object. These behaviours may operate continuously, or they may be specifically invoked when needed by sending a message to the object.

In the original work, each object was seen as being self-contained [29] - the attributes of an object were encapsulated with all the necessary behaviours. Another way of phrasing this is that objects should be 'behaviourally-complete'. This phrase is not meant to imply that the object provides every behaviour that might possibly be needed for all current and future applications. It simply

means that all the behaviours associated with an object that are necessary to the application being developed are encapsulated in that object and not implemented somewhere else in the system.

The value of behavioural-completeness is that any required changes to the application map simply onto changes in the program code. For example, adding a valve between two pipes in a Simula model of an oil refinery simply involves creating a new instance of the Valve class, setting its operating parameters, then linking it to the appropriate Pipe objects. The new Valve object brings with it the ability to be opened and closed, altering the flow of oil appropriately, as well as to model the impact on construction costs. If the same refinery were modelled using a conventional 'procedural' programming, the behaviours impacted by the new valve would likely be distributed around the program and therefore harder to find and change.

Smalltalk and the object-oriented user interface

Although the Norwegian work continued (and the 'Scandinavian school' of OO is still sometimes distinguished [71]), by the early 1970s a new stream of object-oriented thinking was emerging from Xerox's new Palo Alto Research Center (Parc). Alan Kay, who led the Learning Research Group at Parc, was attracted to object-orientation for several reasons.

The first had to do with scalability. At that time, discussions about software scalability were usually concerned with scaling up by one or two orders of complexity. But in 1965, Gordon Moore, who later co-founded Intel, stated that the number of transistors on an integrated circuit would continue to double every year for at least 10 years [75]. The actual trend has been closer to doubling every two years, but it has continued unabated to the present day. Kay was one of the few researchers to take the implications of the newly-coined 'Moore's Law' seriously. He was interested in how software complexity could scale up, not by one or two orders of magnitude, but by a factor of a billion or more, to take advantage of an equivalent scaling up of hardware. Kay's conception of the future of computing - notebook-sized computers with wireless connections into a gigantic network of information - looked like science fiction in the early 1970s.

Drawing an analogy from microbiology, Kay argued that software could only scale up in complexity by a factor of a billion if it was self-similar at all scales: if the most elementary software building blocks were, in effect, complete virtual machines [63] - in other words, 'objects'.

Kay also saw that the concept of objects had potential as a cognitive tool: they correspond well to the way people think about the world [62]. He noted that a verb can be thought of as a property of a noun: in the phrases 'the boy runs', 'the dog runs', 'the water runs', and 'the trains run', the meaning of 'run' is derived from the noun in the phrase. This gave rise to the object-oriented principle known as 'polymorphism': the same command (verb) can be issued to different objects, but it is up to the object to apply meaning to that verb and behave accordingly.

One product of this way of thinking was Smalltalk [63], which, as well as providing a pure object-oriented programming language, was also very innovative in its support for the user interface. (Simula had not by that stage provided any significant support for an interactive user-interface - output was usually to a line printer.) Graphical user interfaces (GUIs) were not a new idea: Ivan Sutherland had demonstrated many of the key ideas in both graphical output and direct-manipulation input with his Sketchpad system in 1963 [113] but his ideas were not easy to generalize into other applications, and also relied on specialized (vector graphic) hardware for the displays. By the early 1970s the falling cost of processing power made it possible to create similar effects in software using bit-mapped displays. Smalltalk married these capabilities with an object-oriented programming language, to powerful effect, and eventually established most of the elements of what is now the 'dominant design' [117] in interactive GUIs: overlapping windows, icons, and a desktop metaphor.

A defining principle of the early versions of Smalltalk (that is, up to and including Smalltalk-76) was that

> *'all objects are active, ready to perform in full capacity at any time. Nothing of this liveness should be lost at the interface to the human user . . . all the components of the system should be able to present themselves to the user in an effective way'.* [53]

In other words, objects were able to represent themselves directly to the user.

Although the Simula and Smalltalk languages had many differences, one thing they had in common was that in most cases the user was also the programmer. The user of a simulation written in Simula was likely to be the engineer who wrote it. In the early days of Smalltalk the intended user/programmers were children [53]. Indeed, the idea that all users should be programmers has continued to be axiomatic for Kay and subsequently led to the development of the Squeak language [54], an attempt by Kay and several of his ex-Parc colleagues to get back to that original vision.

However, by the late 1970s Smalltalk was becoming a general-purpose programming language with potential applicability to business as well as education and science. This had two far-reaching implications. The first was that the users would not typically be programmers. The second was that there would need to be a way for individual business objects to be viewed in multiple ways, either as different visual representations (such as a graph or a table), or in different business contexts, requiring different attributes to be shown or hidden.

The emergence of the Model-View-Controller pattern

This line of thinking lead to the invention of the Model-View-Controller (MVC) pattern by Trygve Reenskaug in the late 1970s (the earliest source available is [92] but MVC was not publicly documented until 1988 [66]). The motivation for this change is summarized by Reenskaug:

'One of the great inventions of the Smalltalk group at Xerox Palo Alto Research Center (PARC) in the seventies was the idea that objects can be made visible on the computer screen so that the user can see and manipulate them directly. This makes the abstract computer data appear concrete and the underlying object model visible. The user can easily adjust his mental model to this computer model and operate on it with confidence. The well-designed direct manipulation object interface is intuitively obvious and therefore easy to learn for the uninitiated. This strength of the direct manipulation object model is also its main weakness. Each object can only appear once on the screen and must always be presented in the same way to preserve the illusion of concreteness. This is insufficient for large and complex models where we need to view objects in different ways.' [93]

Under the MVC pattern, object classes are characterized into three distinct archetypes (see figure 2.1):

- Model objects model the entities of the application domain including both their state and behaviour.
- View objects create a user representation of Model objects, and handle all the interfacing with the display device. Each View represents a single Model object, but one Model can have multiple Views.
- Controller objects handle user input on a given View. They interface with the input device, and update the associated Model and View objects as needed.

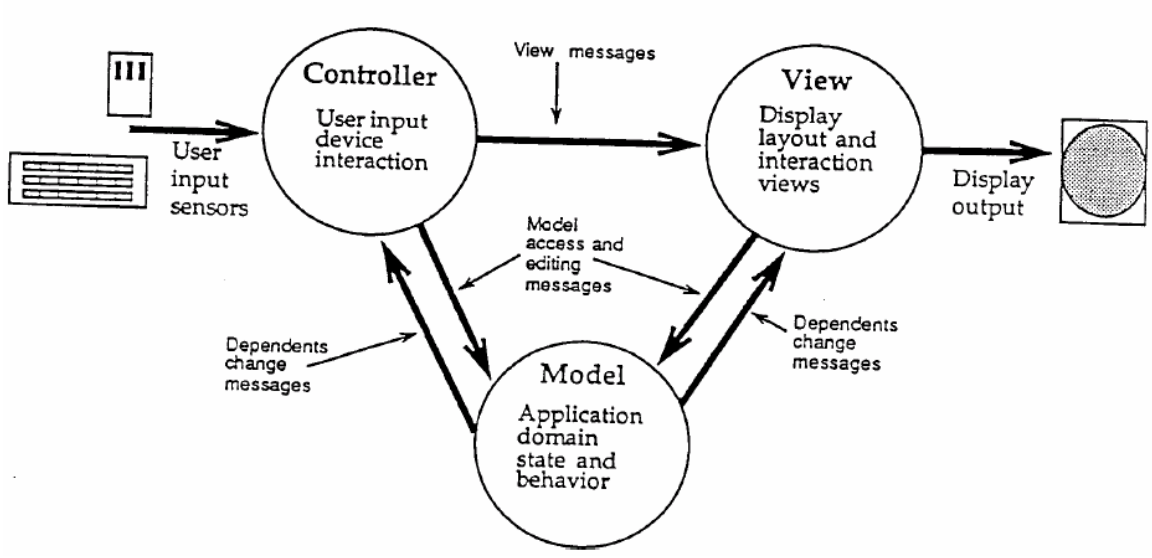


Figure 2.1: The Model-View-Controller pattern (reproduced from [66]**)**

The principal intent of the MVC pattern is the separation of concerns. If the business demands a new view of a particular (Model) object, or if the whole application needs to be ported to a new

user platform, this requires changes only to the View and Controller objects - the Model object need not be touched.

MVC is now characterized as an architectural pattern [19], meaning that it informs the whole approach to the design of the system, not just some particular feature. Moreover, this pattern, or some variant of it, has now become pervasive in new business systems that involve user interaction, to the point that it is rarely criticized or questioned.

Shortcomings of MVC

But MVC does have some known shortcomings and problems. From the outset there was a tension between the advantages of MVC and the advantages of the previous paradigm where objects were exposed directly. Reenskaug wrote many years later that:

> *'The Model-View-Controller paradigm extends the power of the user interface at the expense of increased demands on the user's mental model.'*[93]

In the Portland Pattern Repository he is quoted[1] as saying that:

> *'MVC was an outgrowth of the original direct-manipulation metaphor popularized in early OO practice (see Brenda Laurel's Computers As Theatre[69]), where you want the objects on the screen to be the objects in the program. MVC actually works against that metaphor but evolved as a necessary evil. Why? Because the user object maintains multiple simultaneous views of the model at once; the factoring into user, model, view, and controller allows one to support that.'* [94]

Alan Kay has recently further suggested that:

> *'One of the original motivations for the models, views and controller idea (that, in my opinion, never got well done) was to be able to automatically produce a default graphical interface for any object (and Steve Putz at PARC actually did a version of this but it didn't stick).'* [64]

It has also subsequently been suggested that MVC challenges the principle of encapsulation (i.e. 'behavioural-completeness'). Holub, for example, states that:

> *'MVC is okay for implementing little things like buttons, but it fails as an application-level architecture. This extract-data-then-shove-it-elsewhere approach requires you to know too much about how the model-level objects are implemented. A system based on that*

[1] The (unknown) contributor to the repository is reporting an informal conversation with Reenskaug – the quotation should not be taken as completely reliable.

approach cannot be called object-oriented: there's too much data flowing around for the system to be maintainable.' [49]

Buschmann *et al* suggest that this in turn leads to:

'close coupling of views and controllers to a model. Both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code for both view and controller. This problem is magnified if the system uses a multitude of views and controllers.' [19]

MVC thus encourages the extraction of certain behaviours of an entity object (i.e. those concerned with representing itself to the user) and placing them in separate structures. This is deliberate, and is based on the argument that the advantages outweigh the disadvantages. But in subtle ways, the MVC pattern encourages the extraction of other behaviours from the entity (Model) objects. It is very tempting to build small amounts of business logic into the Views, for example to calculate the running total of an invoice being built up on screen.

The Use-Case controller pattern

This can be even more pronounced in the case of the Controllers. Controllers were originally defined (see above) as being concerned solely with managing input. However, this definition has been progressively distorted over the years so that the term Controller is now commonly defined as 'governing the flow of control associated with a complete user task' [1]. For example, Rumbaugh (writing in 1994 about the use of MVC) stated that:

'the state diagram of a controller defines the allowable sequences of interactions inherent in a use case ... Start by assuming one controller per use case...' [98].

Fowler defines the 'use-case controller' as an explicit pattern. He positions it closer to the 'Transaction Script' pattern used in a conventional procedural programming environment, than to the 'Domain Model' pattern (which is more compatible with the goal of behavioural-completeness) [41].

The use-case controller pattern is effectively adopted by the Unified Process (UP), wherein business objects are designed according to three archetypes: Entity, Boundary and Control - the 'EBC' pattern [57]. Although EBC superficially resembles MVC there is an important distinction. In EBC, the Boundary objects are responsible for all aspects of interfacing (i.e. both input and output), to the user and to other systems; they therefore combine the View and Controller roles from MVC. In EBC, Control objects sit in between the Boundary and Entity objects and control the flow of events. They are often

'used to encapsulate control related to a specific use-case.' [57]

In recent years new technologies have appeared specifically to support the implementation of such controllers. These include workflow engines and business process modelling languages[2]. It has been argued that in addition to the benefits described above, these new technologies also make the resulting systems more agile, because the business process or task representations are directly editable [109]. However, this claim assumes that the principal requirement of agility is the ability to change the order in which tasks are fulfilled, or the flow of work between individuals. Arguably this is only one of many forms of change that business systems may be required to support. Sheth *et al* argue that:

> *'the ability to respond effectively to changes is a critical challenge for workflow management systems. . . today's workflow management systems have problems dealing with various kinds of change, ranging from ad hoc modification for an individual customer to evolutionary changes as a result of Business Process Re-engineering efforts .... and are too rigid to handle [these] dynamics.'* [104]

The four-layer generic architecture

Although not all new business systems designs explicitly adopt the MVC and use-case controller patterns, most of them do so implicitly in the form of the generic four-layer architectural pattern shown in Figure 2.2. This 4-layer pattern was first recorded by Brown in 1995 [17] although he indicates that it had clearly been practiced for some time before that. In the diagram the four layers are labelled presentation, controller, domain object, and data management. In a given implementation the names of the layers may differ, and the four principal layers may be subdivided into further layers, but the basic concept has become the dominant design for client-server business systems.

[2] See, for example, www.bpmi.org

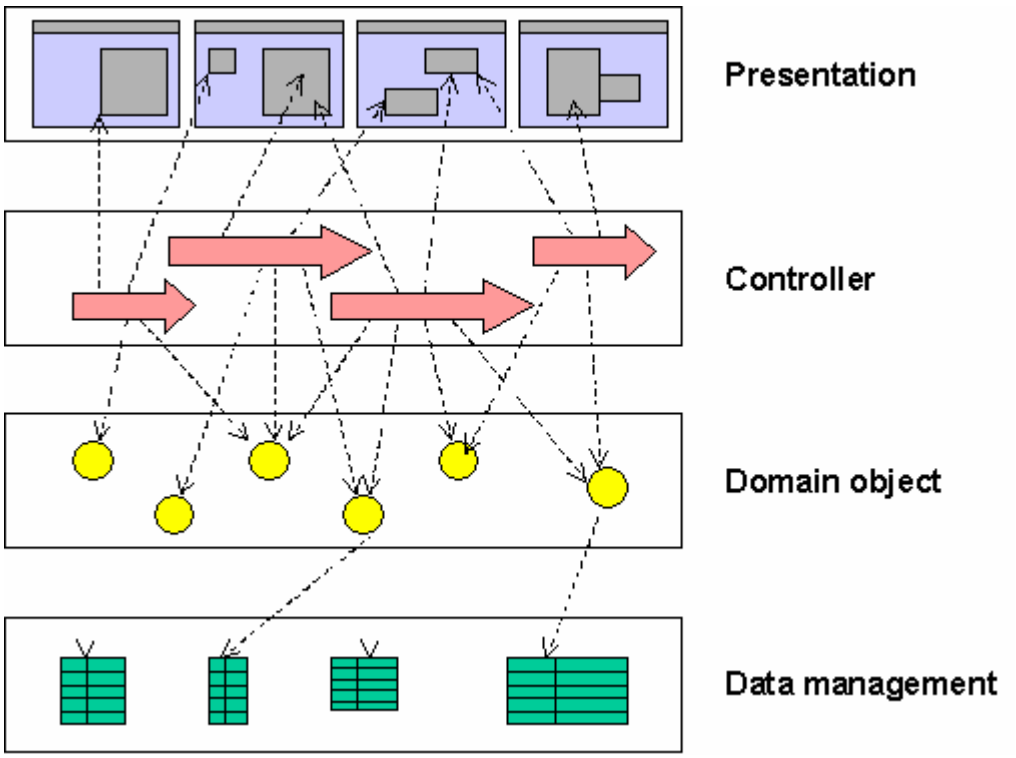


Figure 2.2: The architectural pattern adopted by most business systems is based on four generic layers.

In this four-layer pattern, a single business concept (such as a Customer) will usually be represented in all four layers, in different forms. Moreover, as the diagram indicates, the relationships between the elements in those four layers often require a complex, many-to-many mapping. Although this generic architectural pattern has evolved over the years to meet certain needs, and although each of the layers may be object-oriented in some sense, this is a far cry from the original principle of behaviourally-complete objects.

Conclusion

The conclusion is that the biggest single factor promoting the continued separation of procedure and data in object-oriented business systems design is the use-case controller pattern. Moreover the use of the use-case controller pattern seems to be almost inevitable where the MVC pattern has been adopted - even though the latter was conceived for quite different reasons.

Prior to the proposal of the MVC pattern, objects were seen as having the responsibility to display themselves. This simpler architectural model did not encourage the separation of data and procedure. However, it had limitations in terms of the flexibility of the user interface, and there is no evidence that this earlier approach was ever tried in the context of designing core transactional business systems.

The challenge that this research presents is to find a technique that will combine the simplicity of the original objects-display-themselves approach with the flexibility offered by MVC, but in such a way that is does not encourage the insertion of use-case controllers between the domain model and the user interface.

CHAPTER 3 INTRODUCING NAKED OBJECTS

The solution to the dilemma posed at the end of the previous chapter is to make the View and Controller roles (as originally defined in MVC) completely generic. In such an approach a business application is written solely in terms of the domain entity (i.e. Model) objects. The presentation layer, which permits the user to view those objects and to invoke behaviours on them, would be provided automatically. The author has dubbed this concept 'naked objects', because as far as the user is concerned he or she is viewing and manipulating the 'naked' business domain objects.

These business objects actually reside in a domain object layer of the architecture, which is often implemented on a shared server platform. Thus the user does not strictly view and interact with the business objects, but rather with Views and/or Controllers that correspond to those objects and reside in a presentation layer. However, the concept of naked objects implies an enforced correspondence between the two layers, so the illusion of manipulating the business objects is total. See Figure 3-1.

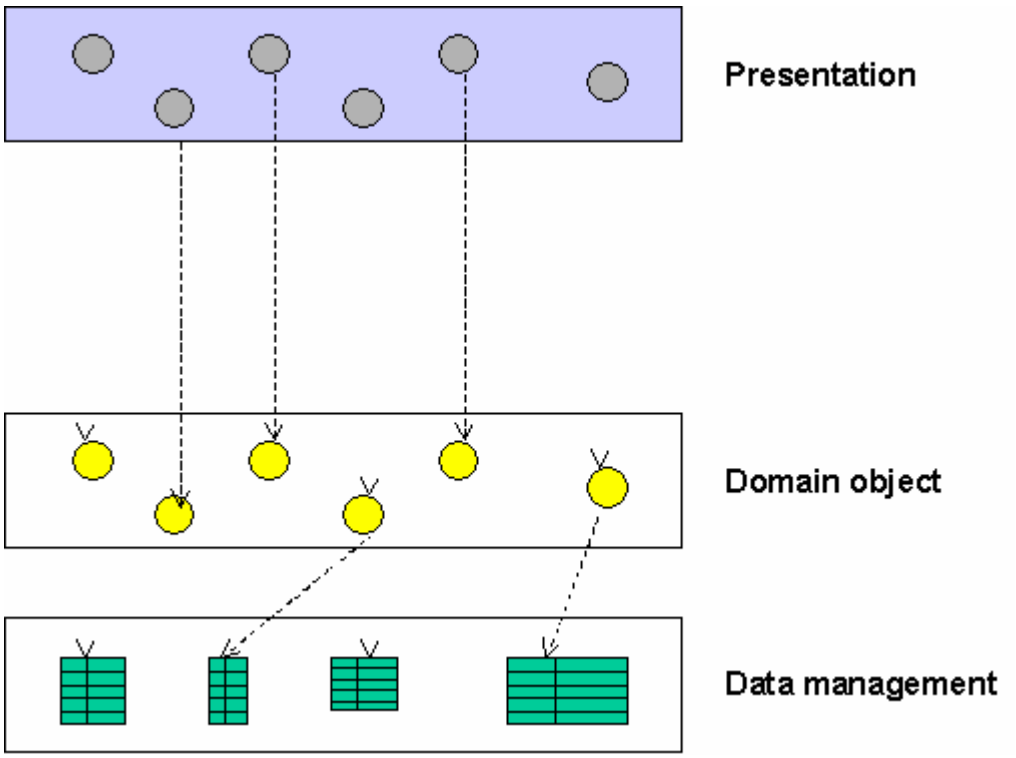


Figure 3-1 With naked objects the domain objects are rendered visible to the user by means of a completely generic presentation layer or 'viewing mechanism'. All required business functionality must therefore be encapsulated on the domain objects.

The idea of auto-generating a user interface from an underlying business model definition is not new [90]. Established examples include several fourth-generation languages; emerging examples

include the W3C Xforms standards for the web-services architecture [35] and the idea of a device-independent User Interface Mark-up Language [88]. However, all these technologies are strongly data-oriented rather than object-oriented, and their motivation has principally to do with reducing the effort associated with developing and maintaining the user interface rather than with improving the object modelling.

With naked objects, the resulting user interface is strongly object-oriented. The idea of object-oriented user interfaces (OOUIs) is well established and the relationship of naked objects to existing work on OOUIs will be addressed in Chapter 8. However, in none of the existing work on OOUIs is there a suggestion that the objects seen and manipulated by the user must correspond exactly to the domain objects in the underlying system.

How do naked objects encourage the design of behaviourally-complete entity objects? In a system designed using naked objects, the only way that the user can initiate action is by invoking a behaviour on a business object. A simple way to envisage this is to have the objects represented to the user as icons, and the behaviours made available as options on a pop-up menu for that icon. (This is by no means the only possible means of implementing the concept - others will be described later). It follows that the system designer must encapsulate all behaviours required of the application with the entity objects.

In other words, the first way that naked objects encourage behavioural-completeness is essentially negative: there is nowhere to put behaviour except on the entity objects. There is also a positive way in which naked objects encourage behavioural-completeness: envisioning domain objects as being manipulated directly by users should make it easier to identify the behaviours each one needs.

3.1 Frameworks to support naked objects

Building systems using naked objects implies some sort of software framework. ('Framework' is used here in the sense defined by Deutsch i.e. a set of classes that forms a skeleton around which an application is constructed [33]). In this case the framework must provide, at minimum, two capabilities. The first capability is an implementation of the generic presentation layer (i.e. a set of classes that fulfil the roles of View and Controller in an MVC architecture). The second is some mechanism whereby that generic presentation layer identifies the domain objects and their behaviours, in order to render them available to the user.

Two examples of such frameworks have emerged during the course of this research, the *Naked Object Architecture* and the *Naked* Objects framework, both explicitly designed to support the concept of naked objects. Both were inspired directly by the author's work, and in both cases the author played a key advisory role in their design, though not in their implementation. The author

has used both frameworks to conduct experiments, and to design and implement real business systems, some of which are described in subsequent chapters.

The *Naked Object Architecture*

The *Naked Object Architecture* was commissioned by the Department of Social and Family Affairs (DSFA) in Ireland. (The *Naked Object Architecture* was previously known as the *Expressive Object Architecture*, reflecting the author's own terminology [84] prior to 2001 when the term 'naked objects' was coined. The DSFA officially renamed its architecture in June 2003.) The background to the creation of this architecture, and the experience of building the new Child Benefit Administration system on top of it, are described in Chapter 4.

The *Naked Objects* framework

The *Naked Objects* framework is an open source project started by Robert Matthews[3] It was written in Java, and uses Java's reflection capability (known in some languages as 'introspection') to allow the generic viewing mechanism to identify the behaviours of any object.

Naked Objects is a set of Java classes that can be instantiated or sub-classed by an application. It is not a development environment. It can be used with any development environment that supports Java - ranging from a minimal text-editor and Java compiler, through to a sophisticated Integrated Development Environment (IDE) that supports graphical modelling, code analysis and support for design patterns. The systems described in Chapter 6 and Chapter 7 were creating using the *Naked Objects* framework: the former using the *IBM VisualAge* IDE and the latter with the *Togethersoft Control Center* IDE.

In addition to using reflection to generate the user presentation, the *Naked Objects* framework also uses reflection to render the objects persistent through one of several possible 'object stores', including a relational database. However, this capability is not definitional to the concept of naked objects, and is not present in the DSFA's *Naked Object Architecture.*

Naked Objects has subsequently been ported onto Microsoft technology, allowing the business objects to be written in the C# or VB.Net programming languages if so desired (though all the examples described in this thesis were written in Java).

A brief description of the framework is provided in Appendix VI. For a more complete description see [86][4].

---

[3] The framework is hosted on www.nakedobjects.org (which was co-founded by Matthews with the author).

[4] This book can also be read online www.nakedobjects.org/contents.html

The framework was featured in the 'Intriguing Technologies' track of OOPSLA 2001 [85] [5], in IEEE Software magazine [82] and many online forums. The screenshots shown in the remainder of this chapter are all from systems built using the *Naked Objects* framework.

3.2 Some immediate issues

Even before attempting to apply and evaluate the concept of naked objects, some immediate issues arise:

- How can the user create a new object instance, or perform other operations that cannot naturally be associated with a single object instance?
- How does the concept of a generic presentation layer permit alternative visual representations of an object?
- How is the concept of a generic presentation layer compatible with the requirement to support multiple forms of user platform?
- With no use-case controllers permitted, how can naked objects support the idea of business process?
- If core objects are exposed directly to the user, how is it possible to restrict the attributes and behaviours that are available to a particular user, or in a particular context?
- How is it possible to invoke multiple parameter methods from the user interface?

These are now addressed in turn.

How can the user create a new object, or perform other operations that cannot naturally be associated with a single object?

Certain required user actions are not obviously associated with any particular object instance. Examples include: creating a new instance, finding an existing instance, and creating a list of instances that match some criteria. However, all of these actions can be thought of as equivalent to class methods in object-oriented programming (known as 'static' methods in Java). Therefore, in addition to exposing individual object instances, a naked object system should provide the user with a direct representation of the business object classes, and through that representation also provide the user with access to appropriate class methods. In Figure 3-2, the icons in the leftmost window represent naked object classes ('Bookings' etc), while the other windows and the icons

[5] This paper is reproduced in Appendix IX

embedded in them represent individual naked object instances. The pop-up menu for one of the class icons ('Cities') reveals class methods to create a new City, find existing Cities and so forth.

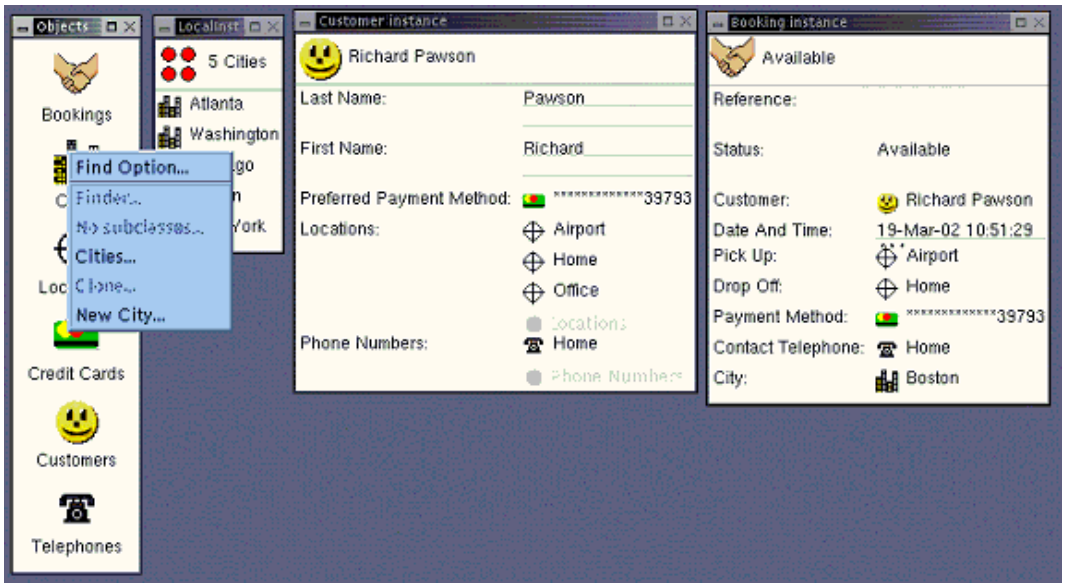


Figure 3-2 Class icons and instance icons

Similarly, it is possible to treat any collection of domain objects as an object in its own right and therefore viewable and manipulable by the user (in the same way that collections are treated in object-oriented programming languages.) In Figure 3.2 the icon consisting of four red blobs represents a collection - in this case a collection of five City objects.

How does the concept of a generic presentation layer permit alternative visual representations of an object?

One of the motivations behind the Model View Controller pattern was to allow alternative visual representations of the same object such as a table or a graph [66]. The prevailing view is that this is a common need, though this view has occasionally been challenged, for example, by Holub:

> *'How often in your work has this problem actually come up? In talking about object-oriented architectures for the user interface to many hundreds, if not thousands, of programmers, only two or three have ever raised their hands when I asked that question. If I need a generic presentation program that has no notion of what the data means, I'll buy a copy of Excel or Quattro Pro. I won't write a program. The fact is that data has meaning -- it's not just an arbitrary collection of numbers. For a given set of data, I would argue that that there is only one "best" way to represent it for a specific problem domain. If there's no "best" way, then just settle on one "good" way. This degree of flexibility is rarely required.'* [49]

It must, however, be accepted that there are circumstances in which the ability to generate alternative visual representations is a real requirement. In these cases, how is it possible to make alternative generic visual representations available to the user without requiring either a programming intervention in the viewing mechanism or the incorporation of user-interface-specific code into the domain model?

The solution is to make use of programming 'interfaces'. For example, in the Java programming language a class can only inherit functionality from one super class, but it can implement any number of interfaces. Implementing an interface means that the object can then be used in any context where that interface is specified, irrespective of which super class the object inherits from.

Thus it is possible to define a specialized visual representation based on a particular object interface. Any object that implements that defined interface is capable of being rendered into that particular visual form, and the option to do so is automatically be provided to the user.

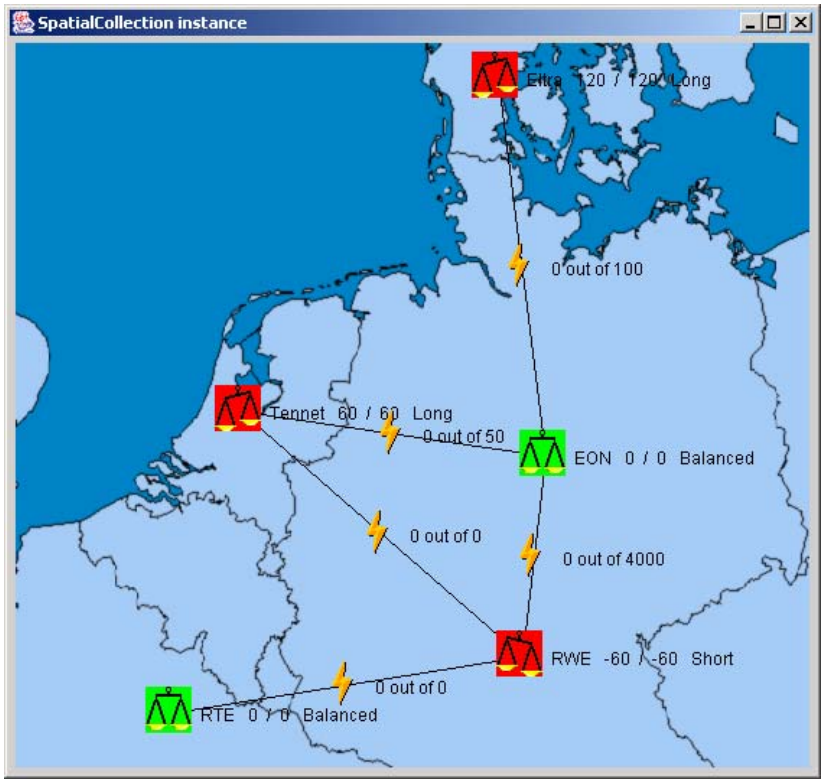


Figure 3-3 Generic map view, from an energy trading application.

The screenshot shown in Figure 3-3 is taken from the energy trading application (described more fully in [86]) in which the trading 'positions' are shown laid out as icons on a map of Europe. Each icon represents an object and is fully functional - it can be right-clicked to reveal a pop-up menu offering its behaviours, including the ability to open up a separate view showing that object's attributes and associations. To achieve this a 'Spatial' interface was defined such that any collection of objects that each implement the Spatial interface (which merely requires the object to

provide getLattidude() and getLongitude() methods) can optionally be displayed in their correct positions against the background of a map.

Similarly, any collection of numerical value objects could optionally be displayed as a graph, with that representation still providing full access to the functionality of each object. Or any collection of objects that implements a 'date' and/or 'periodOfTime' interface could have the option to be displayed in a calendar representation.

How is the concept of a generic presentation layer compatible with the requirement to support multiple forms of user platform?

Another motivation for the Model View Controller pattern was to facilitate the porting of systems to different client platforms. Using MVC, such a port involves changing the View and Controller objects, but not the Model objects.

The initial implementations of both the *Naked Object Architecture* and the *Naked Objects* framework both assumed that the system was being accessed from a PC with a high-resolution graphical display, and a mouse (or equivalent pointing device). Moreover, in both cases the generic presentation layer or 'viewing mechanism' was written in Java (though quite differently) and needed to be downloaded to the PC either as a standalone application, or as a substantial 'applet' running within a browser.

Many organizations have adopted a policy that all new business applications should be 'thin client', which in most cases means that it must run within a browser and in some cases means that it must run within native HTML, thereby ruling out even the use of Java applets. Such a policy, it is argued, eliminates the need to maintain software on the client side, and also facilitates operation over a wide area network. (The logic of these arguments is not being defended here).

A further complication is that there is a growing desire to make business applications accessible from a broad range of user platforms and devices, including Personal Digital Assistants (PDAs), and third generation mobile phones.

In fact, naked objects makes it easier to address this problem, not harder. Given the idea of a generic viewing mechanism that automatically reflects the underlying domain object model, it is quite possible to envisage alternative generic viewing mechanisms, each tailored to the capabilities of a particular viewing platform. Some of these platforms may offer a much lower bandwidth of communication between the device and the user than (say) a PC, both for output and for input. Drag-and-drop may not be feasible - for example, on a hand-held device. However, to be consistent with naked objects, the user interface does not need to make use of icons and direct manipulation. The user interface need only preserves the notion that the user is dealing directly with the domain objects and explicitly invoking behaviours on those objects. In other words, the

style of interaction is 'object-action' (or 'noun-verb') [91]. It is quite possible, for example, to envisage each object as a web-page and with each behaviour as a hot button on that page.

With no use-case controllers permitted, how can a naked object system support the idea of business process?

The definition of naked objects requires that all business functionality must be encapsulated within entity objects rather than within use-case controllers (or other broadly equivalent structures) that sit on top of those entities. It will be argued later on that this approach provides the user with a more 'expressive' (or empowering) user interface.

Nevertheless, there are situations where there is a real need for some kind of sequencing: for example, to enforce adherence to a regulated business process ('A customer must be given a full written quotation two weeks before a contract becomes binding') or a fundamental piece of business logic ('A booking cannot be confirmed until availability has been checked').

This is achieved using naked objects by distinguishing two broad categories or stereotypes of entity object: 'purposeful' and 'non-purposeful'. Some examples of non-purposeful objects are Product, Customer, Employee, and Location. With non-purposeful objects the state of the object is defined implicitly by the agglomeration of its various attributes and associations. The state of these non-purposeful objects will change over time, and that state will be made persistent, but the changes do not advance in any particular direction. They can be thought of as random.

By contrast, the state of a purposeful object is usually defined explicitly, and is often represented by a single field that can take one of a finite set of pre-determined values. Moreover, this state generally changes in a pre-ordained direction; and this is typically defined using state-transition diagrams, which specify the conditions under which the object will move from one of those pre-determined states to another. Thus, an Order may go from the state of an Enquiry, to Committed, to Shipped, to Invoiced. The status may occasionally backtrack, or the Order may be terminated prematurely; but there is a clear intended direction.

This concept of purposeful objects is broadly similar to the 'moment-interval' archetype identified by Coad [22], (the non-purposeful objects corresponding usually to the 'people, place or thing' archetype) which could be used as an alternative, as could the stereotypes defined by Wirfs-Brock [119].

It might be suggested that a purposeful object is just a use-case controller by another name. There is, however, an important difference. In a naked objects system, both non-purposeful and purposeful objects are entity objects. They are rendered explicitly to the user as an object (for example as an icon). They are made persistent (at least by default). They continue to exist as objects even when they have reached their intended end-state. In most approaches to object

modelling, once a transaction, process or use-case is completed, there is no way of referring to it explicitly. In a naked object system, these activities show up as objects in their own right: they can be viewed and inspected, and it is possible to invoke any of their behaviours that remain valid given their status. Thus, even after a (bank) Transfer has been made, the user could examine it, decide to reverse it, charge a fee for it, or notify the customer of its successful completion - all methods that can naturally be encapsulated in that Transfer object.

Any business activity where the verb describing the activity can easily be mutated into a noun is a prime candidate for a purposeful object. Thus the users might have a requirement to 'adjust' (verb) the prices. But they will also talk readily about 'making price adjustments'. This phraseology should be a cue for thinking about PriceAdjustment as an instantiable entity object, rendered visible to the user (e.g. as an icon).

The idea of treating a bank transfer as a persistent entity object rather than a transient controller is not original (Riel, for example, makes the case for treating Transfers, Withdrawals and Deposits in this manner [96]) but naked objects make the advantage of this way of thinking clearer.

If core objects are exposed directly to the user, how is it possible to restrict the attributes and behaviours that are available to a particular user, or in a particular context?

Using naked objects, domain objects and their capabilities are exposed directly to the user. But there are many circumstances where it is not desirable for all the attributes or capabilities of an object to be available to the user. For example:

- A particular user's role may not qualify them to view all data or invoke all actions on an object.
- If an object is used in many contexts (for example a Customer object) it may acquire a great many attributes and behaviours. Displaying them all would result in unnecessary screen clutter, database accesses, or network transmission.
- Specific attributes or behaviours should be unavailable when the object is in certain states. For example, it should not be possible to attempt to check the availability of a booking until sufficient information has been provided. (This applies mostly to 'purposeful' objects as described in the previous point.) This requirement might be summarized as the need for selective viewing of an object's capabilities.

If the user interface is auto-generated from the definitions of the business objects, and no programming intervention is permitted, it seems impossible to provide selective viewing of an object's capabilities. The solution is to provide some capability on the domain object itself for controlling availability: of that whole class of objects, of specific instances of that class, and of individual methods on that class. (Note that, assuming the normal object-oriented practice is observed of keeping all variables private - accessible only through 'accessor' and 'mutator'

methods such as ‘get’ and ‘set’ - then control over the availability of methods gives control over the access to attributes and associations as well as to richer behaviours.)

In the *Naked Objects* framework this was achieved by permitting any method to have a corresponding ‘about’ method. Thus the method actionConfirm() on a Booking object could have a corresponding aboutActionConfirm() method which controls its availability. This aboutActionConfirm() method might control availability based upon the state of the Booking object, or it might delegate the decision to an authorization server that looks up the authorization for the user. In one version of the viewing mechanism, unavailability is signalled to the user by greying-out a menu action, rendering a particular field (attribute) uneditable, or excluding it from the view. The ‘about’ method can also return a reason for the lack of availability, which the viewing mechanism can render as ‘balloon’ help or similar.

How is it possible to invoke multiple parameter methods from the user interface?

Any zero-parameter methods on a business object can automatically be rendered accessible to the user as options on a pop-up menu. Single parameter methods can be rendered accessible using drag-and-drop gestures - dropping object A onto object B would invoke the method on B that has an object of type A as its parameter. (If there is more than one such method, then the user interface could generate a pop-up menu from which the user can choose). But what about multiple-parameter methods? The two frameworks introduced in section 3.1 differ in the way that they address this issue.

The DSFA's *Naked Object Architecture* renders multiple-parameter methods automatically into dialog boxes. The user selects the method from a pop-up menu, which then returns a dialog box with labelled fields corresponding to the various parameters and into which the user can type values or drop other business objects as appropriate. The method is then executed via the 'OK' button.

The *Naked Objects* framework does not (at the time of writing) make use of dialog boxes, and as such does not provide any means for the user to invoke a multi-parameter method on a business object. Interestingly, experience of deploying the framework across a variety applications suggests that this is less of a limitation than it might seem at first.

Consider the example of a banking system where Account is a class of business object, with methods for depositing, withdrawing, generating statements and applying charges, amongst others. How should the system handle a transfer of funds between accounts? A comventional design might implement transfer as a use-case controller sitting on top of the objects, but this is contrary to the naked objects approach. However, implementing transfer as a method (on the Account that the

money is being transferred from) would require two paramaters at minimum: the amount to transfer and the Account to transfer it to. This is not possible with the *Naked Objects* framework.

The solution is to implement Transfer as a (purposeful) object in its own right . Its attributes are the two accounts, the amount to transfer, and the date/time. The Account object then has a method called 'Create New Transfer', which creates a new instance of Transfer, ready populated with the 'from' account. Alternatively, the user could shortcut this by dragging one account onto the other, which returns a new Transfer object with both the 'from' and 'to' fields populated. After specifying the amount to transfer, the user then invokes the 'Execute' or 'Make it so' method on the Transfer object.

3.3 Proposed benefits for naked objects

Four possible benefits from designing with naked objects can be inferred from the preceding discussion:

- Behaviourally-complete objects, leading to more agile systems
- A faster development cycle
- A common language between developers and users
- A more empowering user interface

These are briefly defined below.

Behaviourally-complete objects, leading to more agile systems. The initial motivation behind the idea of naked objects is to encourage the design of behaviourally-complete objects, and thereby to deliver systems that are more agile - meaning that they can be adapted more easily to accommodate unforeseen future changes to business requirements.

A faster development cycle. If an appropriate framework is used, naked objects will speed the development cycle. This is principally because it is no longer necessary to develop a presentation layer, which often accounts for a high proportion of the total development effort expended on a business system.

A common language between developers and users. Naked objects will provide a genuinely common language between the developers and users during the business-analysis and requirements-gathering phase of a project. Traditionally, the bulk of the effort in this phase of development is concerned with translating the language of the user presentation into that of the systems modelling domain. Using naked objects the two representations are the same. This, in combination with the previous benefit, should make it possible to prototype in real time in front of users - and to prototype not just the screen presentations and user actions, but also the underlying

object model at the same time. Moreover, this common language of objects, classes, associations and methods, should mean that the business user representatives can easily relate to the object model, and perhaps gain further benefits from thinking about their business using object-oriented patterns.

A more empowering user interface. The user interface resulting from the use of naked objects will clearly be an object-oriented user interface (OOUI) [24]. The naked objects approach does not improve upon the best work on OOUIs to date (which will be discussed in more detail in Chapter 8), but whereas OOUIs are traditionally considered to be quite difficult to implement, naked objects (or strictly speaking, the framework that supports them) makes them trivial to implement. It is not suggested that OOUIs are universally better than other forms of user interface. But the most significant advantage is that they empower the user - they treat the user more like a problem-solver and less like a simple process-follower [83].

These four benefits are potentially synergistic - each one strengthening the others and/or making them easier to realize.

CHAPTER 4 DEVELOPMENT OF A NEW BENEFITS PROCESSING SYSTEM FOR THE IRISH GOVERNMENT

In order to test the viability of the naked objects approach, and to validate the benefits predicted in the previous chapter, the author sought appropriate opportunities to design and implement business systems from naked objects. The first such opportunity arose at the Irish Department of Social and Family Affairs early in 1999.

4.1 Background to case study

The Department of Social and Family Affairs (DSFA) is the department of the Irish government responsible for administration of social benefit schemes. Prior to 1998 it was known as the Department of Social Welfare. The Department depends heavily upon information technology to fulfil its tasks. It has some 2000 PCs, but its core transaction processing programs (both on-line and batch) are mainframe-based, and accessed via some 4000 green-screen dumb terminals. These systems are technologically out-of-date and increasingly expensive to maintain. For example, they require manual re-programming every time the government changes the benefit rates or rules. Currently, there is a separate system for each major type of benefit - Child Benefit, Disability, Unemployment and so on. Although there is a Central Records System (a customer database, effectively), there is much less sharing of both information and functionality than there could be. For example, many systems have their own separate mechanisms for generating payments. Word processing, email and calendar facilities are provided through the mainframe-based All-in-One suite, and there is no integration between this and the transactional systems.

Demand for greater organizational agility within the Department has been growing for some years, and this translates into demand for greater agility within the information systems. Technological developments such as the Internet and smart cards offer significant potential benefits both to the DSFA and its customers, including ease of access, richness of information, and cost savings. The government has also been pressing for more agility, both in the ability to introduce new forms of benefits, and in improving service to the customer. There are various e-government initiatives in Ireland, of which the most significant is the REACH programme, which will provide a common 'e-broker' for accessing information about the services offered by multiple government agencies, a central means of identification and authentication, and a personal data vault that gives the customers greater control over their own privacy.

In response to these various demands, in 1999 the DSFA conceived a new Service Delivery Model (SDM) that emphasizes electronic commerce, agility and customer-responsiveness. The SDM highlighted the need for a complete new architecture for the core systems, one that would not only

support the specific needs of the SDM, but would also be more adaptable to future, as yet unforeseen, business changes.

4.2 Early experimentation

Early in 1999 the IS management became aware of the author's research on building business systems from behaviourally-complete objects, and the emerging concept of naked objects. Although the management did not necessarily accept all the theoretical arguments for naked objects at that stage, the hypothesized benefits meshed well with the Department's own goals for its proposed new architecture. The IS management was also attracted to the visual concreteness of naked objects (i.e. the fact that core business objects were also user constructs), and felt that this would make it easier to get non-IT managers more involved with crucial design decisions.

Early in 1999, the IS department initiated some educational workshops on object-oriented thinking for some of its senior managers, both IT and business, with the help of the author. Given that there was no framework to support naked objects at that point, and there were no known examples of transactional business systems having been built this way, a piece of consumer software called The Incredible Machine was used as a metaphor throughout the workshop. (This is a good example of using a single metaphor to guide the high-level design of a business system, as advocated in the discipline of Extreme Programming [9]). In The Incredible Machine (see Figure 4-1), the user is presented with a series of physical challenges, and must construct a simulation of a complex, improbable-looking, machine (in the style of the artists Heath Robinson or Rube Goldberg) to solve them. As well as demonstrating the notion of a problem-solving system, The Incredible Machine is also very clearly object-oriented from the perspective of the user: the elements that the user drags from a parts catalogue into the workspace are not just visual representations, but bring with them the complete simulated physical behaviours of that part.

Figure 4-1 The Incredible Machine was used as the system metaphor during the early stages of the design project.

One of the workshops (a one-day exercise involving six managers) attempted to identify the core business objects that could potentially model the DSFA's business. The participants were invited to suggest object categories directly: no attempt was made to specify use-cases up front. By the end of the morning a list of approximately twenty candidate object classes had been produced. During the afternoon this list was reduced by looking for duplicates or synonyms, by recognizing that certain candidates were natural sub-classes of others, and by eliminating candidates that, with further discussion, turned out not to be core business objects, because they could be better represented as simple attributes or as methods of other objects. The result was a list of seven core classes: Customer, Case, Scheme (meaning a Benefit Scheme), Payment, Service (a non-monetary benefit such as a bus pass), Community, and Officer. (Of these original seven, all but one have survived in roughly the form that they were defined on the first day. Community was dropped due to a subsequent minor redefinition of the Department's responsibilities. It had briefly been the Department of Social, Community and Family Affairs.)

The group was next asked to imagine how users would interact with each object, by asking such questions as:

- What will this object look like on the screen?
- Where might a user want to drag it?
- What actions will a user want to be able to invoke by right-clicking on this object?

The result was a list of 'know-what' and 'know-how-to' responsibilities [120], with emphasis on the latter to avoid thinking of the objects merely as complex data sets. The result was a very rough definition of a set of behaviourally-rich objects.

These draft object responsibility definitions were translated into a crude visual mock-up of a system that might be used by an officer who makes decisions relating to a claim (there are many hundred such officers within the Department). The mock-up was actually a series of hand-drawn screenshots held as PowerPoint slides (see Figure 4-2), but a well-rehearsed demo created a realistic simulation of a working system, with the impression of icons being dragged around the screen and menus apparently popping up in response to a right mouse-click. This simulation demonstrated a small set of specific use-cases. However it is important to understand that these use-cases were used only to illustrate the potential functionality of a system built from a small set of behaviourally-complete objects. The use cases were not used to analyse the requirements or identify objects; indeed, they were written only after the objects and their high-level responsibilities had been specified. (This theme is picked up again in Chapter 5).

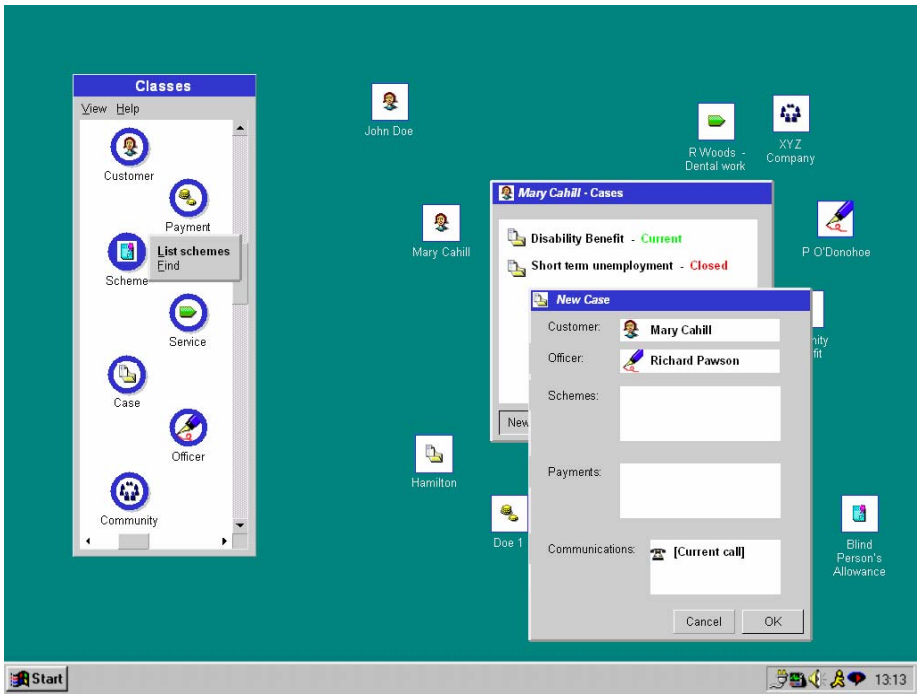


Figure 4-2 Original PowerPoint mock-up of the user interface for a system designed using naked objects

The mock-up was demonstrated to a group of senior managers who had not been involved in the modelling workshop. A few of the responses sum up its impact:

- *'I can see how everyone in the entire organization, right up to the Minister himself, could use the same system'*. This did not mean that all users would perform the same operations, or indeed have the same levels of authorization; rather, that everything the organization

does could be represented as actions on the handful of key objects. Such a consistent interface could help to break down some of the barriers between internal divisions, as well as making it easier for individuals to move into different areas of responsibility.

- *'This interface might be sub-optimal for high volume data entry tasks'*. There was some debate about this until someone pointed out that the DSFA's commitment to increasing electronic access (via the web, smart cards, and telephone), plus a more integrated approach to the systems themselves, means that much of the routine data entry work will disappear anyway.

- *'This system reinforces the message we have been sending to the workforce about changing the style of working'*. The DSFA is committed to moving away from a conventional assembly-line approach to claims processing, where each person performs a small step in the process, towards a model where more of its officers can handle a complete claim, and appropriately-trained officers might in future handle all the benefits for one customer. The managers at the demonstration felt that even this simple mock-up could help to convey to users the message that they are problem-solvers, not process-followers. This was in marked contrast to the approach that had been proposed by some vendors, which emphasized using rules-based technology or 'intelligent software agents' to automate as much decision making as possible. Instead, the object-based mock-up suggested an environment where the users' natural problem skills would be highly leveraged. In this sense, the design of the system could be seen as helping to facilitate a fundamental cultural change. In other words, although the initial motivation for exploring the naked objects approach was its potential to improve the agility of the systems, after the first mock-up demonstration the managers also became interested in the potential benefit to be gained from the more empowering style of user interface that naked objects would bring about.

In addition to this positive reaction from the user representatives, the IS management were also impressed with the productivity of this exercise compared to previous attempts at department-wide modelling (whether as objects, data, or processes). Although the DSFA was by no means ready to commit to using naked objects for a full-scale implementation at that stage, all agreed that the concept should be explored in more depth.

4.3 Applying the concept to Child Benefit Administration

At the beginning of 2000 the business case for action became more urgent. The existing Child Benefit Administration (CBA) system needed to be replaced with some urgency. The Government had indicated possible future changes to Child Benefit that the existing system simply could not be modified to address (specifically, the possibility of the rate of benefit being dependent upon the age

of the child). Child Benefit is one of the simpler schemes that the DSFA administers and is relatively small in scale: the existing system had around 80 users. Yet the activities involved in processing a claim had much in common with other schemes. It seemed an ideal opportunity to test the deployment of a new approach.

In a series of workshops involving both senior managers and user representatives, the responsibilities outlined in the original object model (from the one-day workshop) were now fleshed out from the particular perspective of Child Benefit Administration. Object responsibilities were refined and new responsibilities identified. New sub-classes and secondary (or 'aggregated') objects were also added. And the whole model was crudely tested against a number of operational business scenarios. Additionally, the model was tested (on paper) against a variety of scenarios of how the DSFA's business might change in future. The business object model was captured in the form of textual descriptions, which are reproduced in Appendix II.

During these workshops, the DSFA made use of a very early prototype of the Java-based framework being developed by Robert Matthews, which was eventually to become *Naked Objects*. Although this was not the technology that the DSFA eventually used for implementation, using the framework helped to build the participants' understanding of the concept of naked objects. The ability to capture the object model in the form of a usable system (rather than just as diagrams) was a significant factor in persuading managers to participate in the modelling process.

4.4 Technology demonstrators

Meanwhile, another group of IT managers within the DSFA worked on defining the principles of the *Naked Object Architecture* that would fully support the concept of naked objects and at the same time meet the DSFA's other specific needs in an architecture for core business systems. The principles are reproduced in Appendix I, but the following excepts demonstrate the commitment to naked objects:

> ***'Exposure***. *Naked objects are exposed directly to the user - in a form that makes it obvious to the user that they are dealing with an object. This includes the use of an icon to represent the object. Most importantly, though, it includes the exposure of the object's potential behaviours to the user who typically selects an object and then invokes a behaviour upon it. This implements a 'noun-verb' style of user interaction, rather than the more common verb-noun style. In this way, the Naked Object Architecture presents the user with a set of tools with which to operate and allows a business system to be designed that does not dictate the users sequence of actions. This allows the user to be a problem solver rather than simply a process follower.*

Class methods. *As well as exposing object instances, the [Naked Object Architecture] provides the user with an explicit representation of the classes of Naked objects. Using these class representations, the user can initiate a set of class methods, including for instance methods for creating a new instance of that class, for retrieving a particular instance of that class from storage using a unique identifier and for finding instances that match specified criteria.*

Single point of definition. *Naked objects are ideally defined in a single place. The representation and role of the Naked object in multiple tiers of the architecture are all derived from that single definition - including the definition of persistent storage in a database.*

Auto-generated user interface. *The default user interface is also ideally automatically created from the central definitions of the Naked objects. The Business Object defines for each of its methods the business-related information the Presentation Layer needs to display and to capture from the user. This information is subject to security filtering, to ensure that each user is only shown the information he/she is authorised to see.'*

As a public sector agency, all procurement is subject to the European Union tendering rules. Accordingly, the DSFA published an RFT (Request For Tender) for 'technology demonstrators' that would implement the principles of the *Naked Object Architecture* including all the services required of a scaleable enterprise architecture (e.g. message broking, transaction monitoring, and persistence using a relational database). The Department commissioned three such demonstrators from three different vendors, each using different object technologies:

- Sun's Enterprise Java Beans (EJB)
- Microsoft's COM+
- A proprietary object-oriented package.

4.5 Phase I implementation

Having evaluated the demonstrators, the DSFA put out a new RFT for the development and implementation of a full-scale *Naked Object Architecture* plus a specified set of domain object classes needed to provide a complete new application that would replace the existing Child Benefit Administration system. The RFT did not provide a full functional specification, but it did provide the description of the high-level responsibilities required of the core business objects (Appendix II). The contract was awarded in February 2001 to DMR Consulting (since renamed 'Fujitsu Consulting') - with the original intent that the new system would go live in March 2002. (The author continued to act as an advisor to the DSFA on an ad hoc basis throughout the

implementation - but had no direct connection to the contractor). The project was subject to delays - some from over-runs of effort, some from technical difficulties, and some from industrial action. However, the system eventually went live in November 2002, completely replacing the old system by the end of that month.

Figure 4-3 shows the physical architecture of the system. The client machines (shown on the left) run a generic viewing mechanism written as a Java applet running within a browser. The business objects run on the 'COM+' servers shown to the right of the client machines. Microsoft SQL Server is the primary persistence mechanism. However, several of the object classes obtain some or all of their data from existing databases running on Oracle or OpenVMS storage platform (because that data is shared with other applications).

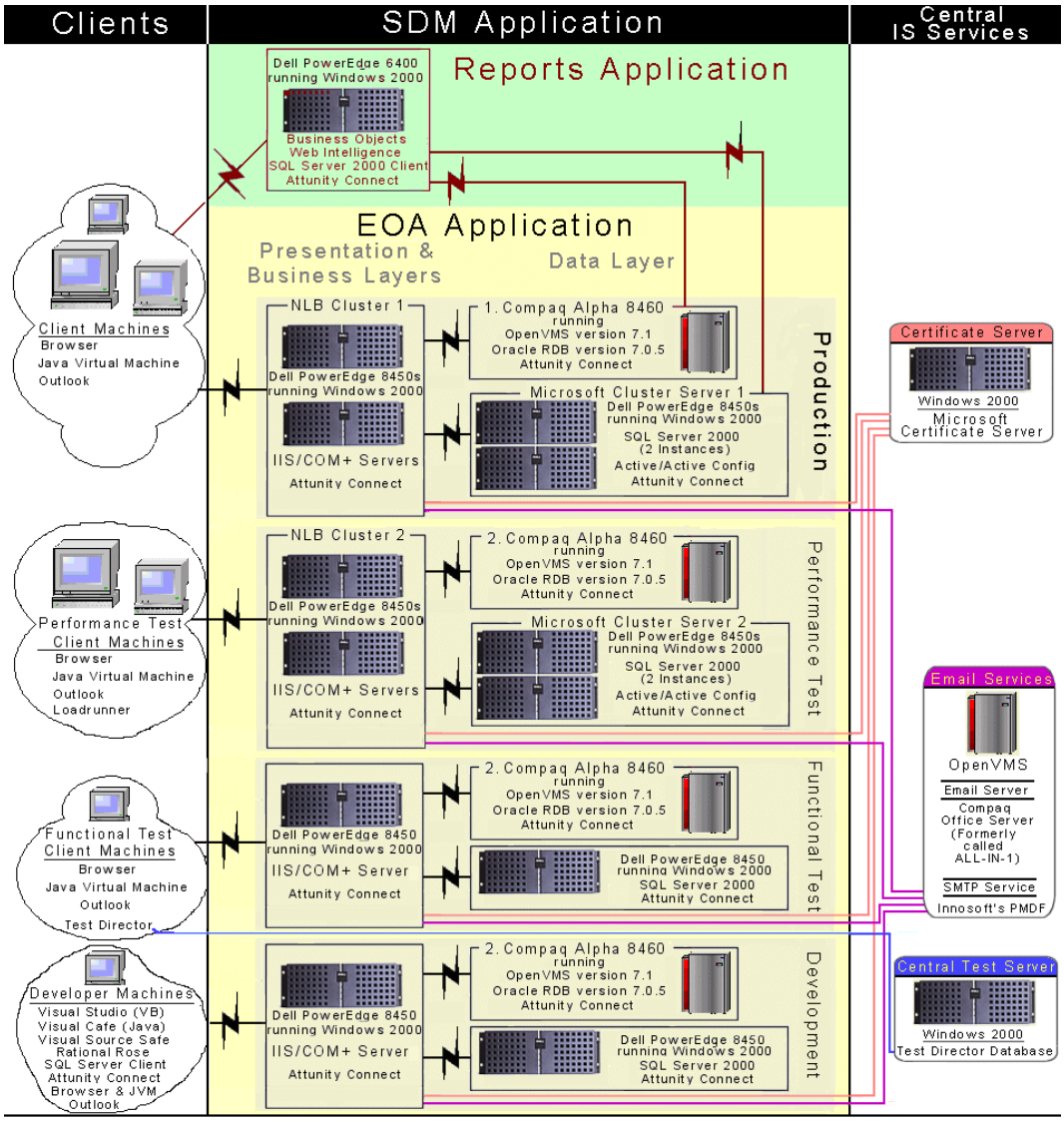


Figure 4-3 Physical architecture of the DSFA's Child Benefit Administration system

Figure 4-4 shows the logical architecture. All business logic is contained in the entity object classes (labelled 'Business Objects' in this diagram). Each business object has a corresponding

Data Access Object, which manages the mapping of the object's attributes onto a persistent storage mechanism. Part of the reason for this split in responsibilities is that some of the business objects share their persistent representation with other systems, in particular through the Central Records System (CRS). A product called Attunity Connect manages the interchange of data with all existing databases. The business objects and the data access objects were written in Visual Basic 6.0.[6]

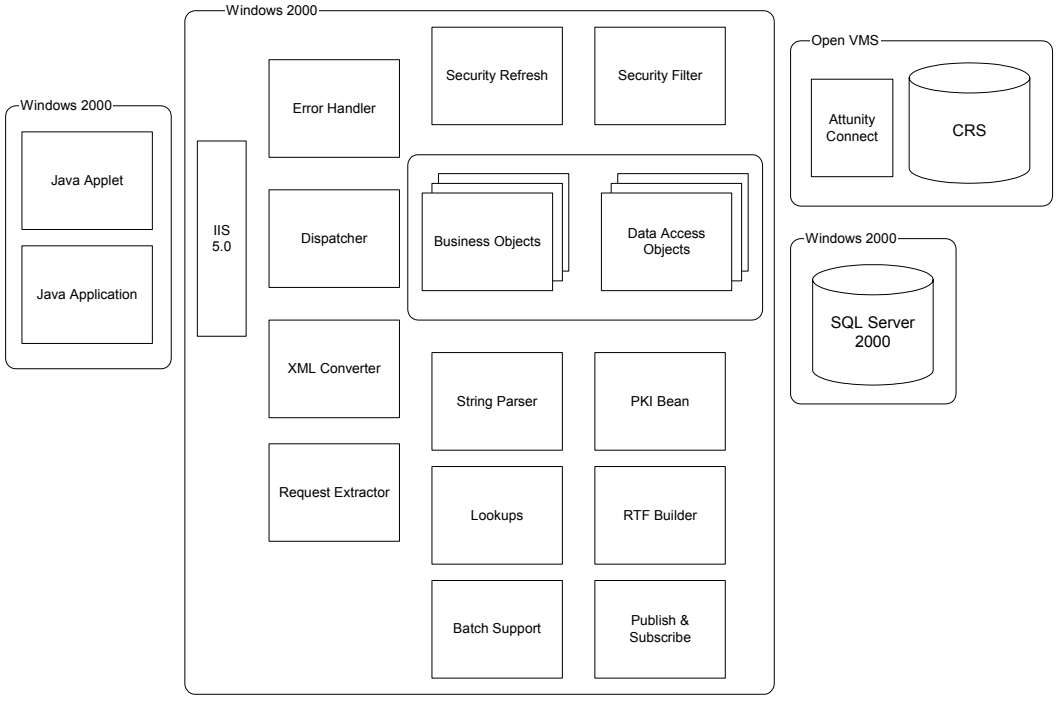


Figure 4-4 Logical architecture of the *Naked Object Architecture*

The communication between the business objects and the generic viewing mechanism, which passes through several layers of middleware, takes the form of XML messages. Thus, a business object in the domain layer will send an XML message that instructs the viewing mechanism to present an object on the screen, by default as an icon. The XML message also specifies which attributes can be viewed and/or edited if the user chooses to open a view of that object; associations to other objects (which will show up as icons in their own right), and a list of methods that will be offered in a pop-up menu if the user right-clicks on that object. A similar mechanism generates the class icons with their pop-up menus of class methods. Figure 4-5 shows a screenshot from the application.

[6] In June 2003 the DSFA issued a new RFT to migrate these objects onto VB.Net. As well as moving from the COM+ to the .Net service architecture, this migration will also allow them to take advantage of the more object-oriented nature of the VB.Net language, which, unlike VB 6.0, supports inheritance.

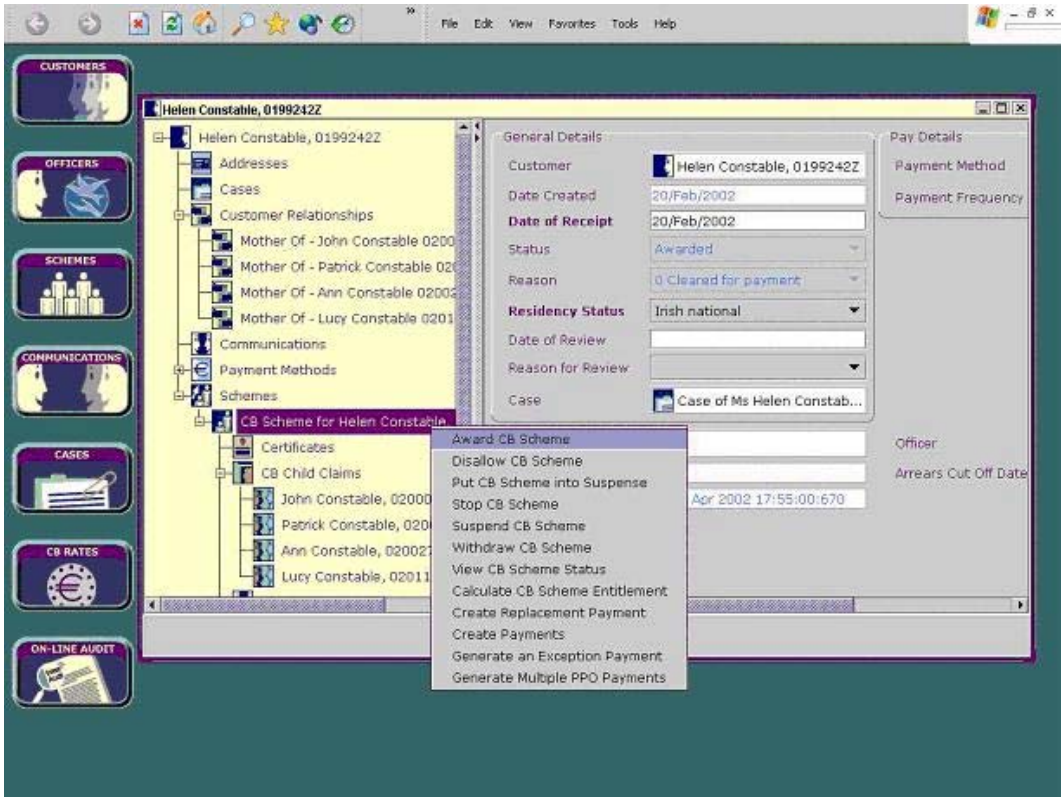


Figure 4-5 Screenshot from the new Child Benefit Administration system showing the naked objects

The user can move objects around, expand or contract views, explore pop-up menus and edit allowable fields without any communication passing back to the middle tier. When the user is ready to save the state of a modified object, or wants to create a new association between object instances, or to invoke a business method from a pop-up menu, then the viewing mechanism will generate and send a concise XML message to the appropriate object in the business model layer, specifying the action and any necessary parameters.

4.6 Phase II

At the time of writing, the DSFA has indicated its intention to proceed with rolling out the *Naked Object Architecture* into other benefit areas, starting with the state pension administrations systems. The outline object model has already been extended to cover this area. An RFT was issued for this in the Autumn of 2002, but was subsequently withdrawn due to government-level budget constraints. Phase II has been rescheduled to commence late 2003.

4.7 Evaluation

In February 2003 the author conducted an evaluation of the implemented system, by interviewing 25 individuals connected in some way with the project:

- 7 IT managers who had some responsibility for the specification and/or implementation of the *Naked Object Architecture*
- 3 business managers who were involved in the business object modelling for the first business application (Child Benefit Administration) using the new architecture.
- 15 users of the implemented Child Benefit Administration system all located in the DSFA office in Letterkenny, Donegal.

Details of the interview method, the questionnaires used, summaries of responses, and additional commentary on these three sets of interviews are provided in Appendix III, Appendix IV, and Appendix V respectively.

The following is a list of general messages derived from the analysis of those interviews and the author's evaluation of the system itself.

4.7.1 User reaction to the system has been very positive

Figure 4-6 shows that almost all of the users recognize and value the flexibility that the new system provides them in terms of their way of working.

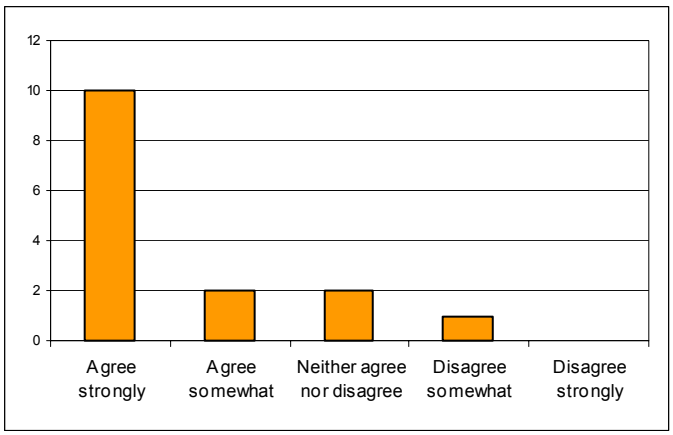


Figure 4-6 Response of 15 users to the proposition:
'I value the flexibility that the system provides in choosing how to undertake a task'

Early in the project some managers had expressed concern that users might be unhappy with the unscripted style of the naked object user interface and might prefer that system guide them through the steps of the tasks they wanted to undertake. As shown in Figure 4-7 that turned out not to be the case.

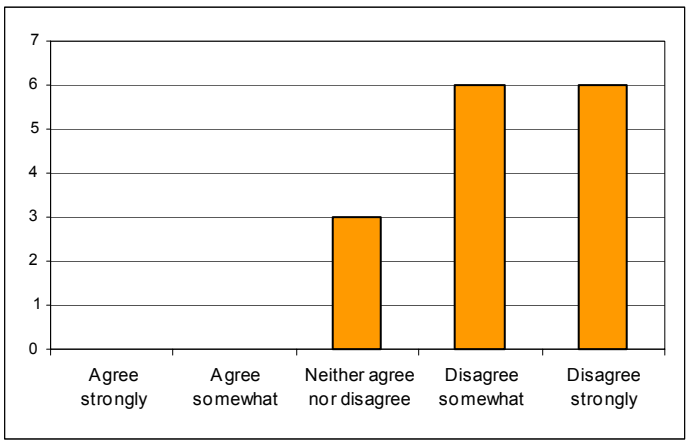


Figure 4-7 Response of 15 users to the proposition:
'I would prefer the system to guide me through the steps of a task'

Figure 4-8 shows that, at least in the user's minds, the system's operational flexibility translates into better customer service.

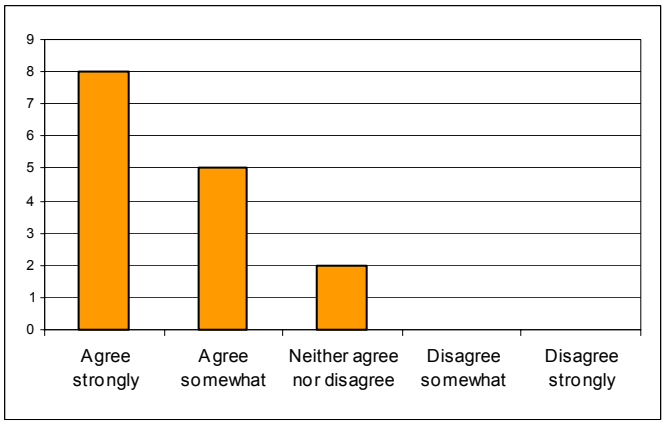


Figure 4-8 Response of 15 users to the proposition:
'The new system permits me to better deal with the needs of individual customers'

Figure 4-9 shows that 9 out of 15 felt empowered by the new system. Given the other results shown in this section it is perhaps surprising that the percentage is not higher still. The six who responded 'Neither agree nor disagree' all seemed genuinely puzzled by the question, suggesting that there was a terminological issue. Several of them asked the interviewer for clarification. However, it was felt that on this particular question, any such clarification would probably lead the respondent toward endorsing the proposition, so none was offered.

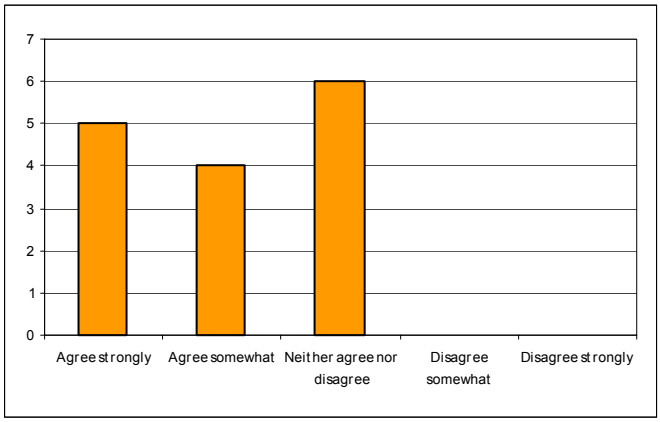


Figure 4-9 Response of 15 users to the proposition:
'The system makes me feel more empowered as an individual.'

In any event, the responses to the final question, shown in Figure 4-10 are the most conclusive. 15 out of 15 interviewees agreed with the statement that 'The new system contributes positively to my job satisfaction', ten of them strongly. Such a reaction in relation to a core transactional business system is probably quite rare.

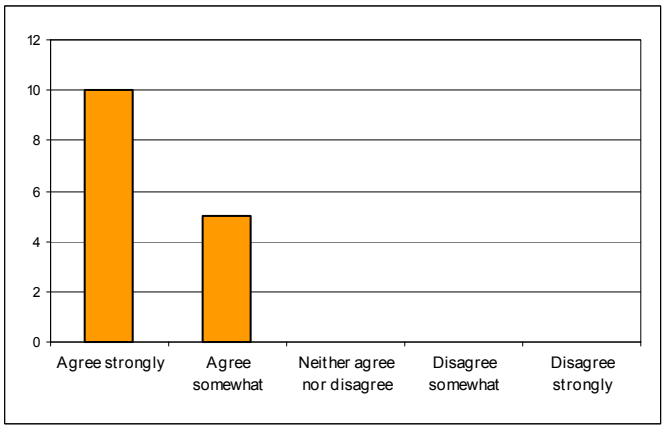


Figure 4-10 Response of 15 users to the proposition:
'The new system contributes positively to my job satisfaction'

4.7.2 The system is efficient from a business viewpoint

The business goals for the new system focused heavily on improved agility. Increased throughput was never an explicit goal. Indeed, the management recognized that adopting naked objects might mean that the new system might be slightly slower for straightforward tasks than a more conventional system based on scripted interactions (as the older system had been). However, as Figure 4-11 shows, now that the system is implemented the perception of users is that it is faster for most tasks.

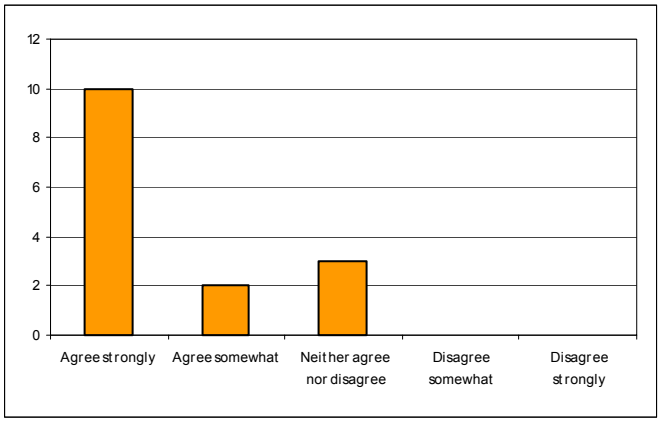


Figure 4-11 Response of 15 users to the proposition:
'The new system allows me to process most claims and enquiries faster than before.'

At the time when the new system went live (November 2002) the Child Benefit department had a backlog of more than 20,000 claims awaiting processing, such that a new claim was taking up to 6 weeks to be processed. By the end of February 2003, this backlog had been virtually eliminated, and the department was approaching its goal of turning round most applications within a week. Clearing the backlog required a massive short-term use of overtime labour - it cannot simply be attributed to the efficiency of the new system. Nevertheless, the management believes that even though the new system was still 'bedding down' in January and February, it was still a very positive factor in helping to clear the backlog.

4.7.3 It is too early to assess the strategic agility of the resulting systems properly, though initial signs are positive

Strategic agility - one of the DSFA's explicit goals for the new system - is defined as the ability to modify the system easily to accommodate future and unforeseen business change. This capability is not going to be fully tested until the system has been through several cycles of business change, which will take several years. As shown in Figure 4-12 and Figure 4-13, most IT managers remain convinced that this agility will be demonstrated in future, and two of the three business managers said that they could envisage how possible future business changes could be realized through the business object model.

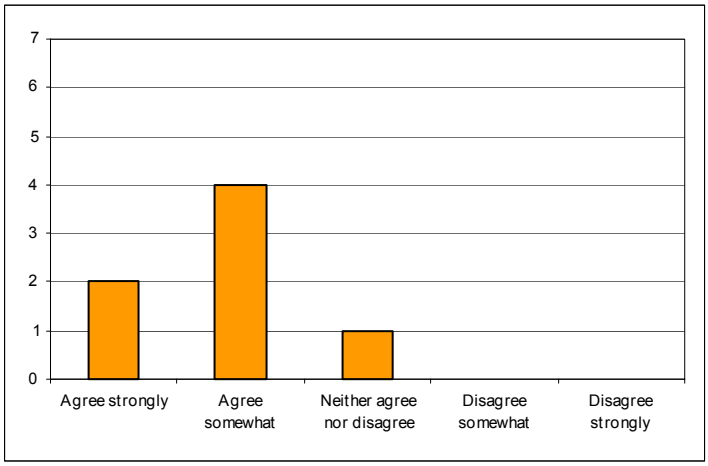


Figure 4-12 Response of 7 IT managers to the proposition:
'To the extent that either of these forms of agility have not yet been demonstrated, our expectation that they will yet be demonstrated remains as strong as at the start of the project'

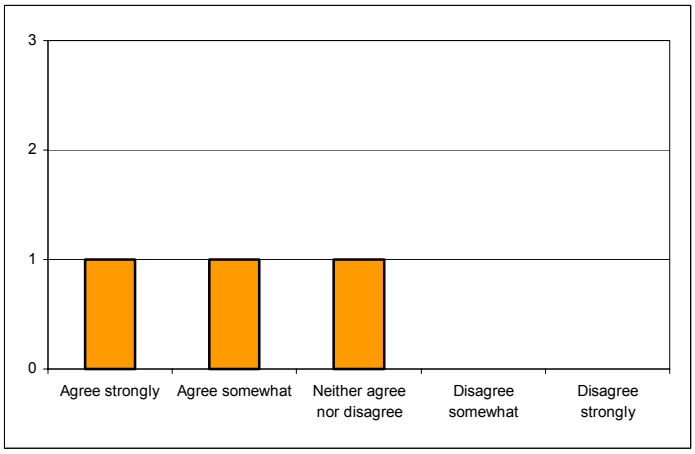


Figure 4-13 Response of 3 business managers to the proposition:
'I can envisage how a range of possible future business changes might be realized through the object model'

However, as shown in Figure 4-14, four of the IT managers believe that the system has already demonstrated the ability to support this strategic business agility.

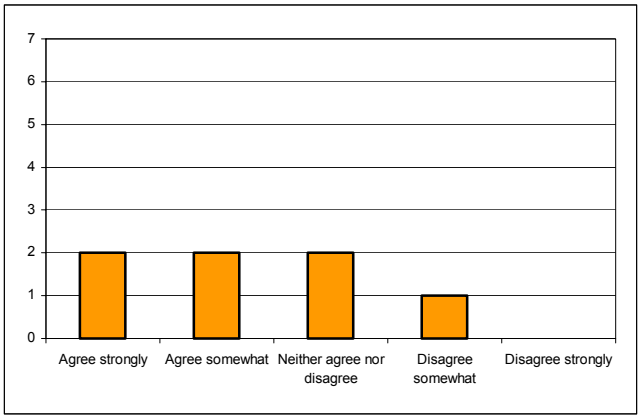


Figure 4-14 Response of 7 IT managers to the proposition:
'The system has already demonstrated the ability to support strategic business agility'

Based on comments from those and other managers and on direct observation, there are three pieces of evidence to support that view.

The first is that the business requirements changed during the development of the CB system. This is likely to happen on any substantial business system project, but several managers stated that it proved somewhat easier to accommodate these changing requirements, even quite late in the project, than they would traditionally have expected. As one put it: 'On several occasions, the business sponsors asked if it would be possible to accommodate a certain new requirement, fully expecting that the answer would be "no". They were surprised when the answer was "yes".'

Second, when the new system was introduced in November 2002, the opportunity was taken to introduce some broad organizational changes, of a scale and nature that could justifiably be called business reengineering [45]. Previously, each claim was handled by three separate people. One registered the incoming claim, entering the data from the paper claim form into the system. Another officer 'decided' the eligibility of the claimant and the amount of benefit (entitlement) that should be paid. A third officer put the claim into payment. The motivation for this arrangement was the specialization of labour, and reducing the risk of internal fraud. The deciding officers were further specialized according to the type of the claim or action required, with officers specializing, for example, in handling '16+ extensions' (when a child stays in education above age 16).

The new organization introduced the concept of 'once-and-done'. Now most claims are processed, from acceptance to payment, by a single officer in a single session. This has been hugely popular with the staff (there are many positive references to it in the user survey responses shown in Appendix V). It is also worth pointing out that this was only introduced because the department's auditors were satisfied that the design of the new system actually provided more protection against possible fraud than the old one.

Additionally, instead of being managed as specialized units, the officers have been reorganized into five teams, each of which will be able to process all types of claim and query. Currently, there are still specialized roles within the teams, but with an ongoing programme of training, officers will increasingly be able to handle all types of claim within CB - and there are longer-term discussions about the possibility of well-qualified officers being able to handle multiple Scheme types.

These organizational changes could, in theory, have been implemented on the old system with massive retraining, although the result would have been very awkward in practice. It could certainly have been achieved had the new system been conventionally designed to suit the new process. However, the striking thing about the contribution of naked objects is that the concept of 'once-and-done', while vaguely recognized as a future possibility during the object modelling, was never specified as part of the system requirements. Indeed there was some doubt about the acceptability of the concept to various stakeholders. Yet the manager of the SDM team that took the decision to implement once-and-done, and to start the reorganization into multi-skilled teams, reported that nothing in the design of the system got in the way of this planning. That would almost certainly not have been the case with a conventionally-designed system.

The third major piece of evidence for strategic agility is that having gone into production, the CB system is already being subject to its first significant business change. The Irish General Register Office (GRO), responsible for registering births, deaths and marriages, has recently commenced a massive modernization programme, which includes introducing the generation of electronic certificates. The DSFA is collaborating with the GRO to exploit this capability. The idea is that whenever a mother who has already been deemed eligible for CB has another child, the act of registering the birth of that child (which can happen on-line at the hospital) will automatically add the new child to the existing claim and adjust the monthly payments accordingly. This is a radical change to the existing procedure. It has yet to be implemented, but management at the DSFA has already worked out that this capability can be introduced with only very modest changes to the CB system as implemented.

4.7.4 On-line performance has turned out well within the response times specified, but performance for batch processing is a significant concern.

During the early stages of the project there were significant concerns that the on-line performance would be slow, given that the client and server platforms are separated by a wide area network. However, on-line performance has proved to be well within specified response times.

When it first went live batch processing was a significant problem, with some batch processes taking twice as long as the specified performance level, even after some performance tuning. The DSFA's batch processing requirements are demanding: not only do they involved processing large numbers of customers (half a million for CB alone), but processing each customer involves

significant calculation. For the monthly electronic funds transfer batch process the system must calculate how much benefit is payable to each customer individually, allowing for intervening birthdays and other events. The intent of the *Naked Object Architecture* was that the batch processes should invoke this functionality on the core business objects, instead of duplicating it in dedicated batch scripts or stored procedures This means that each batch transaction may require the instantiation of many business objects, each instantiation incurring a significant processing overhead. .

In a review of the problem in June 2003 (in which the author participated) it was concluded that the performance problem could be avoided by arranging for the Scheme object to calculate several future payments at a time - which could easily be over-ridden if there is a change in the customer's circumstances. (This approach is necessarily the one taken for the printing of Post-Office Payment Order books). For most customers being paid by EFT, the monthly batch process would then simply be turning a predicted payment value into an actual Payment object, thereby significantly reducing the number of instantiations and other methods invoked for each transaction.

Another area of concern is the use of association tables in the relational database, which were adopted in order to support the many-to-many relationships between objects, and to provide greater agility. The CB system alone has 95 million rows in its association tables and it is perceived that this approach will not scale up from the CB system to larger applications such as the Pensions Administration system. Initial investigations suggest that the contractor was too generous in providing for possible future associations between any potential objects. In practice the need for associations between object classes is quite well known and this provision could be considerably scaled back.

The overall conclusion from these issues is that the *Naked Object Architecture* has not generated any performance and scalability problems *per se.* However, because it encourages a purer approach to object-oriented design, it can bring into sharper focus issues such as the mapping of objects onto relational databases. These issues are very well known and solutions exist (see, for example, [41]) but the naked objects approach may force more attention than usual to be paid to them .

4.7.5 Because the DSFA had to develop the whole architecture as well as the CB application there was no gain in the overall development cycle, but this is expected to change with future applications

The elapsed time from the contract being awarded to the new system going live was originally intended to be 12 months and was in fact 21 months. Even taking into account some unrelated delays (such as industrial action) it is unlikely that this was faster than could have been achieved using a more conventional approach to the development of a brand new client-server system.

However, this is understandable given that for the CB system the DSFA had to commission the development of a brand new architecture as well as the first application to run on top of it - the latter being dependent upon the former for many aspects of its development and testing.

The DSFA, and to an even greater extent the contractor, had to learn a great deal about the implementation and application of naked objects. The methodology used by the contractor did not take best advantage of the capabilities of the naked objects approach, nor, arguably, did it address all of its needs. (This theme will be picked up again in Chapter 5).

Nevertheless, as shown in Figure 4-15, most of the IT managers believe that the development of subsequent applications should be faster now that many elements of the intended architecture are in place and experience is growing and being codified into an appropriate methodology. Some commented that this may not show up even in the planned Phase II application, but will in subsequent phases.

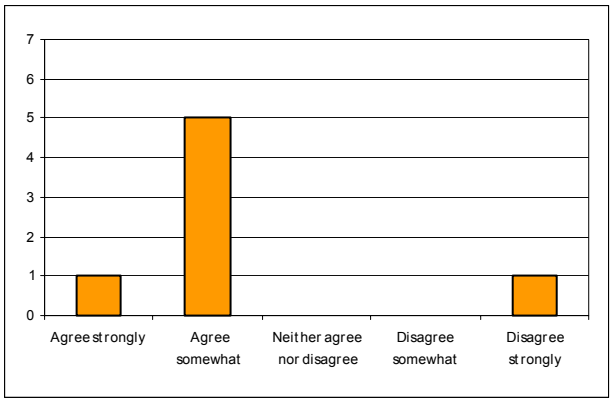


Figure 4-15 Response of 7 IT managers to the proposition: 'My expectation is that any subsequent business system developed on the [*Naked Object Architecture]* will be developed faster than achievable using a more conventional approach'

4.7.6 Naked objects did facilitate communication between developers and users, but greater use should have been made of prototyping

Figure 4-16 shows that the IT managers believe that the use of naked objects directly facilitated communication between developers and users. Although this question was not directly asked of the 3 business managers who had been involved in the process, their answers to a broad set of overlapping questions (see Appendix IV) could be deemed to support this point.

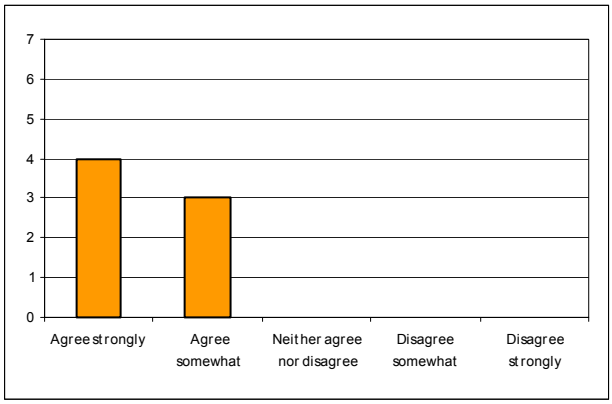


Figure 4-16 Response of 7 IT managers to the proposition: 'The [naked objects] approach to designing the system directly facilitated communication between the developers and the users'

However, as shown in Figure 4-17 and Figure 4-18, both groups felt that the process could have benefited from more prototyping. In recorded comments, several individuals expressed disappointment that the prototyping approach used during the early exploratory work was not continued into the main development phase (which was by a different contractor). The implication of those comments is that the combination of prototyping with naked objects would improve communication further. This theme will also be picked up again in Chapter 5.

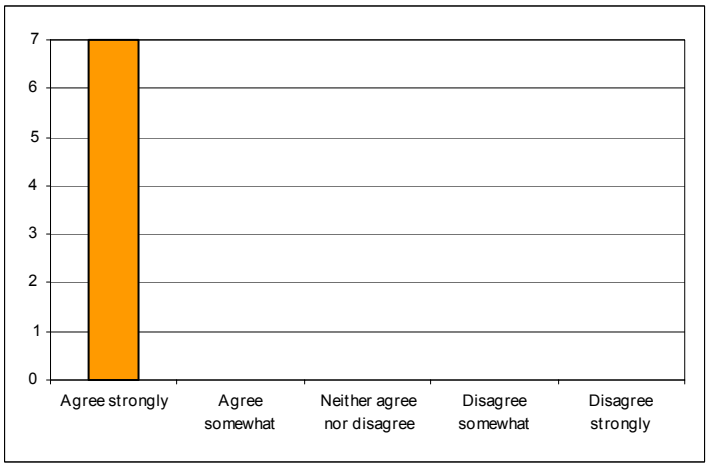


Figure 4-17 Response of 7 IT managers to the proposition: 'The process could have benefited from greater use of prototyping'

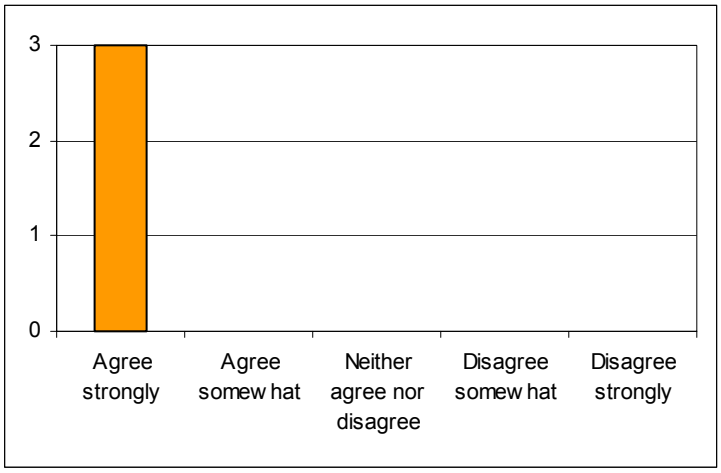


Figure 4-18 Response of 3 business managers to the proposition:
'I would like to have seen more use made of prototyping to test out the business scenarios'

4.8 Conclusions from this case study

The DSFA's new CB application running on the *Naked Object Architecture* clearly demonstrates the primary characteristics of naked objects:

- All business functionality is encapsulated into behaviourally-complete business entity objects, the responsibilities of which are shown in Appendix II. There is no form of business scripting or use-case controllers sitting on top of those objects.
- Those objects are exposed directly to the user in the form of a truly object-oriented user interface (OOUI). At a subjective level, the CB system delivered (as shown in Figure 4-5 Screenshot from the new Child Benefit Administration system showing the naked objects) looks and feels remarkably like the very first PowerPoint mock-up (Figure 4-2), which was written in February 1999 after just two days of workshops on object-oriented techniques.
- The presentation layer is agnostic with respect to the business object model definition. The object views and actions presented to the user are generated dynamically from the business objects themselves at run-time.

Although there were some problems with the project, and there remain some technical issues to resolve, overall the approach has been deemed successful. Many of the answers to the questionnaires point to this. The strongest evidence of its success is the decision to proceed with further, larger-scale systems based on the same approach.

Comparing the findings of the evaluation with the benefits proposed in Chapter 3 suggests that the DSFA case study provides:

- Strong evidence of increased usability/operational agility

- Some evidence of increased strategic agility
- Some evidence of improved communication between users and developers, but a feeling that this required more prototyping
- No evidence of a faster development cycle, although this can be attributed to the demands of creating the architecture in parallel with the first system, and the mismatch of the methodology to the approach.

CHAPTER 5 GUIDELINES FOR DESIGNING NAKED OBJECT SYSTEMS

The early work on the DSFA's business object model and its application to Child Benefit Administration was conducted without any formal methodology or written guidelines. It relied heavily upon the author's commitment to the concept of naked objects, and experience gained from other exploratory applications conducted over the same period using the *Naked Objects* framework. (Some of these other exploratory exercises are described in [86]).

It has long been acknowledged that good object modellers do not need formal guidance for identifying objects and defining their responsibilities [97]. And it can be argued that any attempt to use step-by-step guidance to reduce the need for object modelling skills is in the long run likely to be detrimental. It is also worth pointing out that there is a general move away from formal heavyweight methodology towards more agile techniques [2] and even a growing argument that heavyweight methodologies are largely a fiction [78] and even that software development does not necessarily require any formal methodology at all - so-called 'amethodological development' [116].

Nevertheless, there are also good arguments for providing some formal guidance for the design of business systems using naked objects. It is clear, for example, that later on in the DSFA project, when the development was being conducted by a third party contractor, the project suffered from not having any such formal guidance. The contractor used its own 'waterfall' style methodology throughout the project. Several managers at the DSFA have since expressed the view that the contractor's methodology did not fully exploit the positive advantages of naked objects, nor fully address its particular needs.

The negative impact of this on the delivered product was probably small, if only because the *Naked Object Architecture* enforced a commitment to naked objects. However, the negative impact on the development process was substantial. Several of the DSFA managers felt that a more appropriate methodology would have saved not only time and effort, but also a considerable amount of frustration. Figure 5-1 and Figure 5-2, extracted from the survey of DSFA IT Managers (Appendix III) show strong support for the view that the process could have benefited from greater use of prototyping, and preference for a more iterative approach to delivery than was used.

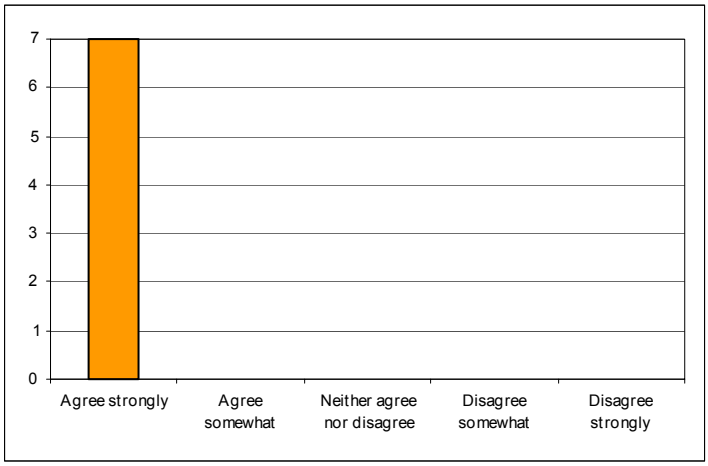


Figure 5-1 Response of DSFA's IT managers to the proposition: 'The [development] process could have benefited from greater use of prototyping'

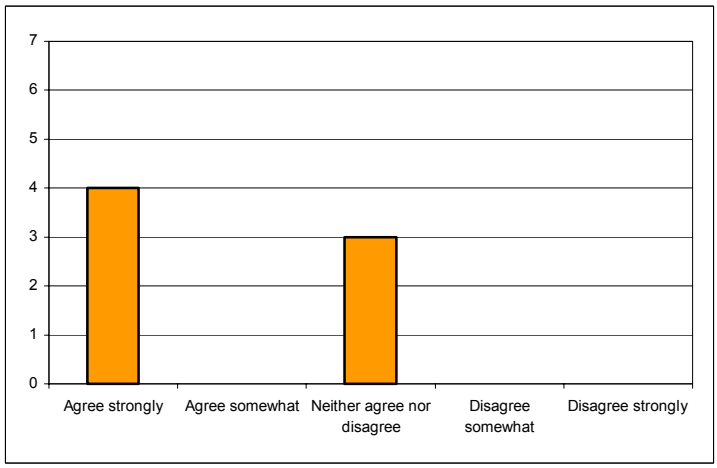


Figure 5-2 Response of DSFA's IT managers to the proposition: 'The [development] process could have benefited from a more iterative approach to delivery'

Further evidence supporting this interpretation can be found in the DSFA's Request for Tender (RFT) for the second phase of application development using the *Naked Object Architecture* (intended to replace the existing pensions administration system). This excerpt is from the section of the RFT entitled 'Required approach to development':

> *'The Department's experience of software development in the [Naked] Object environment has led it to conclude that the most suitable development approach should include two techniques:*
>
> - *Extensive use of prototyping to assist in the definition and refinement of the Business Object Model.*

- *Delivery of functionality in an iterative, incremental fashion with the emphasis being on regular iterations delivering increasing functionality.*

It is expected that the Phase 2 development project will make extensive use of both techniques.'

In response to the DSFA and other experiences, a set of guidelines for structuring and managing a development project using naked objects have been developed. Seven broad guidelines have emerged:

- Look for projects with characteristics that will benefit most from using naked objects
- The pre-requisites for starting a naked objects project are: good OO modelling skills, a suitable software framework, and a common understanding of the intent
- Structure the project in two distinct phases: exploration and delivery.
- During exploration, identify objects and their responsibilities directly, not from use-cases
- During exploration, capture the object definitions directly into working code
- Develop the production system from scratch, and one scenario at a time
- During the delivery phase capture the scenarios as executable acceptance tests

The remainder of this chapter expands upon these seven guidelines.

5.1 Look for projects with characteristics that will benefit most from using naked objects

One of the benefits proposed in Chapter 3 is that a user interface consisting of naked objects treats the user more like a problem-solver and less like a process-follower. Any business problem where this would be seen as a benefit is therefore a good candidate. Systems at the customer interface are a good example. Pricing and promotions, resource planning, trading, network operations, and risk management are also examples of intense operational activities that demand a problem-solving approach. Any business activity that fits the value shop [111] model of business (as distinct from the value chain [89]) - such as project management, emergency response, campaign management - is also a candidate.

The naked objects approach is also well suited to situations where requirements are uncertain or likely to change during the course of the project, because it facilitates communication between developers and users.

It can be argued that most business systems projects fit into one of two broad categories. Some are dominated by engineering considerations because of their very large scale or transaction rates - examples include a credit card transaction clearing system, an airline reservation system, or a telecommunications billing system. It is generally accepted that in order for the appropriate engineering disciplines to be properly applied, it will be necessary to specify the requirements early on in the project. It is generally accepted that subsequent changes to these requirements will prove very expensive, and therefore considerable emphasis is placed on specifying them correctly including the prediction of future growth in usage. For this type of project, the traditional, heavyweight, 'waterfall' style of methodology is acceptable and, arguably, necessary. It is also worth observing that this is precisely the kind of problem that those methodologies were designed to address. Jacobson's Object-Oriented Software Engineering [56], for example (much of which has now been embedded in the Unified Process [57]) arguably reflects its heritage in the design of telephone exchanges.

For many other new business applications the engineering considerations are straightforward. These projects are often dominated by requirements considerations: which are very likely to change during the course of the project, either because of a rapidly evolving business context, or as a result of customer reaction to prototypes or early iterations of the system. The naked objects approach is well suited to this second category i.e. requirements-dominated projects.

Limitations

Experience gained from the case studies described in other chapters suggests that naked objects do have some potential limitations, and these should be considered when deciding whether to use naked objects in a particular project.

The first suggested limitation is that because the user interface is auto-generated, there is no scope for hand-designing the user interface: either to optimize it for the execution of a particular task, or to customize it to the needs of an individual user. In some circumstances, this can be seen as a benefit. In the words of one interviewee from Safeway (discussed in Chapter 6) 'The naked objects approach stopped us from focusing on all the irrelevant things that we normally focus on in requirements analysis.' One could also argue that the disadvantage of not being able to hand-craft the user interface is more than offset by the greater expressiveness offered by the auto-generated object-oriented user interface.

Nevertheless there are circumstances where a hand-crafted user interface is necessary. These include safety-critical user interfaces, disability access, highly graphical or spatial applications, and where typographical design is considered critical, say, for marketing reasons. In these circumstances naked objects could be used to prototype the business object model, and then a conventional user interface designed and built to interface to those same objects. The project

would still gain some of the other benefits of having used naked objects initially - such as the improved object modelling.

Another option is to make the user presentation somewhat customizable by the end-user directly, without involving any programming. For example, frequently-used options on pop-up menus could be dragged into an object to appear as a button. The user configuration would be remembered in that user's profile, to appear next time the object is opened. IBM's original work on the *Common User Access* included this concept [52].

The second limitation is that naked objects applications may take longer for the user to learn - because they lack the formal, scripted, guidance of a conventional system. There are ways to add user prompting to naked objects applications without compromising the approach. Nevertheless, naked objects systems are better suited to frequent-use applications, where the greater flexibility offered by naked objects is more important than the ability to use the system with no training or learning period. Thus customer self-service applications (where usage is infrequent) are not typically good candidates for using naked objects: they require a more conventional approach where the system guides the user through a task (and for which the marketing department will almost certainly demand tight typographical control anyway). Exceptions might be self-service applications that are used frequently by the same customers, such as on-line grocery shopping, personal financial management, or travel planning for frequent flyers.

The third possible limitation is in the area of batch processing - as encountered by the DSFA (page 52). These batch-performance issues arise from the relationship between the business object layer and the relational database, not directly from the naked objects approach, which is concerned only with the relationship between the business objects and the presentation layer. However, because naked objects encourage (even force) a purer form of object modelling, they perhaps bring the persistence mapping issues into sharper focus. If naked objects are to be used in systems that involve a substantial element of batch processing, and the desire is to keep all business functionality on the objects, then further work is going to be needed on persistence-mapping patterns for naked objects.

Given these benefits and limitations, the naked objects approach is best deployed in systems where <u>any</u> of the following three statements is true:

- There would be benefit from characterizing the role of the user as a problem-solver rather than a process-follower
- Future business agility is a primary concern
- Requirements are uncertain

and <u>all</u> of the following three statements are true:

- There is no clear rationale for having a hand-crafted user interface
- The users are likely to be frequent users
- Any batch processing is relatively simple in nature, or can be treated as a separate system.

5.2 The pre-requisites for starting a naked objects project are: good OO modelling skills, a suitable software framework, and a common understanding of the intent

There are three pre-requisites for starting a naked objects project. The first is that there exists within the team someone with strong object modelling skills. There is evidence to suggest that using naked objects can help to improve participants' understanding of object-oriented techniques (see, for example, the last chart in Appendix VII), and can encourage better object modelling, but it is still necessary to have at least one individual with good experience in object modelling to help the team to identify objects and their responsibilities.

The second pre-requisite is that the team has access to a software framework designed to support the concept of naked objects (such as the DSFA's *Naked Object Architecture* or the *Naked Objects* framework). The team must have developers who are experienced in using the framework. Attempting to learn the framework at the same time as learning object modelling, or to develop a new framework as part of the project, is possible (as demonstrated by the DSFA) but is likely to hinder the realization of the benefits of using the naked objects approach *on that project*.

Third, it is necessary that everyone participating in the project, be they developers, project managers, user-representatives or other business stakeholders, has some level of understanding of the concept of naked objects. (This is an unusual pre-requisite: when a system is being designed according to, say, the Model-View-Controller pattern or the generic 4-layer architecture, there is no requirement for any business representative to have any understanding of those patterns.) Beck has argued that all projects should ideally have a metaphor that guides the overall design, some common examples being that the system will function like a spreadsheet, or a production line [9]. Naked objects provide a ready-made example of such a metaphor. The best way to establish this metaphor in the minds of participants is to give them all a chance to see, and ideally to use, an existing system that has been developed this way.[7]

It is possible to refine the metaphor further, for example describing the system by analogy to the Windows (or other equivalent) 'desktop', to a 'workbench', to a 'drawing package', or to a

[7] Several such examples are provided on www.nakedobjects.org.

'problem solving system'. A good example is the DSFA's use of *The Incredible Machine* as a metaphor for its system (page 37). This consumer product is still used by the DSFA to introduce new IT staff to the *Naked Object Architecture*.

5.3 Structure the project in two distinct phases: exploration and delivery

The advantages offered by naked objects will be best realised if the project is structured in two distinct phases: exploration and delivery. (The relationship between these concepts and other methodologies will be addressed in Chapter 8.)

The exploration phase, which is conducted by a small team of developers and users (and/or other business representatives) comprises both object modelling and prototyping. A unique feature of naked objects is that these two activities become strongly synergistic. Conventionally, development teams must choose between a screen-based prototyping approach to capturing requirements, and a more abstract object-modelling approach. Naked objects fuse these two activities together, and deliver the benefits of both. The object modelling encourages the team to explore alternative representations and to build in sufficient abstraction to facilitate future changes, while the concreteness of the working prototype provides real-time feedback on practicality - as well as stimulating new suggestions in its own right. It is quite possible that such a period of exploration will result in a change to the intended scope of the system.

For this reason it is recommended that the exploration phase be strongly time-boxed - otherwise the exploration activity could extend indefinitely. (Time-boxing also permits the exploration phase to be fixed price). Within the time box, the team should be able to take its own decisions concerning the breadth and the depth to which each idea is explored.

By the end of the exploration phase, the team will have established an outline object model of the business domain, and implemented it in the form of an executable prototype that has been used to test a variety of business scenarios. All of this can be achieved using relatively little resource, and thus forms a very effective basis for deciding whether to proceed with a the development and implementation of the system i.e. whether to conduct a delivery phase.

For the delivery phase it is recommended that the system be coded from scratch. This is partly to avoid perpetuating simple coding errors that may have been overlooked during prototyping. More generally, though, exploratory prototyping and development of a production system require quite different styles of development. The former works best with an 'optimistic' style of programming - where it is assumed that the user will always do the right thing and that all data is correct - and the latter demands a more 'pessimistic' style, where as many error conditions as possible must be anticipated. And as will shortly be argued, whereas use-cases potentially provide an effective

structure for managing the delivery phase, they should not be used to identify the objects in the exploration phase.

Of the remaining four guidelines, the next two relate to the exploration phase, and the final two to the delivery phase.

5.4 During exploration, identify objects and their responsibilities directly, not from use-cases

The use-case driven approach, formalized by Jacobson [56], is now so well established that it is rarely questioned. From a business perspective, specifying systems as scripted procedures fits well with the ideas of task optimization and efficiency maximisation, concepts first espoused by Frederick Taylor in the late nineteenth and early twentieth century [61, 114]. The limitations and negative social consequences of translating Taylor's approach into the modern world of information processing have been clearly described by Garson [43], Brown [16] and others, but there is no doubting that it is still very popular.

Andersen criticizes the use-case driven approach [7] from the perspective of interaction design. Shah makes a more general critique [102]. Firesmith, however, attacks use-case driven approaches specifically in terms of their negative effect on object modelling, which is the primary concern of this thesis:

> *'Use cases are not object-oriented. Each use case captures a major functional abstraction that can cause numerous problems with functional decomposition that object technology was supposed to avoid . . . . Since they are created . . . . before objects and classes have been identified, use cases ignore the encapsulation of attributes and operations into objects . . . [A use-case driven approach results in] the archetypal subsystem architecture . . . a single functional control object representing the logic of an individual use-case and several dumb entity objects controlled by the controller object . . . Such an architecture typically exhibits poor encapsulation, excessive coupling, and an inadequate distribution of the intelligence of the application between the classes'.* [39]

Berard makes a similar argument in [14]. Given that most OO practitioners do not seek to achieve behaviourally-complete (or even behaviourally-rich) entity objects, these concerns may seem somewhat irrelevant. However, when attempting to use naked objects the issue comes sharply into focus: starting the analysis by specifying use-cases does not lead naturally to the identification of behaviourally-rich entity objects, and is much more likely to encourage the traditional separation of procedure and data.

Meyer supports this view that use-cases are not a good tool for finding classes, and suggests that relying on them in any significant way raises three risks:

> *'Use cases emphasize ordering . . . This is incompatible with object technology, [which] shuns early reliance on sequential properties, because they are so fragile and subject to change. The competent O-O analyst and designer refuses to focus on properties of the form "The system does **a** then **b**"; instead he asks the question "What are the operations available on instances of abstraction **A**, and the constraints on those operations." '*
>
> *'. . . the system picture that use-cases will give you is based on existing processes, computerized or not.'*
>
> *'Use cases favour a functional approach, based on processes (actions). This approach is the reverse of O-O decomposition, which focuses on data abstractions.'* [74]

It is worth noting, however, that Jacobsen saw use-cases as serving two roles. The first was for identifying objects, which role has been criticized above. The second role was testing the resulting system:

> *'The use cases constitute an excellent tool for integration test since they explicitly interconnect several classes and blocks. When all use cases have been tested (at various levels) the system is tested in its entirety'* [56].

Meyer emphasizes this second role:

> *'[use cases] remain a potentially valuable tool but their role on object-oriented software construction has been misunderstood. Rather than an analysis tool, they are a **validation** tool.'* [74]

Using use-cases for validation is entirely compatible with naked objects, and this theme will be addressed again in the context of the delivery phase.

In the context of the exploration phase, however, if starting the analysis with use-cases is not recommended, how are the business objects to be identified? The answer is through direct and unstructured conversations between the users and developers. Rosson demonstrates that, given such a context, good object modellers are able to identify the objects directly without the need for other formal artefacts [97].

Having identified object candidates, behaviours are best identified by modelling the high-level responsibilities that could naturally be associated with each object, for example as advocated in the approach called Responsibility Driven Design [120].

5.5 During exploration, capture the object definitions directly into working code

During the exploration phase it is recommended that object definitions be captured directly into working code i.e. actual classes in the programming language being used. With an object-oriented programming language (such as Java, C#, or Smalltalk) a new domain class can be added in seconds, as can new attributes. If the programming is done within a state-of-the-art development environment that supports macros, coding patterns and idioms, then it is possible to define associations between object classes, including many-to-many and bi-directionally navigable associations, almost as quickly.

Most business system development projects do not attempt to capture the business object model directly into code because such a definition would be almost unreadable by any non-programmers involved in the analysis process. However, the naked objects approach (if supported a suitable framework such as the *Naked Objects* framework) means that business objects can immediately be rendered visible from the perspective of the user, typically in iconic form. Opening a view of an object will immediately show its attributions and associations; right clicking on an object will reveal its business methods as a pop-up menu. From the perspective of users or other business representatives, this representation is far more concrete than any diagram or textual form. Moreover, it is immediately possible to create instances of the business objects and start manipulating them to simulate actual business scenarios. (The ability to drag and drop instances also helps the user to build what the cognitive psychologist Jerome Bruner calls an 'enactive' mental representation of the domain [18] to complement the symbolic and iconic (visual) representations. Bruner's theory on representation was one of the inspirations for Alan Kay's work at Xerox Parc [62] - discussed in Chapter 2).

Naked objects and UML

However, there are limitations to viewing the object model solely through a naked object user interface. For example, sub-classing is not obvious. Nor is it easy to view the constraints on associations such as their cardinality. These things are readily visible if the object model is rendered into the 'class diagram' form of the Unified Modelling Language (UML) [99]. The class diagram is only one of several different representations provided by UML, including state transition diagrams and sequence diagrams to show dynamic behaviour. The latter can sometimes clarify complex interactions, although the working prototype can be a more effective way of viewing some of the more straightforward behaviours of the system.

However, the advantages of these alternative forms of visual representation must be weighed against the risk that multiple representations (or indeed multiple forms of documentation of any kind) get out of synchronisation with each other.

There are now many development tools that are capable of generating one form of representation or documentation automatically from another, including the ability to generate program code from UML diagrams[8]. However, the majority of these tools translate in one direction only. If the generated program code is modified directly - and as a project proceeds this is increasingly likely - then it becomes inconsistent with the UML diagrams. Not only does this hinder developers navigating the system, but subsequent change to the UML diagrams may not be implementatable without losing the direct code modifications.

The long term goal for these tools is to achieve 'round-trip engineering', meaning that any change to the model in diagram form would change the code and vice versa. One of the first tools to demonstrate this is the Togethersoft Control Centre[9]. What is unique about this tool is that the working code is the primary representation. Even when the user is working in a UML class diagram representation, all user actions are translated directly and immediately into code and then translated back into diagramatic form - all transparently to the user. Thus although the Togethersoft Control Centre superficially resembles a modelling tool, it is fundamentally a sophisticated code development environment. It also has very good support for coding macros, patterns and idioms.

In April 2002 the author facilitated a workshop on the naked objects approach at the OT2002 conference in Oxford[10], using the *Naked Objects* framework. After some initial training on using the framework (conducted by Robert Matthews) the 20 participants were invited to suggest a simple application that they would attempt to model and then develop in four separate teams. The application chosen was a conference management system, where the domain model included classes for Conference, Venue, Session, Room and so forth. In the two hours made available for the task, all four of the groups developed a simple working prototype and most had gone through multiple iterations of the model.

Each of the four groups used the *Naked Objects* framework from within their own choice of development environment installed on their own laptops. The most productive of the four groups by a significant margin had used Togethersoft Control Centre. One of the participants, Dan

---

[8] Examples include Rational's Rose (www.rational.com), Interactive Objects' ArcStyler (www.arcstyler.com), and Kennedy Carter's iUML (www.kc.com).

[9] www.togethersoft.com

[10] See http://www.ot2003.org/scripts/wiki/ot2002/?KkjjjedfhdhpghnasbisdunetnetKk

Haywood (co-author of [20]) was an expert in this tool, and at the start of the project wrote a few simple macros to support the simple coding conventions required by the *Naked Objects* framework. The object model was then entered and edited directly as a UML class diagram, one version of which is shown in Figure 5-3.

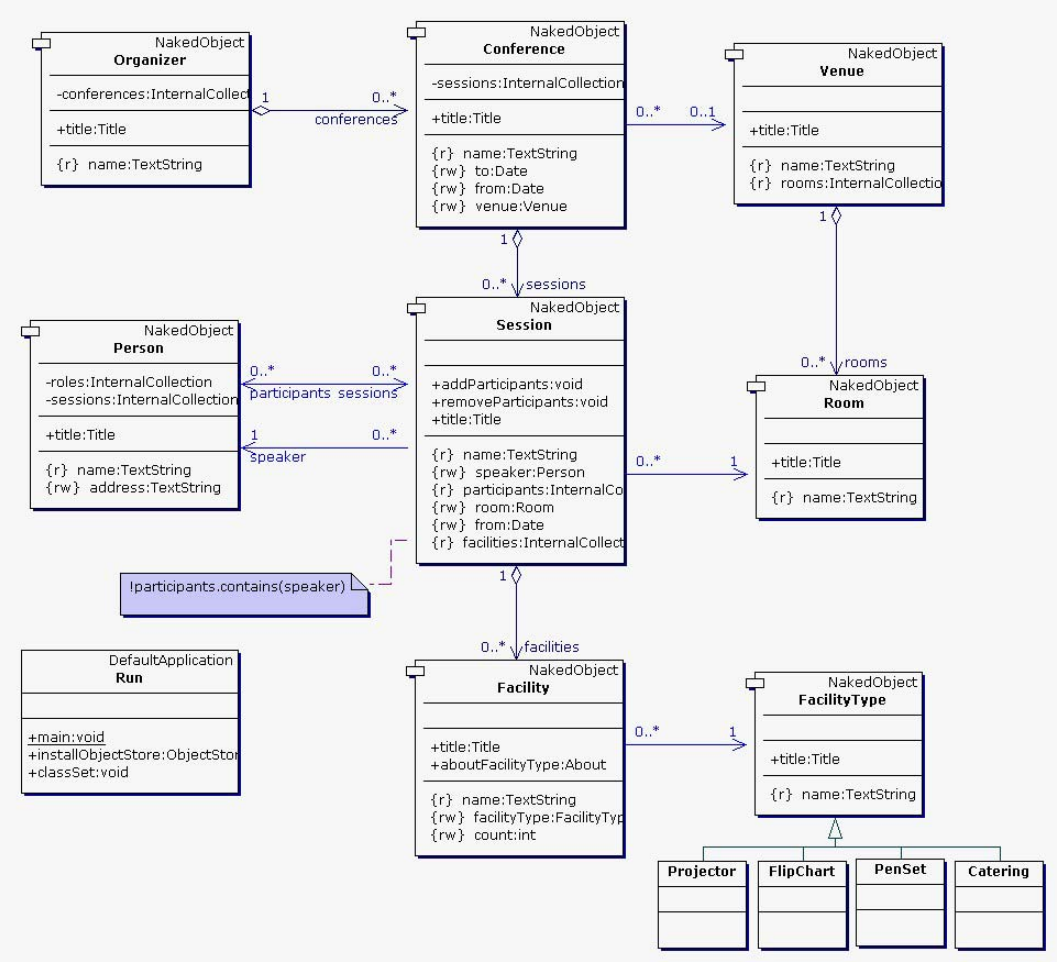


Figure 5-3 UML class diagram showing the business object model for the conference planning system

Behind the scenes the Togethersoft tool created all the code necessary to build a running *Naked Objects* prototype, one screenshot of which is shown in Figure 5-4.

The first iteration of the working prototype was thus created without the team having to write a single line of Java code. The functionality was enriched in subsequent iterations, and in several cases this involved writing business methods directly in Java. However, as these methods were written, their method signatures immediately became visible in the UML representations also.

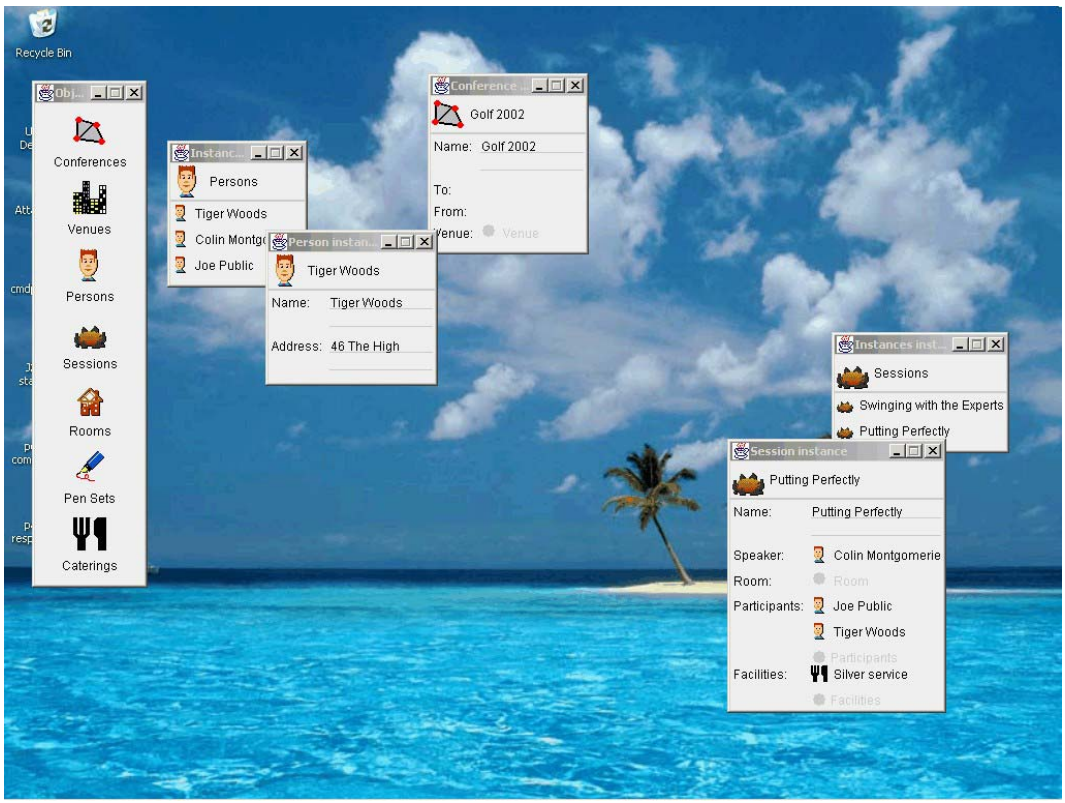


Figure 5-4 This naked object application was auto-generated from the UML diagram shown in the previous figure.

Counter arguments

There are two principle counter arguments to the idea of capturing the business object definitions directly into working code:

- It might discourage abstraction in the model
- It forces a commitment to a specific programming language from the outset

Expanding on the first of these points, it could be argued that though the visual concreteness of the approach described above undoubtedly helps in communication, it could discourage the team from building sufficient abstraction into the model. For example, it could be argued that the developer will be encouraged to focus on concrete concepts such as Customer instead of more abstract concepts such as TradingParty. However, the idea that more abstraction is always better is by no means universally accepted. George Lakoff has argued that there are natural levels of abstraction in most people's minds [68]. It can be argued that the naked objects approach helps to focus on these natural levels of abstraction. Further levels of abstraction can and should be achieved through the definition of interfaces rather than classes themselves (the distinction being that one class can be made to implement multiple interfaces). Thus, if a Wholesaler class was subsequently added, both could be made to implement a TradingParty interface without having to change the class hierarchy. Certainly the experience of the various case studies presented in this thesis (and in

[86]) show very good evidence of abstraction in the object model - both by the small number of core business classes in each resulting model and the ease with which they could be modified.

Another way in which poor abstraction might be observed is in the definition of the responsibilities. The ease with which attributes and associations can be captured directly into code, compared to the specification of a behaviour, might encourage the team to focus on the 'know-what' responsibilities of an object, instead of the 'know-how-to' responsibilities - something that Wirfs-Brock[121] and others warn against. However, in this regard UML is no better than using code: it also potentially encourages emphasis on the attributes and associations rather than the abstract responsibilities - evidenced by the fact that UML is used for conventional data modelling as well as object modelling. The use of CRC cards [11] arguably does encourage emphasis on the 'know-how-to' responsibilities, but it is a simple and limited notation.

However, it can also be argued that capturing business object definitions directly into a working naked objects prototype can help in the abstraction of responsibilities. First, because more than any other approach to object-modelling, the naked objects approach encourages behavioural completeness - the business objects have to fulfil know-how-to responsibilities because there is nowhere else to locate business functionality. Secondly, even if it is not possible to specify the implementation of a higher level responsibility at the outset, many ideas can be immediately captured as method signatures and rendered visible to the user. For example, it might be conceived during exploration that one of the responsibilities of a Customer object would be to be able 'to communicate with the (real) customer using the customer's preferred medium (email, letter, fax etc)'. Given the work needed to specify this behaviour in full, let alone to code it, further consideration is likely to be deferred to the delivery phase. But the idea can be immediately captured in the form of a non-working 'Communicate' method on the Customer, such that it appears on the menu of any instance of Customer in the prototype. The team can then walk through scenarios, invoking this method (even though it doesn't yet work) when needed. This way the powerful idea that all communication with the customer is delegated to the Customer object itself is much less likely to be overlooked and there is therefore less risk of duplicating that functionality unnecessarily on other objects.

The other argument against capturing the object definitions directly into program code is that this implies making an early commitment to a specific programming language.

UML is independent of programming languages, but historically it has not been able capture functional definitions. However, UML is now evolving towards a vision called Model Driven Architecture [80] - in which functionality can be completely specified in a platform-independent notation such that a platform-specific implementation could be automatically generated. UML 1.4 defined the semantics for an Action Language Semantics (ASL) but not the syntax - with the result that proprietary implementations of ASL are unlikely to be interchangeable. UML 2.0 is

expected to rectify this. This concept is sometimes referred to as 'executable UML' [95]. But if UML ever becomes rich enough to fully specify the behaviour of a business system then it will, in effect, have become another programming language. Historically, attempts to create the programming-language-to-end-all-programming-languages have never succeeded in the long run..

Furthermore, much of the current desire to develop systems in a platform-independent manner reflects the dichotomy formed by the two leading platforms for business systems design: the Sun J2EE platform and the Microsoft .Net platform. The former depends on the Java language and the latter on C# or VB.Net. In fact, these languages are so similar in structure that utilities are already beginning to emerge for auto-translating between them.

Moreover, it has already been demonstrated using the *Naked Objects* framework that if the business objects are written in Java 1.1 (as distinct from the more recent versions of Java such as 1.4) then they can be compiled either onto the Java platform or the .Net platform because Java 1.1 is compatible with Microsoft's J#.Net compiler. Given that most of the extensions provided by the subsequent versions of Java are concerned with the user interface (e.g. the *Swing* framework), and the naked objects approach does not involve coding a user interface, there is really no disadvantage to restricting the business objects to using Java 1.1 only. In effect this gives all the advantages of the MDA concept, but using a programming language that is well established, expressive enough to span the programming domain from high level business functionality to low level technical functionality, and, to all intents and purposes, an open source language[11].

In his keynote speech at the OMG Conference on Model Driven Architecture in London in September 2002, Oliver Sims cited the naked objects concept as a potentially important future direction for MDA [108]. Haywood has also argued that tools such as Togethersoft's Control Center, in which the behaviour is captured in a real programming language are more effective than the MDA concept [46].

5.6 Develop the production system one scenario at a time

If the decision is taken to proceed from exploration to delivery, the system should be recoded from scratch. The exploratory prototype can certainly be used to guide and inform the main development, but only the object definitions (including, where appropriate, the interfaces and method signatures) should be carried forward.

[11] There are now several open source implementations of the Java Virtual Machine compatible with Java 1.1 or above. See http://joeq.sourceforge.net/other_os_java.htm

Furthermore, it is recommended that in the delivery phase the system be developed incrementally - where each increment delivers something of value to a user, such as a use-case. How is this recommendation consistent with the earlier injunction against use-case driven approaches during the exploration phase (page 65)? To reiterate the point, where use-cases are defined before the objects and their natural responsibilities have been identified, the risk is that it will encourage the specification of behaviourally-thin business objects, supported by use-case controllers. But once exploration has been completed the principal business objects and their responsibilities will have been broadly defined and both programmers and user representatives should have grown comfortable with the idea that a system can consist solely of behaviourally-complete domain objects exposed directly to the user - because they will have developed and used a prototype. Introducing use-cases after this point avoids the risks discussed earlier, and indeed potentially provides an appropriate discipline.

Nor are use-cases the only possible unit of incremental user value. Extreme Programming [12], for example, advocates developing the system one 'story' at a time, and Feature-Driven Development [81] advocates developing the system one user-valued 'feature' at a time. (Both of these methodologies are discussed further in Chapter 8, including their potential compatibility with naked objects.) For the remainder of this chapter, the word 'scenario' will be used to mean an increment of value to a user and can be interpreted as a 'use-case' or a 'story' or a 'feature' if desired.

In the delivery phase, an incremental approach to delivery, one scenario at a time not only encourages the early delivery of value to the users, but also provides an appropriate degree of discipline to ensure that the system provides all the functionality needed to complete the most common tasks.

Some of the scenarios will already have been prototyped during exploration, and if they have been formally documented they can be used to guide development of the same scenarios in the delivery phase. However, this small set will almost certainly not constitute a full specification. Many more scenarios will need to be specified during the delivery phase. Writing these scenarios will be considerably easier having already conducted the exploration phase, because the scenarios can be documented directly in terms of operations on the naked objects. Many of the new scenarios can be acted out by the user on the prototype - at least in part - before being formally documented. Even where a scenario requires the addition of a new attribute or method, or a new sub-class of object, it will be relatively easy for the team to imagine it.

5.7 During the delivery phase capture each scenario as executable user acceptance tests

This concept of articulating requirements in the form of scenarios that are expressed directly in terms of operations on specific objects opens up another intriguing possibility: that of capturing those scenarios in the form of executable acceptance tests.

The idea of writing executable user-acceptance tests in advance of developing the actual system is advocated by Extreme Programming (XP). In XP, planning is done at the level of user-generated 'stories', two or three sentence statements of requirement, which are prioritized into releases. When a particular story is to be implemented, the short description is fleshed-out through direct discussion between developer and user. In theory, this discussion culminates in one or more acceptance tests for that story, written by the user, or jointly by the user and a developer. By writing them in executable form, the developers can run these tests frequently during the development of the story, to get an indication of progress, and can run them as regression tests after subsequent refactoring [40].

However, it is notoriously difficult to write executable acceptance tests for systems with graphical user interfaces (GUIs) [60]. There are many tools that can capture and replay the keyboard and mouse events of an actual usage, but this approach to testing has many problems [44]. Any change to the layout or style of the user interface will require these tests to be re-recorded, as, in many cases, will porting the application onto a machine other than the one where the test was recorded. Moreover, this record-and-playback approach to testing can only be applied after the system has been developed.

Some of these recording tools provide high-level GUI scripting languages that, in theory, would allow the test scripts to be written in advance of writing the system. However, this still leaves the problem that it is very difficult for the user to imagine a yet-to-be-implemented user interface in sufficient detail to be able to write a detailed test script.

A more promising approach lies in framework-based tools, in which the tests are written not in terms of specific user gestures but in terms of higher-level actions such as opening a file, which abstracts the tasks of browsing, selecting and opening a file from a dialog box [38]. Such tests could be written in advance of coding a particular story, and would be relatively robust to minor changes in the user interface. However, the fact that they do not test the execution of the user interface itself somewhat reduces their efficacy. This would not be a problem if the user-interface was auto-generated from the same high-level user-interaction constructs used to specify the tests. There has been some research into this possibility (see, for example [90] or, for a more general review [77]). Generally, however, there is a paucity of general-purpose tools to support the high-

level framework approach to acceptance testing. The XP pioneers recommend building your own such tools, specific to a particular project [27], but this carries an obvious overhead.

With naked objects, however, the idea of a framework-based approach to executable acceptance testing becomes much easier to realise, for two reasons. The first is that there is a 1:1 correspondence between the user actions and operations on the underlying business object. The second is that the presentation layer (or 'viewing mechanism') is completely generic: once that software has been thoroughly tested it does not need to be retested for each application any more than the basic operating system needs to be retested. Taken together it is possible to envisage a testing framework that executes a sequence of actions on the business objects just as though they were coming from the user via the generic presentation layer.

Such a test harness has been written for the *Naked Objects* framework based on the author's ideas. (The DSFA is believed to be exploring the possibility of a similar concept for its *Naked Object Architecture*). Using this capability, the user and programmer sit down and verbally translate the new scenario into a script of user operations on the business objects. The programmer captures these, live, as a sequence of methods on specialized test classes. These test classes simulate the interaction between the *Naked Objects* viewing mechanism and the business objects. A (partial) example of such an executable acceptance test is shown below:

```
public void story2Reuse() {

    story("A booking where the previously used locations are re-used");

    step("Retrieve the customer object.");

    View customer = getClassView("Customers").findInstance("Pawson");

    step("Create a booking for this customer.");

    View booking = customer.rightClick("New Booking");

    booking.checkField("Customer", customer);

    step("Retrieve the customer's home and office as the...");

    booking.drop("Pick Up", customer.drag("Locations", "234 E 42nd Street, New
York"));

    booking.drop("Drop Off", customer.drag("Locations", "JFK Airport, BA
Terminal"));

    booking.checkField("City", "New York");

    step("Use the customer's mobile phone as the contact...");

    :
```

(Note: These test scripts simulate sequences of actions that a user might initiate to perform specific tasks. The test scripts do not form part of the application itself, and should not be confused with the idea of use-case controllers, discussed in Chapter 2.)

When the acceptance tests for a story are completed, the programmer(s) start designing and coding the necessary functionality. The acceptance tests run in a manner very similar to the popular open-source Junit tool for unit testing[12] - which is typically used for testing the correct execution of individual methods on objects. When all the acceptance tests for a given story run, the scenario is deemed to be implemented.

Auto-generated documentation

Such a testing framework could potentially also translate the test code into a plain English (or other user-language) equivalent i.e. a sequence of instructions to an actual user accessing the system through the viewing mechanism. This idea has now been implemented on the *Naked Objects* testing framework. Figure 5-5 shows an example of the auto-generated documentation, created in HTML. Objects are represented automatically as icons, and menu-commands are formatted to resemble the pop-up menus on the screen.

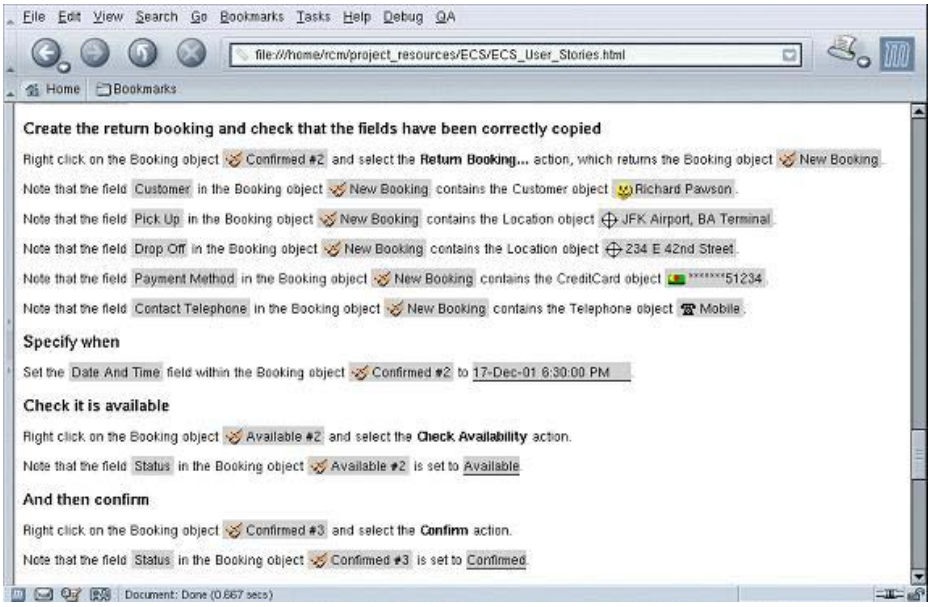


Figure 5-5 Example of HTML user documentation that has been auto-generated from an executable user acceptance test.

[12] www.junit.org

This capability means that is possible to auto-generate a significant proportion of the user training manual, consisting of a ready-formatted English (or other user language) script of user-operations for each test scenario - which constitute the principal business tasks.

Auto-generating large sections of the user manual eliminates a time-consuming task that few people enjoy. But a greater benefit is that this user training documentation is guaranteed to be consistent with the operation of the system. An alternative way to think about this process is that when fleshing out a particular scenario, the users are writing the training manual for that scenario, and an executable version of this training manual forms the acceptance test for the system.

CHAPTER 6 TESTING THE APPLICATION OF THE GUIDELINES AT SAFEWAY

Having developed the guiding principles defined in the previous chapter, an opportunity was sought to apply them on a new project starting from scratch. Such an opportunity arose at Safeway Stores in the UK, where two small naked objects project were undertaken during the course of 2001. Specifically:

- Both the projects were assessed against the characteristics defined on page 60 before commencement.
- In both cases the three pre-requisites specified on page 63 were met.
- Both projects started with a time-boxed exploratory phase involving a small team of developers and business representatives, and resulting in a working prototype that implemented an outline business model as naked objects.
- In both exploration phases the objects were identified directly instead of through use-cases, and were captured directly into working code.
- One of the projects went forward into a delivery phase, and was recoded from scratch on a incremental one-scenario-at-a-time basis.
- The concept of capturing scenarios as executable acceptance tests was introduced towards the end of the project, when the test classes became available as part of the *Naked Objects* framework, but these were then successfully deployed.

6.1 Background

Safeway Stores is the fourth largest supermarket chain in the UK, with over 480 stores ranging from hypermarkets to local convenience outlets.

IT management at Safeway became aware of the author's work on naked objects early in 2001. It was initially attracted to the concept not as an approach to software development, but as a way to train developers to think in a more object-oriented fashion. Although Safeway has to maintain a significant range of systems developed in Cobol, by 2001 Java had become its language of choice for new systems development, and the company had previously invested in training a number of its developers to use that language. However, the management recognized that while it had been reasonably successful in teaching its developers to use the syntax of Java, many had not adopted the object-oriented way of thinking, and were, in effect, writing Java programs in the Cobol style.

Using a very early version of the *Naked Objects* framework, Safeway arranged two 2-day training events on object-oriented thinking for Java programmers (the first one led by the author). Both events were successful, with very positive feedback from almost all of the attendees. In particular many of the developers stated that it gave them a much better understanding of object-orientation.

The success of the training exercises persuaded the head of the Java services team that it would be worth experimenting with naked objects as an approach to development.

6.2 Opportunity

The first opportunity arose in the area of pricing and promotions. Safeway had recently changed its marketing strategy. The new strategy is to compete through special promotions that offer up to 50% discounts on particular food and drink lines, designed to bring more customers into the store. Each week it prints and distributes some 11 million 4-page colour 'flyers' to households in the catchment areas for its stores. To prevent the competition from matching these offers, the set of promotions is constantly changed. Stores are grouped into clusters, and each cluster offers a different package of around 40 special promotions each week.

Implementing these promotions involves managing the supply chain to cope with big increases in demand for the discounted items, communicating the price changes to the point-of-sale systems in the stores, and printing and distributing the promotional flyers, in-store banners and shelf labels. Systems exist to manage each of these activities individually, but the overall planning and coordination of these activities is intensely manual, as is the planning process. Promotions managers are constantly exploring combinations of special offers with the intent of attracting the maximum number of shoppers who will then go on to buy regular items from the store, without merely encouraging 'cherry pickers' who take the best offers and nothing else. Each special offer must be coordinated with the supplier for logistics planning and, in some cases, to share the cost.

These managers would benefit from having a purpose-designed 'Deal Nominations' system to nominate new deals, forecast sales and availability, simulate their roll-out through the store clusters, and then coordinate their execution through the supply chain and price coordination systems. Using the criteria from page 60, this potential application was assessed for its potential fit with the naked objects pattern:

- **There would be benefit from characterizing the role of the user as a problem-solver rather than a process-follower**. 'Deal nominations' is essentially a problem-solving activity: any particular deal might start with a proposal from a supplier, or it might be initiated to complete a partly-assembled offering. A purpose-designed system should allow users to construct multiple offerings, simulate their effect, cut and paste them until they felt right - and then implement them.

- **Business agility is a primary concern.** This was not seen as a major issue for this system.
- **Requirements are uncertain.** Previous attempts by the systems department to analyse the requirements for such a system had not gone well. The activity did not fit well into the strongly process-oriented perspectives that are required for, say, supply chain management systems.
- **There is no real need for a hand-crafted user interface.** The system would be used only by internal staff.
- **The users are likely to be frequent users**. It was expected that those working on the promotions team would use such a system almost continuously.
- **Any batch processing is relatively simple in nature, or can be treated as a separate system.** All the batch processing would be taken care of by existing systems to which the new system would simply pass data.

This assessment suggested a very good fit and a decision was taken to proceed with an exploration phase, time-boxed at four weeks.

6.3 Exploration phase

A team was put together consisting of business domain experts, potential users, Java developers, database managers (because the prototype would need to access copies of existing databases) and an overall project manager. The project manager and two developers were full-time roles; others were very part-time. In addition to the core team, there were a number of managers closely monitoring the project to evaluate the effectiveness of the naked objects approach. The author acted as a consultant, providing the strong object modelling skills cited as one of the pre-requisites on page 63. The decision to use the *Naked Objects* framework fulfilled one of the other pre-requisites. The third one - a common understanding of intent - was achieved by giving all members of the team an opportunity to understand and explore the naked object application prototype that had been built previously for the DSFA.

On the first day of the exploration phase the team spent a couple of hours discussing the dynamics of the business in order to give the developers some familiarity with the domain. They then immediately started to identify the set of business objects that would best model the deal nominations area. No use-cases were recorded or discussed at this stage. Around twenty candidate objects were suggested, but by the end of the first day this list had been halved, and some high level responsibilities for the remaining candidates had been captured as plain text.

By the second morning the developers were already capturing the successful object candidate descriptions directly into Java code, using the *Naked Objects* framework, drawing icons suggested

by the business representatives, and assembling some realistic data for Products, Stores and so forth.

The next four weeks followed an iterative pattern. The whole team met once a week and reviewed the whole object model and the state of the prototype, deciding what the priorities would be for the next iteration. During the week there were many smaller iterations. A particularly effective way of working was to have an individual business representative sit down with a developer and evolve the prototype in real-time: adding new attributes or associations, new sub-classes, and simple new business methods. For more complex business functionality (especially where it involved searching collections of objects, or navigating long chains of command) the developers worked alone, or in pairs.

Throughout this period there was almost constant demand for demonstrations, both from members of the team, and from other parties that had heard about the radical approach of the project and wanted to know more. The project manager took on the role of chief demonstrator, recording and managing a set of demonstration scripts. (These demonstration scripts were, effectively, use-cases, but they were only created after the object model was beginning to stabilize). Apart from engaging the team, the demonstrations thus served the important task of continuously validating the object model.

Additionally, on various occasions during this exploratory period, the team was asked to identify 'agility scenarios'. These were not requirements, nor even likely future extensions. They were purely hypothetical scenarios, relating to future changes in the business organization, strategy and relationships, as well as technology-driven scenarios. Although these were not explored in detail, the team was asked to briefly explore what changes that new scenario might require in the model. Ideally, the answer would be that the changes would be limited to just one of the object classes, or perhaps to the creation of a new class that implemented an existing interface so that it could substitute for an existing object in any context.

The Deal Nominations project was not taken forward into development and implementation - it was undertaken solely in order to evaluate the potential of naked objects. However, the business representatives involved in that exploration have strongly expressed their desire to see such a system implemented.

6.4 The second project

Meanwhile, another group at Safeway had seen the Deal Nominations prototype and thought that the approach could help them with another difficult business problem. This project is known as 'cluster-based pricing' (CBP), but the details of this application are commercially sensitive. Naked objects were initially seen as a way to facilitate the modelling of the CBP requirements rather than

to implement the finished system. As the exploration progressed, however, it became clear that the users liked the concept very much. IT management also recognized that this system made an ideal candidate for a full-blown trial of the *Naked Objects* framework: the system offered high business value but had a small user base.

At that time, the *Naked Objects* framework lacked the enterprise services needed to implement real systems. Safeway therefore made available its best Java developer to explore possibilities. It soon became clear that the object/relational mapping required between Naked Objects and Safeway's existing mainframe databases could be achieved using Enterprise Java Beans (EJB) and XML. (Source code for this 'mid-tier' has since been developed, tested and contributed to the open source community.)

The exploration phase lasted four weeks, followed by three weeks of planning for the delivery phase, which included some architectural considerations. When the delivery phase commenced, only the object definitions were carried forward. All the Java code needed for the release was written from scratch, adopting a more rigorous approach to both coding and testing. Development was done on a scenario-by-scenario basis. The framework to support the writing of executable acceptance tests became available only towards the end of this project, but the developers were able to make some use of it and liked the approach.

The first release was ready for user testing after 90 days, which is remarkable given that this included developer training, Christmas breaks, and delays caused by changes and teething problems with the framework and the middleware. Initial performance was poor. However, this was because the EJB server was operating on a separate machine to the database. When the former was ported over to the mainframe, the whole system ran (in the words of the development manager) "as fast as anything we are used to running on the mainframe running under CICS".

A subsequent management decision meant that it was not possible to deploy the system on the required platform and it was consequently re-implemented using CICS/Cobol accessed via a dumb terminal. (This decision was not in any way a reflection on the success of the naked objects approach). However the developers of that implementation reported that the *Naked Objects* prototype provided a very effective specification for the delivery phase, and resulted in a better internal design for the finished implementation than would otherwise have been likely.

6.5 Evaluation

The intent behind the guidelines set out in Chapter 5 was to make it easier to realize the full set of benefits claimed for naked objects in Chapter 3. In evaluating the Safeway projects, emphasis was placed on evaluating those of the claimed benefits that had not already been clearly demonstrated by the DSFA case study.

To conduct this evaluation the author (who had acted as a part-time consultant and coach on both projects) interviewed 10 individuals who had been involved in one or both of these projects. The interview group was split approximately equally between developers and business user representatives. (A couple of the individuals combined both roles). This population of 10 represented almost all those who had had a significant involvement in these projects.

Despite the fact that the interviews were conducted 12 months after the last naked objects project had been completed, the responses convey a passion for the approach, from developers and business representatives alike. Several expressed the view that this was the way that the organization ought to be developing all its new business systems, and felt disappointed (even 'cheated') that the exploration projects had not gone on to implementation using the same approach.

Each individual was interviewed face-to-face and asked to agree/disagree with a series of statements, as applied to whichever of the project(s) that they had been involved in. Appendix VII describes the interviewing method, the questionnaire used, the complete results and further commentary. What follows is a selection of the findings that have the greatest significance in regard to the two claimed benefits listed above.

6.5.1 Naked objects facilitates communication between users and developers

One of the potential benefits that had been claimed for naked objects in Chapter 3 but not clearly demonstrated in the DSFA case study was that the naked objects would facilitate communication between users and developers. The Safeway participants were asked directly about this effect, and the responses shown in Figure 6-1 shows a strong endorsement for the claim.

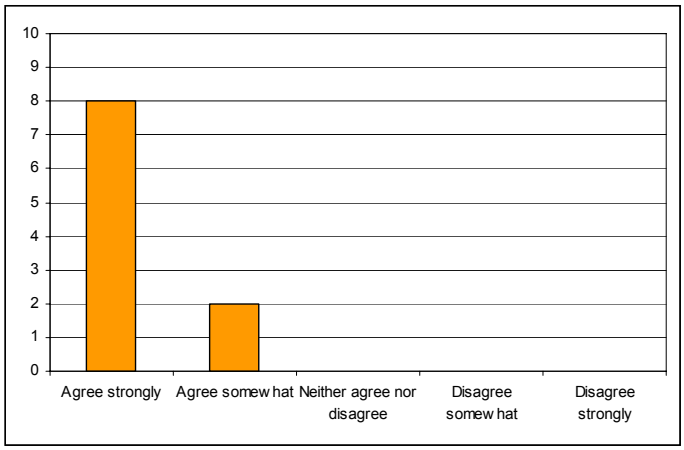


Figure 6-1 Response of ten participants to the proposition: 'Using Naked Objects greatly facilitated communication between developers and business representatives, during the discussion of requirements'

All ten of the participants further confirmed that they found it easy to get into thinking about the business system purely in terms of behaviourally-complete business objects (Figure 6-2).

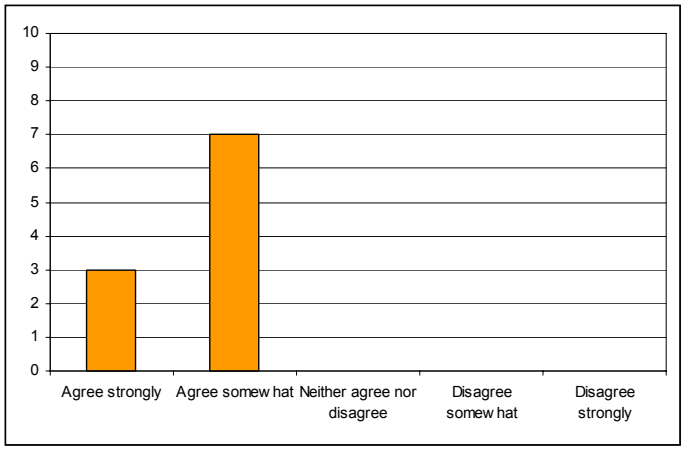


Figure 6-2 Response of ten participants to the proposition: 'I found it easy to get into thinking about the business system purely in terms of behaviourally-complete business objects

The participants who classified themselves as fulfilling a business role in the project were further asked whether they found any difficulty in adopting object-oriented concepts such as class, instance and method. Figure 6-3 shows that they did not find this particularly difficult.

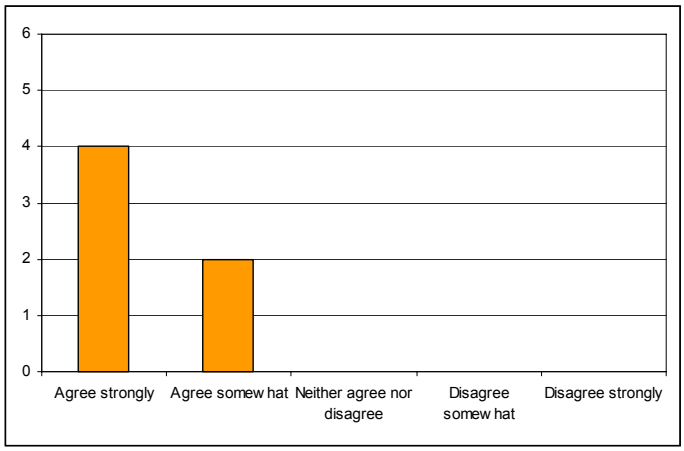


Figure 6-3 Response of six participants (business roles only) to the proposition: 'I did not find it difficult to adopt the object-oriented concepts (such as class, instance, method) used during the exploration'

Finally, although the participants did not necessarily agree that the applications being developed required a problem-solving user interface (see further questions in Appendix VII), Figure 6-4 shows that the majority felt that this problem-solving style of user interaction in the prototype did help during the exploration of requirements.

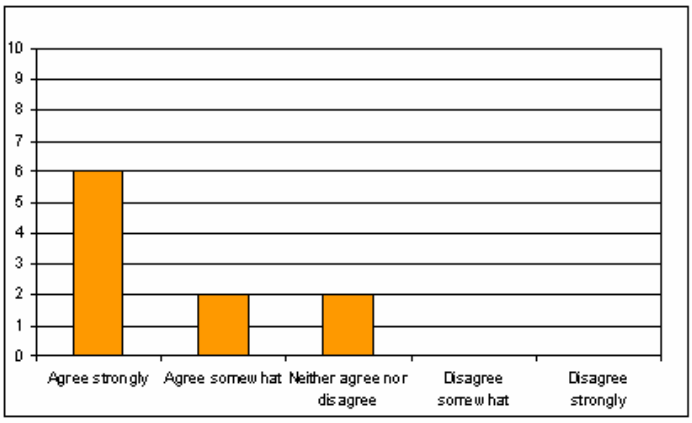


Figure 6-4 Response of ten participants to the proposition: 'The problem solving style of user interaction made a valuable contribution during the Exploration activity.'

6.5.2 Naked objects facilitate rapid prototyping

The final benefit claimed in Chapter 3 and not clearly demonstrated in the DSFA case study was that naked objects facilitate rapid prototyping. The responses shown in Figure 6-5 are unequivocal. Individual comments (recorded in Appendix VII) show that several of the participants felt that prototyping with naked objects was even faster than conventional screen-based prototyping, of which they had previous experience.

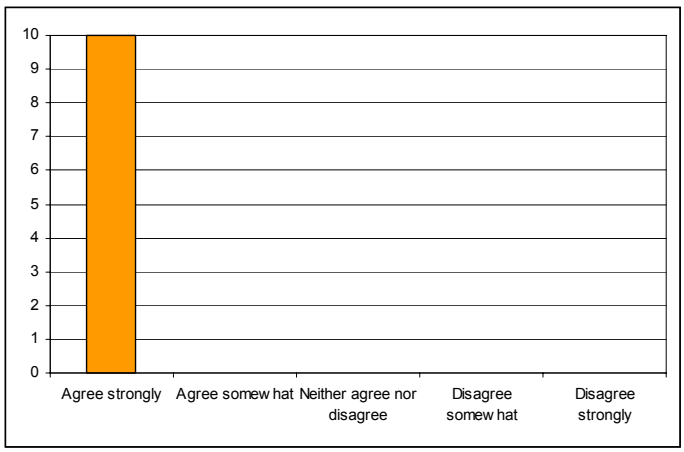


Figure 6-5 Response of ten participants to the proposition: 'Using Naked Objects we were able to prototype the underlying object model at least as rapidly as we could normally have prototyped screenshots alone.'

Six of the participants had direct experience of using the tools to prototype live in front of the business representatives. Figure 6-6 shows that all six of them found that to be a very effective way of working.

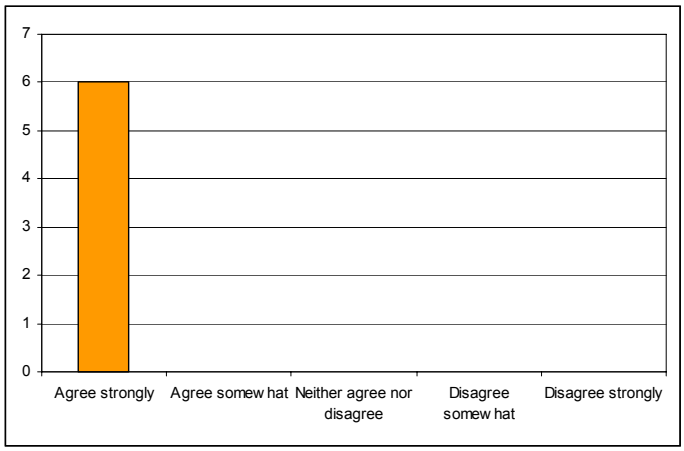


Figure 6-6 Response of six participants to the proposition:
'I found being able to prototype in front of users to be an effective way of working.'

6.5.3 Conducting a period of exploration prior to formal specification was valuable

A final theme worth drawing out from the survey is the effectiveness of conducting an exploration phase using the naked objects approach. Figure 6-7 shows that all ten of the participants were satisfied with the output of the exploration phase, eight of them strongly so.

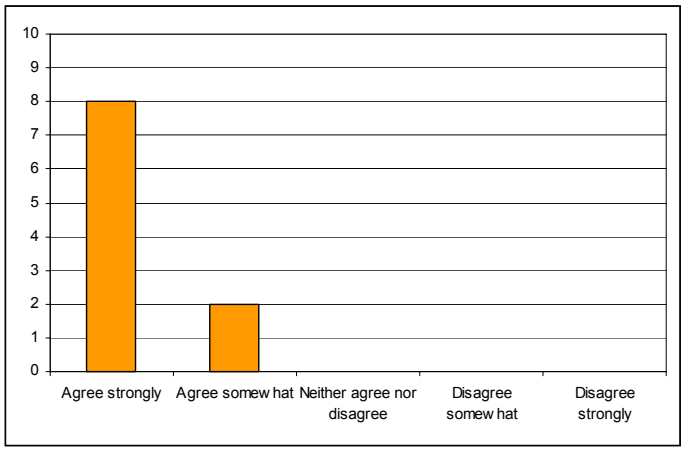


Figure 6-7 Response of ten participants to the proposition:
'Overall I was satisfied with the output of the exploration phase.'

Figure 6-8 further suggests that had the same project been conducted using a paper-based approach to requirements specification (or even conventional screen-based prototyping) most participants felt that some of the user requirements that were identified using the naked objects exploration approach might not have been identified.

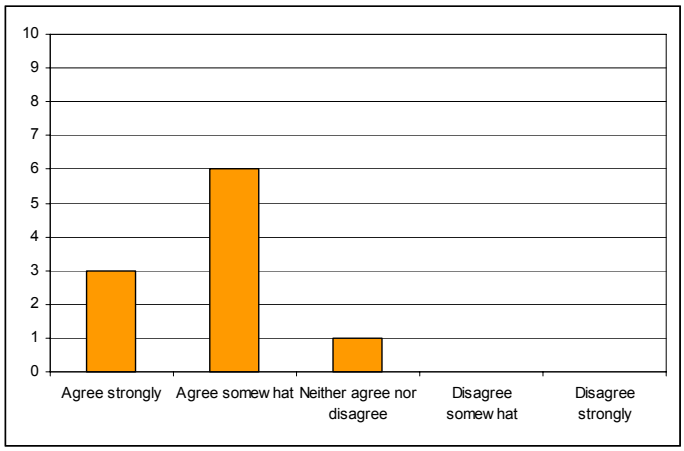


Figure 6-8 Response of ten participants to the proposition: 'The period of Exploration revealed specific user requirements that would probably not have been identified using a paper-based approach to requirements specification (or even conventional screen-based prototyping).'

In conclusion, the Safeway case study has demonstrated that when the guidelines suggested in Chapter 5 are observed:

- The two claimed benefits to the development process (rapid development cycle especially during prototyping, and improved communication between users and developers) can be readily achieved.
- In particular, the concept of an up-front period of exploration, when conducted in conjunction with the naked objects approach and a suitable framework, can be highly effective.

CHAPTER 7 CARSERV - A COMPARATIVE IMPLEMENTATION

Taken together, the testimony of business sponsors and users of real business applications at the DFSA (Chapter 4) and Safeway (Chapter 6) affirm all the benefits predicted for naked objects at a qualitative level.

Two of the benefits (a faster development cycle, and improved agility of the resulting system) are potentially quantifiable. The other two benefits (improved communication between developers and users during requirements analysis, and a more empowering style of user interface) would be difficult to quantify.

This chapter describes a controlled experiment to compare a system designed using naked objects with a system of equivalent functionality designed using a more conventional approache, in order to quantifiably test the hypotheses that the use of naked objects would speed up the development cycle and improve the agility (meaning the maintainability) of the resulting system.

The application chosen for the experiment is called *CarServ*, which is designed to support the operations of a typical automotive dealership including sales, servicing and administration. The conventional implementation of *CarServ* already existed before this experiment. It was written in 2001 by Dan Haywood as a teaching example, and was described in [20]. This conventional version is referred to in the rest of this chapter as *CarServ1*.[13]

7.1 Description of *CarServ1*

CarServ1 adopts a typical multi-layered architecture based on four layers:

- The presentation layer creates the graphical views of the objects, and captures user inputs. The presentation layer was written using Java *Swing* components. A typical screen from the system is shown in Figure 7-1

[13] The full source code for *CarServ1* is available at www.bettersoftwarefaster.org

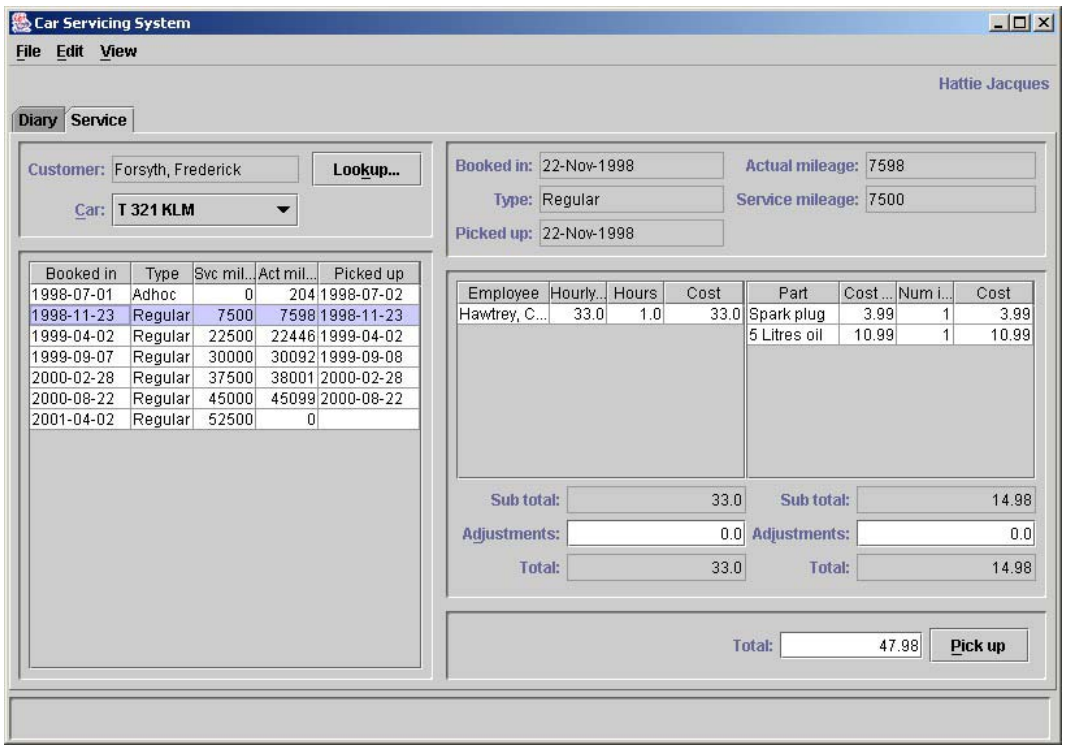


Figure 7-1 Screenshot from CarServ1

- The application layer. The objects in this layer encapsulate the functionality for user commands.

- The domain layer, where the business object entities are defined. The domain objects have some business behaviours, but they are mostly concerned with defining the attributes and relationships. A UML diagram of the domain model from *CarServ1* is shown in Figure 7-2. (The diagram uses the colour conventions proposed by Coad [22] to reflect different archetypes of object. The 'moment-interval' archetype, shown in pink on the diagram, corresponds closely to the concept of 'purposeful' objects described in Chapter 3).

- The data management layer. Each class from the domain layer has a corresponding data management object which is responsible for mapping the domain objects onto the persistence mechanism - a relational database.

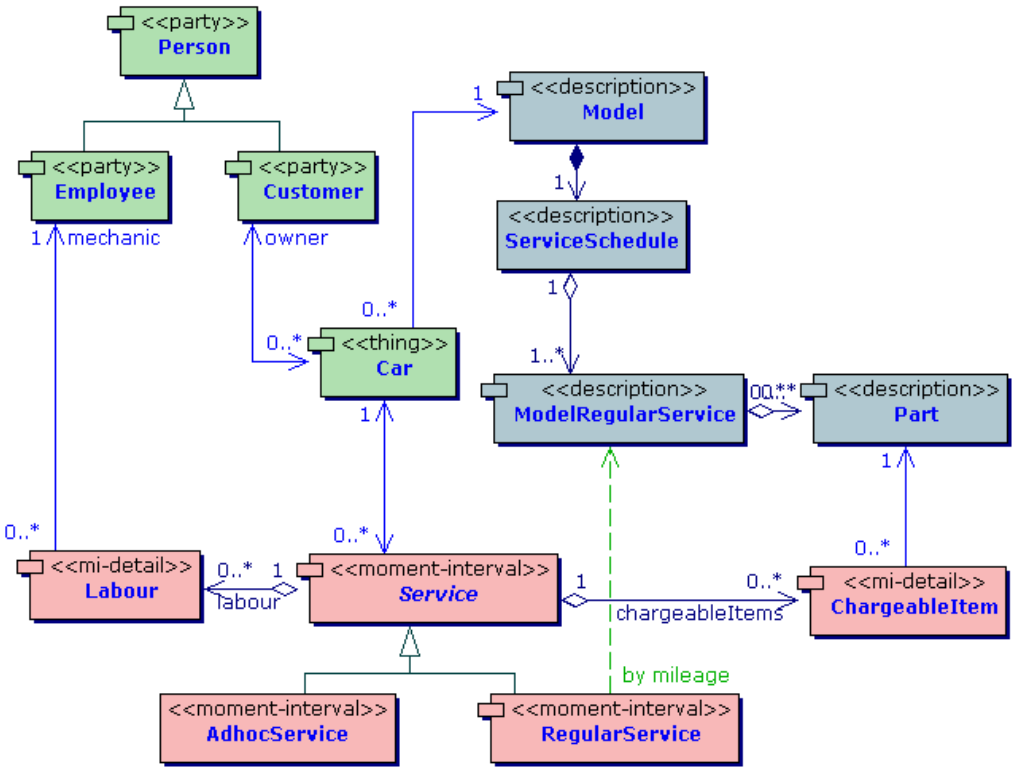


Figure 7-2 UML diagram of the business object domain model for *CarServ1*.

7.2 Defining a comparative implementation

In January 2003, Haywood collaborated with the author to re-implement the application using naked objects (by means of the *Naked Objects* framework). This naked objects version is referred to as *CarServ2*.

Comparative implementations of object-oriented designs can be very instructive - see for example [103] and [32]. But they are also somewhat problematic, being susceptible to several possible biases. It is not claimed that this experiment is proof against all possible biases, but it was designed to eliminate the most obvious ones:

- All the code for both implementations was written by the same person (Haywood), an experienced business systems developer, in the same language (Java), and using the same integrated development environment, the Togethersoft Control Center. They were also written just a few months apart. These factors help to allay suspicions that the results might reflect significantly different levels of programming experience, competence, or tool support.

- *CarServ1* was written and published prior to its author (Haywood) learning of naked objects. There is therefore no case that *CarServ1* might have been consciously or unconsciously written in a fashion that would accentuate the advantages of *CarServ2*.

However, this is not to suggest that *CarServ1* could not have been improved with further review and effort.

- *CarServ2* did have the benefit of two minds rather than one, with Haywood doing all the coding and the author observing and contributing. The author is not an experienced programmer in Java or any other language, so the advantage in coding terms is minimal. However, the author is experienced in business object modelling. For this reason it was decided that *CarServ2* would stick firmly to the domain model developed and publicly documented for *CarServ1*. It would be extended only where it was strictly necessary in order to accommodate the constraints of the new pattern or framework. During the development of *CarServ2*, several other potential improvements to the domain model were identified, but these changes were not made, because to have done so could have biased the comparison.

7.3 Description of *CarServ2*

The development of *CarServ2* consisted of taking the domain layer object definitions (i.e. attributes and relationships) from *CarServ1*, and re-coding them from scratch, observing the conventions required by the *Naked Objects* framework, and then adding behaviours (methods) to those domain objects sufficient to allow the use-cases that had been implemented in *CarServ1* to be fulfilled by a user. A screenshot from *CarServ2* is shown in Figure 7-3.

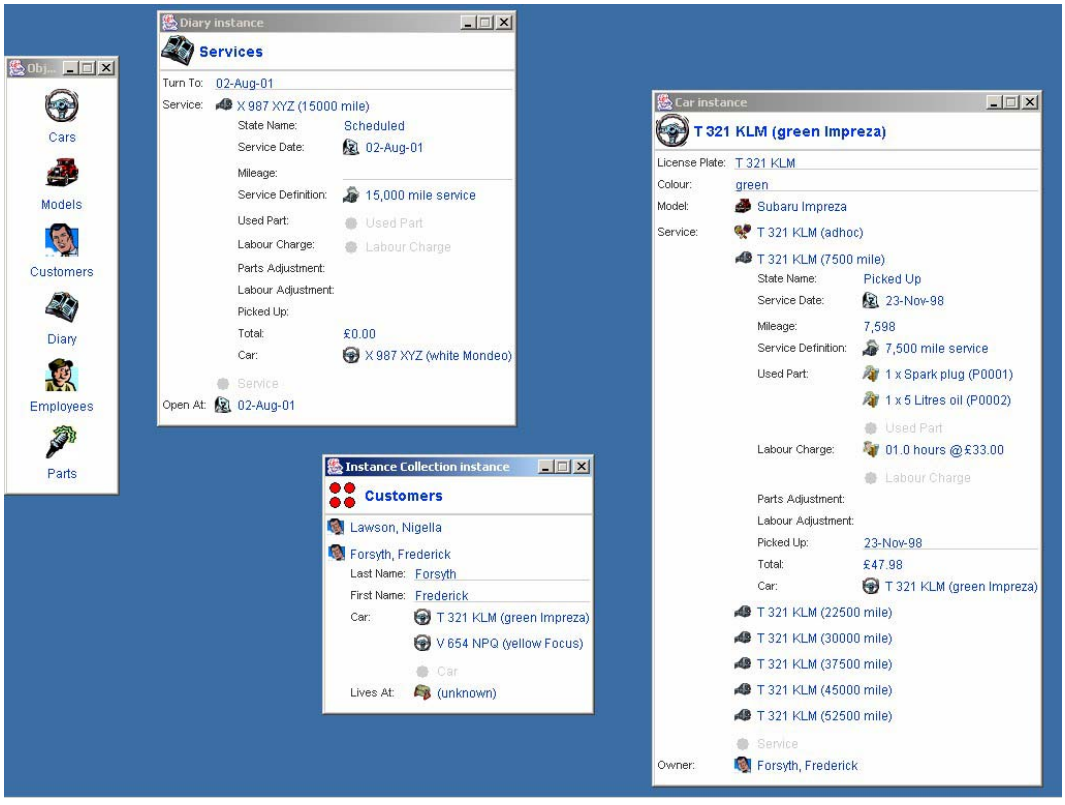


Figure 7-3 Screenshot from *CarServ2*

The objects were made persistent on the same relational database as *CarServ1*. In both cases the tables were defined by means of SQL statements, but since these are largely equivalent, they have been excluded from the metrics. In fact, the *Naked Objects* framework does have a capability to auto-generate the tables from the domain objects definitions, using Java reflection. However, this was not used in *CarServ2*.

7.4 Evaluating the development effort for the two implementations

When *CarServ2* was completed, the number of classes, the number of methods, and the lines of code written by the application developer for both implementations were counted. The results are shown in Figure 7-4.

	Number of Classes	Number of Methods	Average Methods per Class	Lines of Java code (LOC)	Average LOC per Method
CarServ1	190	788	4.1	7304	9.3
CarServ2	27	230	8.5	1726	7.5

Figure 7-4 Comparison of the code for the two implementations

The results show an improvement of (approximately) 7:1 in the number of classes, 3:1 in the number of methods, and 4:1 in the lines of code. The classes in *CarServ2* have more methods on average - reflecting the greater behavioural richness of the domain objects. Yet the average method length (lines of code divided by number of methods) is slightly lower for *CarServ2* than for *CarServ1*. (If the average method length had grown this would have suggested a loss in agility rather than an improvement).

Terser code is not necessarily faster to produce. *CarServ2* involved mastering the capabilities of the *Naked Objects* framework. A more complex framework not only takes time to learn, but requires more mental effort to use, which could offset the advantages of terse code (at least in terms of simple productivity).

One way to assess this complexity is to count the number of framework classes, and the number of unique methods on those classes, that the application developer has invoked explicitly from within the application code. Although the *Naked Objects* framework introduces several new classes and methods, it also eliminates the need for the programmer to deal directly with class libraries for user interface (in particular the *Swing* library). The initial results from this assessment are shown in Figure 7-5. (Note: references to classes within the java.lang and java.util packages, which are

constructs that any Java programmer needs to be familiar with to write any application code at all and are therefore common to both implementations.)

	External classes invoked within application code, ignoring java.lang and java.util	Unique methods on external classes invoked, ignoring java.lang and java.util
CarServ1	142	411
CarServ2	18	56

Figure 7-5 Assessment of framework complexity

The results show a substantial advantage for the *Naked Objects* framework. This suggests that the reduction in total code demonstrated in Figure 7-4 was not achieved at the expense of having to master a more complex framework. Indeed, using the *Naked Objects* framework significantly reduces the number of external classes and methods that the application programmer must deal with. Anecdotally, many developers report that the main difficulty of learning and using Java is not the basic constructs of the language, but the complexity of the class libraries such as Swing.

7.5 Some caveats

Some caveats need to be added to these results.

First, because naked objects auto-generates the user presentation, it was not possible to make the user interfaces the same. In *CarServ1* the programmer had full control over the presentation layer and therefore was able to optimize the screen layout and/or graphical design to the task in hand. In certain visual respects *CarServ2* is a poorer presentation - in particular in the presentation of the diary[14]. On the other hand, *CarServ2* makes more use of drag-and-drop and arguably has a better 'feel' than *CarServ1*. However, this experiment was not concerned with the relative usability of the two systems, but the development effort. The point is that because the user interfaces are quite different, the two systems cannot be said to be functionally identical.

Second, *CarServ2* does not implement 100% of the explicit functionality of *CarServ1*. For example *CarServ1* includes an Undo capability such that each business action can be undone (although the code to perform the undo must still be written for each method). In *CarServ2*, many

[14] A purpose-designed calendar viewer has since been incorporated in the *Naked Objects* framework.

of the simpler actions can easily be undone with a single operation, but some of them would require more than one user operation to reverse the effect.

However, *CarServ1* did not itself implement all the use-cases that had originally been specified for it in [20]. The application addressed only the needs of the service side of the business, not the sales and administrative aspects. Nor did it implement all of the use-cases that had been identified and documented for the service side.

In *CarServ2* functionality was written explicitly to support the same sub-set of use-cases as *CarServ1*[15]. However, the nature of the *Naked Objects* framework meant that *CarServ2* fulfils some of the use-cases not addressed in *CarServ1*, simply as a by-product of the development style. For example, for any business object class *Naked Objects* automatically provides the capability for the user to create a new instance of that class and to retrieve instances of the class matching a given character string. In its published state *CarServ1* could deal only with customers (and other objects) that already existed in the database; it did not implement the use case for adding a new customer. The net effect is that *CarServ2* implements many more use-cases than *CarServ1*.

Finally, some of the advantage in the *Naked Objects* implementation can be attributed to the simpler relationship between the domain objects and the underlying persistence. This is not strictly an aspect of naked objects, which is concerned only with the relationship between the domain layer and the user interface. Ideally, this effect would be factored out of the results, but as some business functionality exists in the Database Management layer in *CarServ1* it would not be realistic to take those objects out of the metrics.

7.6 Testing for agility

The next aim was to test the agility of the two implementations, meaning the ease with which they could be modified to accommodate unforeseen business requirements.

A set of business change scenarios was proposed including:

- Adding an email address attribute to the Customer object.
- Adding a capability to view the total value of previous transactions with the customer.
- Introducing a new kind of fixed-price service offering such as a Winter Tune-up.
- Introducing a pool of cars that customers can borrow during a service (subject to availability).

[15] Note that the use-cases are being used here only in the delivery phase i.e. the domain object model has already been specified. This is consistent with the guideline in Chapter 5.

The two implementations were then modified to accommodate each of these scenarios, and the effort required (measured in minutes) to make the changes to both systems was recorded. On each system the business change scenario was implemented in the form best suited to the design of that system. All the coding was again done by Haywood, who was author of both systems and therefore equally familiar with both.

More detailed descriptions of the change scenarios and the number of existing objects that they impacted can be found in Appendix IV.

The summary of these results is that the changes took less time to implement for *CarServ2* than for *CarServ1*. Averaged across all the changes, the difference between the two was approximately 25%.

More striking, though, was that the modifications to *CarServ1* were much more scattered: each modification to a business object typically involved modifications to objects in all four layers of the architecture. This suggests that if the application were much larger, and/or the developer making the modifications was not the application's author, the difference in modification effort between the two systems might be much larger. Testing this hypothesis would require a much larger effort than it was possible to devote to this experiment.

7.7 Conclusions from this case study

In Chapter 3, it was proposed that use of naked objects would yield four benefits (page 34). The controlled experiment on parallel implementations of the *CarServ* application provides quantified evidence to support two of those proposed benefits:

- **Proposed benefit 1: Behaviourally-complete objects, leading to greater business agility.** In *CarServ2* the domain objects are undoubtedly behaviourally-complete, because business functionality is not written anywhere else. The evidence for agility is that all of the business change scenarios were implemented in less time and involved modifying fewer objects than for the conventional 4-layer implementation.

- **Proposed benefit 2: Faster development cycle through not having to code a user interface.** All the metrics described in this chapter point to far more efficient coding, and it has been shown that this is not achieved at the expense of having to learn a more complex external framework.

The other two benefits were not appropriate to this experiment, because it was not concerned with capturing the requirements of the system (which were pre-defined in this case) nor was there a real user to evaluate the resulting user interfaces. Evidence for those benefits was provided in earlier chapters.

CHAPTER 8 RELATED WORK

In this chapter, the naked objects approach is compared to four other fields of research that have some conceptual overlap:

- Object-oriented user interfaces
- Existing techniques for exposing domain objects to the user
- Empowering user interfaces
- Agile methodologies.

8.1 Object-oriented user interfaces

Any system built from naked objects clearly has an object-oriented user interface (OOUI). In what sense, then, does the naked objects approach differ from, or extend prior work on OOUIs?

Answering this question is rendered more difficult by the widespread confusion over what constitutes an OOUI. Constantine [26] notes that many things labelled as an OOUI are simply examples of direct manipulation interfaces [105], of visual metaphors, or even just graphical user interfaces. In part this confusion may be explained by the fact that many of these ideas were developed at around the same time, as part of the Smalltalk project at Xerox Parc in the early 1970s [62] - although that in turn drew heavily upon earlier work by Sutherland [113] and Englebart [37]. Separating out these various concepts is therefore awkward.

Nielsen seeks to define OOUIs in contrast to function-oriented interfaces:

> *'Object-oriented interfaces are sometimes described as turning the application inside-out as compared to function-oriented interfaces. The main focus of the interaction changes to become the users' data and other information objects that are typically represented graphically on the screen as icons or in windows.'* [79]

Collins defines an OOUI as demonstrating three characteristics:

> *'- Users perceive and act on objects*
>
> *- Users can classify objects based on how they behave*
>
> *- In the context of what users are trying to do, all the user interface objects fit together into a coherent overall representation.'* [24]

A stronger way of stating Collins' second principle is that in an OOUI, from the user's perspective, all behaviours are associated directly and explicitly with an objects. The designers of the Xerox Star, one of the earliest examples of an OOUI, for example, adopted the principle that:

> *'applications and systems features were to be described in terms of the objects that users would manipulate with the software and the actions that the software provided for manipulating those objects.'* [59]

This approach has also been called the 'noun-verb' style of interaction [91] - because all user actions are initiated by selecting an object and then selecting a behaviour that is a property of that object. This is in contrast to the more widely-used 'verb-noun' style, where the user selects a task from a menu and then specifies the data on which that task is to operate. While this concept of noun-verb interaction marries well with that of graphical user interfaces, direct manipulation and visual metaphors, it is not dependent upon them. In the early versions of Smalltalk, user interaction was through a textual interface, but all user commands consisted of a reference to a specific object followed by the invocation of a behaviour provided by that object [63].

It can be argued that noun-verb interaction is the most important characteristic of an OOUI. IBM's Common User Access [52], which was probably the most thorough set of practical guidelines for designing OOUIs, showed strong evidence of commitment to this principle. CUA was conceived as an integral part of the OS/2 operating system. Some of the CUA guidelines were subsequently adopted in Microsoft Windows and other GUI frameworks. However, the OOUI principles have not been carried forward with anything like the purity with which they were originally stated.

CUA also demonstrated a very high commitment to behaviourally-complete objects, at least from a user perspective: demonstrations produced by the CUA team in the early 1990s[16] clearly show that all user actions took the form of invoking a behaviour that was a property of an object representing an obvious business entity.

But, surprisingly, this heavy commitment to behaviourally-complete objects at the user interface did not imply the same commitment underneath the user interface, as the following quotation makes clear:

> *'In an object-oriented user interface, the objects that a user works with do not necessarily correspond to the objects, or modules of code, that a programmer used to create the product. Inheritance and hierarchy in an object-oriented user interface are more subtle than in object-oriented programming. They are based on similarity in appearance and*

[16] In 2002 the author was shown one of the original demonstrations of CUA by Dave Roberts, who had been a team leader on CUA in the early 1990s.

behaviour, rather than on super-classes and sub-classes of objects . . . while object-oriented programming can facilitate the development of an object-oriented user interface, it is not a pre-requisite. An object-oriented user interface can be developed with more traditional programming languages and tools.' [52]

This statement is in marked contrast to the conclusion of Larry Tesler (the first member of the original Learning Research Group at Xerox Parc to move to Apple) as early as 1983:

'Many observers have hypothesized that [the] Smalltalk user interface and the Smalltalk language are separable innovations. Consequently, most systems influenced by the Smalltalk user interface have been engineered without resorting to Smalltalk's implementation approach. . . . At Apple, after using Pascal to implement six initial applications for Lisa, we discovered compelling reasons to change our programming language to incorporate more ideas from Smalltalk. Lisa applications are now written in the language Clascal, an extension of Pascal featuring objects, classes, subclasses, and procedure invocation by message-passing.' [115]

What fundamentally distinguishes the naked objects approach from existing work on OOUIs is that the behaviourally-complete objects at the user interface are a direct and automatic reflection of the underlying object model.

Although Van Harmelen appears to be hinting at the same idea where he states that:

'an object-oriented user interface is simply a user interface that articulates an object-oriented content model.' [118]

other statements in his book make it clear that he does not in fact see an OOUI as a direct reflection of the underlying object model:

'Object-oriented user interface design does not require designers to take an object-oriented view of the problem from the beginning of the project.'

'Furthermore, even if designers take an object-oriented perspective throughout, they will benefit from focusing separately on the object model and the object-oriented user interface design.'

8.2 Existing techniques for exposing domain objects to the user

There is some precedent for surfacing domain objects directly to the user. A recent posting on one of the newsgroups for Squeak [54] (a form of Smalltalk) posed the question:

'Is squeak really an object oriented system or it only claims it is? The point of the question is that instead of working with objects, I work mostly with text. the objects are in fact only in my head, as a consequence of reading sources of objects which are in the browser. but the objects are not tangible, I cannot see them. For example, let's take an instance of an OrderedCollection: this object is in fact not an object but a textual representation of it, I cannot see the collection on my workspace and must simulate all its behavior on my own and imagine it in my head.'[64]

One of the responses to this question is from Alan Kay (part of which was quoted in Chapter 2):

'One of the original motivations for the models, views and controller idea (that in my opinion never got well done) was to be able to automatically produce a default graphical interface for any object (and Steve Putz at PARC actually did a version of this but it didn't stick). These ideas live on in the "Naked Objects" book by Richard Pawson (worth reading by the way).

Randy Smith and the SELF folks did the next round of this at Sun (using the first version of Morphic by John Maloney and RS). There were many great ideas in this way of looking at UI.

Dan Ingalls' and Scott Wallace's FABRIK (in the 80s at Apple) was a direct manipulation of objects system in Smalltalk. Ned Konz' Connectors stuff today is very motivating along these lines, and could be make into such a system. Morphic wrappers by our friends in South America is another way to think about completely tangible objects in Squeak.

Your list of numbers and any other objects that you want to insert directly into source code was a feature of my thesis language of the 60s (FLEX), but it never quite got into Smalltalk, I'm not sure why -- it's the obvious way to deal with literals in code in a system that has many different kinds of things. Something closer to this is in the Etoys and we definitely plan to have the whole thing in the next version.'

The Morphic system [72] referred to by Kay was originally developed as part of the Self language [110] before being migrated into Squeak. Morphic is positioned by the Squeak authors as:

a direct-manipulation User Interface (UI) construction kit based on display trees. It is an alternative to Model View Controller (MVC). It will likely replace and obsolete (MVC)[17]

However, that claim by the Squeak authors is not supported by further argument or other evidence.

[17] See http://www.squeak.org/features/graphics.html

Certainly, there are some similarities with naked objects. Under Morphic, any object that inherits from the Morph class (provided with the framework) is automatically displayable. Moreover, when clicked with the mouse, the object will provide the user with a set of ready-made methods for user manipulation in the form of a 'halo' of coloured buttons (see Figure 8-1) around the object (blue rectangle).

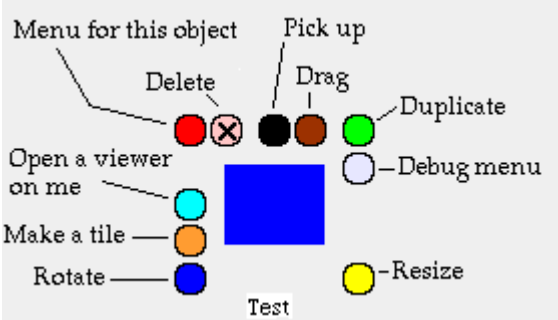


Figure 8-1 The standard 'halo' of manipulation methods made available to the user on any 'morphic' object.

Morphic is intended as an alternative way to build user interfaces. It is not positioned as an alternative way to design systems overall, and it has no stated intention of encouraging the development of behaviourally-complete objects. Additionally, the capabilities of Morphic are oriented strongly towards graphics and multi-media, rather than to the development of business systems.

Nevertheless, some of the thinking behind Morphic could prove to be relevant to the application of naked objects. For example, Maloney's work on 'use-mention' perspectives [52] is likely to become very relevant, if *Naked Objects* (or any other framework supporting the concept of naked objects) is extended to provide capabilities that will allow the user to customize the user interface directly.

The other examples cited by Kay (for an explanation of Fabrik see [55]) are of a substantially different nature. In these cases the emphasis is not on making objects more tangible to the end-user, but rather on making them more tangible (e.g. as icons) to the programmer of the system. The concept of adopting a more visual or concrete approach to programming is a substantial field of research (see for example [28] and [70]). But this is not the intent behind naked objects, and although there is a superficial overlap between the ideas, deeper probing reveals that they have in fact little in common.

One initiative that did attempt to move in the direction of forging a more direct link between the user interface and underlying object model was IBM's New World Interface (NWI or, sometimes, NEWI) project managed by Oliver Sims [107] [36]. This became the basis for a joint venture

between IBM and another software vendor but unfortunately the venture was disbanded shortly after it was formed. NEWI no longer exists and documentation is limited.

There are certainly some superficial similarities between NEWI and naked objects. However, a closer examination reveals that NEWI was not in fact committed to the idea of a 1:1 correspondence between the user interface representation and the object model.

Sims' subsequent work has moved more in the direction of componentization of business systems [47] rather than objects *per se*. However, Sims has publicly endorsed the concept of naked objects on at least one occasion [108].

8.3 Empowering user interfaces

One of the claimed benefits of using naked objects is that the resulting systems will be more empowering from the perspective of the user - because they treat the user more like a problem-solver and less like a process-follower.

Constantine strongly critiqued naked objects in December 2002[18] [25]. He objected both to the idea of auto-generated user interfaces and to the advocacy of pure object-oriented user interfaces - both on the grounds of usability. The naked objects approach combines both. Yet he conceded that:

> *'The ultimate significance of Naked Objects may be in the lessons it offers for practicing [user interface design] professionals, lessons that highlight the need for empowering users as problem-solvers by giving them better tools that enable them to achieve diverse ends by diverse means.'*

This concession reflects the fact that the usability community has not been too successful in paying attention to the concept of empowerment. A casual survey of some of the most cited publications on usability and/or user-interface design (including [79], [106], [15], and [5]) reveals no reference to the concept of empowerment. Although the standard elements of usability - efficiency, effectiveness and user satisfaction - could accommodate empowerment under the third heading, it is seldom explicitly identified as an issue.

Curiously, the concept of empowerment is not even widely recognized by those writing about object-oriented user interfaces (e.g. there is no explicit reference to empowerment in [24] or even any discussion of how an OOUI might change the relationship between the user and the computer).

---

[18] It should be noted that some of Constantine's arguments were based on early publications and presentations of naked objects, and that he had not at the time seen any of the empirical data provided in this thesis.

However, this shortcoming has been observed by a number of researchers approaching systems from a sociological perspective. Brown, for example, argues that the business process reengineering movement of the early 1990s has left many organizations unable to see anything that cannot be described in the language of formal processes [16]. He (and many other authors) suggest that this line of thinking can be traced back to Frederick Taylor, who, with his 'Principles of Scientific Management' [114], started the modern quest for efficiency and optimization.

Garson even suggests a more sinister factor at work:

> *'I had assumed that employers automate in order to cut costs. And indeed cost cutting is often the result. But I discovered in the course of this research that neither the designers nor the users of the highly centralized technology I was seeing knew much about its costs and benefits, its bottom-line efficiency. The specific form that automation is taking seems to be based less on a rational desire for profit than on an irrational prejudice against people.'* [43]

But what is empowerment? Clement identifies two distinct forms: functional empowerment and democratic empowerment [21]. Functional empowerment 'is oriented to improving performance toward organizational goals that are assumed to be shared unproblematically by all participants' - such as when a customer service representative is given greater authority to resolve a customer's problem in the interests of customer retention and the firm's reputation. Democratic empowerment has to do with giving the individual a 'greater grasp and sense of their own powers'. It is done in the interests of the individual and is not oriented towards achieving any explicit external goal, though there may be an indirect benefit to the business in improved motivation and staff retention. Democratic empowerment is a more subtle and elusive notion. Most initiatives aimed at strengthening empowerment are, despite the rhetoric, purely functional. Clement claims that

> *'For empowerment to offer an authentic promise of enhancing work experiences and outcomes, it needs to combine the attention to job effectiveness aspects of the functional approach with the emancipatory aspirations of the democratic approach.'*

Most core business systems are dis-empowering - in both the functional and the democratic senses. The functional disempowerment can be observed whenever the scripting of the system does not fit something that the user wants to do. This can frequently be observed at the customer interface, when the customer service representative cannot deal effectively with a customer's problem because it does not fit any of the standard scripts provided by the system. (Or it may be that the problem would fit one of those scripts, but the order in which the customer wants to provide the information clashes with the order that the script demands). Nor is this phenomenon limited to the customer interface: it occurs in all forms of operations. Airlines, for example, have sophisticated tools for planning and running the schedule. But when significant disruption such as a storm or a

technical fault occurs, the systems provide very limited support for simulating and then executing live workarounds [8]. Brown suggests that the most valued workers are often those who have found ways to work around the constraints deliberately imposed by the systems [16].

What makes most core business systems disempowering in the democratic sense is the style of interaction. Laurel finds that:

> *'Operating a computer program is all too often a second-person experience: A person makes imperative statements or pleas to the systems and the system takes action, completely usurping the role of agency.'* [69]

McCullogh talks about 'participation' (That is in the sense of the user's participation in the task being undertaken - not in the sense of participating in the design of the system, as advocated by the discipline of 'participatory design' [100]).

> *'The degree of personal participation, more than the degree of independence from machine technology, influences perception of craft in work . . . Individual guidance of process participation in outcomes of work was of course the very condition upended by industrialization . . . How to operate technology is not enough; it might be better to ask how to be when using technology. If it were possible to summarize this psychology in a single word, that word would be 'participation.' . . . the control of process, engagement with material, and identification with work that we admire in the traditional craftsman are clearly qualities of participation.'* [73]

Hutchins, Hollan and Norman provide the link between these concepts and object-orientation:

> *'There are two major metaphors for human-computer interaction: a conversation metaphor and a model world metaphor. In a system built on the conversation metaphor, the interface is a language medium in which the user and the computer have a conversation about an assumed, but not explicitly represented world. In this case, the interface is an implied intermediary between the user and the world about which things are said. In a system built on the model world metaphor, the interface itself is a world where the user can act, and that changes in state in response to user actions. . . Appropriate use of the model world metaphor can create the sensation in the user of acting on the objects of the task domain themselves.'* [51].

Naked objects are clearly compatible with their description of the model world metaphor. Moreover, the frameworks that support naked objects make it significantly easier to implement systems according to that metaphor.

This thesis is primarily concerned with the domain of software engineering rather than with the domain of human-computer interaction (HCI or CHI). Nevertheless, it is clear that there is

considerable potential for further research into the specific area of user-empowerment, and that naked objects could facilitate empirical research in this area.

8.4 Agile methodologies

The naked objects approach is not a systems development methodology: it is a small set of architectural principles, plus some guidelines for applying them. However, some of the benefits claimed for naked objects approach (page 34) do overlap with some of the benefits claimed for the various so-called agile systems development methodologies. It is therefore worth examining the nature of the relationship between them.

Although the concept of agile systems development is not new - the Dynamic Systems Development Methodology (DSDM), for example, was developed during the mid 1990s [112] - it came into sharp focus with the creation of the Agile Manifesto [10] in 2001 by a group of recognized but independent experts of software development. The agile manifesto has four principles:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan.

The manifesto does not suggest that the four attributes on the right hand side of these principles are unimportant: they are all considered important and valuable. But the signatories to the manifesto considered the four attributes on the left hand side to be of greater importance.

Naked objects, assuming that they are supported by an appropriate framework, potentially offer a good match with these four principles:

- **Individuals and interactions** over processes and tools. The user interface based on naked objects is one that emphasizes the capabilities of the individual user rather than a standard process - although this is a concept that applies to the resulting system rather than the development process, which is the domain of the Agile Manifesto. It could also be argued that the directness of the style of programming when using naked objects is one that emphasizes and leverages the concept of the 'craft' of programming - something that is very much aligned with this principle of the Agile Manifesto.
- **Working software** over comprehensive documentation. Using naked objects, where the relationship between the code and the object model is homomorphic, the code itself is arguably the best way to document the object capabilities - particularly as new tools

increasingly make it possible to view the code as UML class diagrams. The capability to generate a significant proportion of the user training manual automatically from the executable acceptance tests (described on page 72) is also a significant contributor to this principle.

- **Customer collaboration** over contract negotiation. Naked objects provide a shared representation between user and developer that has proved to be highly effective as a language for discussing requirements and even for imagining new possibilities. This is reinforced by the capability for very rapid prototyping.

- **Responding to change** over following a plan. Naked objects are behaviourally-complete. This makes the model very agile, because a high proportion of changes are localized to a single object. The strong support for automated user acceptance testing also reinforces the notion of responding to change, because a large bed of automated tests that can also be used for regression testing gives the programmers more confidence to make changes without worrying about introducing new bugs unknowingly. [40]

Given this strong fit between naked objects and the precepts of the Agile Manifesto, the next step is to evaluate the compatibility and/or potential synergy with specific methodologies that claim to be derived from, or at least compatible with, that manifesto. Abrahamsson [2] identifies the following list of candidate methodologies or approaches:

- Adaptive Software Development [48]
- Agile Modelling (AM) [6]
- Crystal [23]
- Dynamic Systems Development Process (DSDM) [112]
- Extreme Programming (XP) [9]
- Feature Driven Development (FDD) [81]
- Open Source Software Development (OSS) [34]
- Pragmatic Programming (PP) [50]
- Rational Unified Process (RUP) [67] [19]

[19] RUP may seem like an unlikely candidate for inclusion in a list of agile methodologies, but as Abrahamsson points out, it is, in theory, capable of being reconfigured into a lightweight form.

- Scrum [101].

Abrahamsson indicates that these various methodologies are very different in both their breadth (i.e. the proportion of activities associated with the traditional software development lifecycle that they address) and their depth (which ranges from high-level project management principles to specific programming practices). This is summarized in Figure 8-2, reproduced from that report.

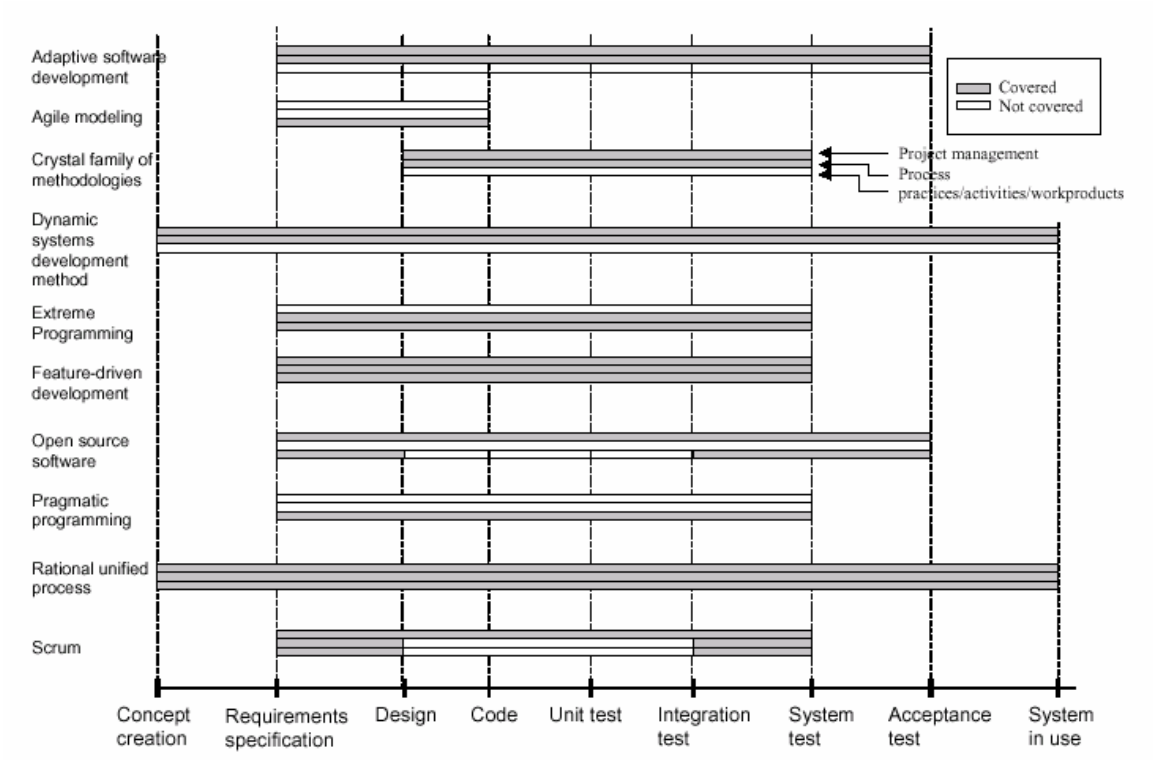


Figure 8-2 Scope of various 'agile' methodologies (reproduced from [2])

Based on further explanations provided by Abrahamsson, five of theses approaches were selected for further evaluation in terms of their potential fit with naked objects:

- Rational Unified Process (RUP)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Dynamic Systems Development Process (DSDM)
- Agile Modelling (AM).

8.4.1 Unified Process (UP)

The Rational Unified Process or RUP [67] is an extended and commercialized version of the Unified Software Development Process (USDP) [57]. For the purposes of this discussion, USDP and RUP can be treated as equivalent and will be referred to as the Unified Process or UP.

UP has two potential points of conflict with naked objects:

- The advocacy of Entity, Boundary, Control stereotypes
- The use-case driven approach to initial requirements analysis

These will be addressed briefly in turn.

As mentioned in Chapter 2, UP advocates the use of three object stereotypes - Entity, Boundary and Control - for both analysis and design of business systems. Naked objects were conceived as an alternative to this and similar patterns. Attempting to use naked objects within the UP would, by definition, require that the EBC pattern be dropped. That is not necessarily a problem because UP is officially a configurable process: the user of the methodology is encouraged to tailor it to the needs of a given project or the capabilities of a given organization, by selecting or removing specific elements. However, while some particular activities or artefacts (documents) can easily be deselected, little guidance is provided on how to remove something as fundamental as the EBC pattern, which is clearly an underlying assumption of the methodology's authors.

'Use-case driven' is one of the four defining principles of the UP (the others being 'architecture-centric, iterative, and incremental'). In Chapter 5 it was asserted that use-case driven approaches should not be used for the initial object identification and definition (i.e. during the exploration phase of a naked objects projects), but that use-case were acceptable and potentially valuable in the delivery phase. UP advocates their use for both purposes, with emphasis on the former. It is difficult to see how use-case driven analysis could be removed from the UP without destroying its essential character, and it is doubtful that the authors of the UP intended this to be one of the configuration options.

In summary, of the two conflicts identified between UP and naked objects, the EBC pattern could, in theory be dropped from UP, although this would be considered to be very unusual, but use-case driven approach to initial requirements analysis is definitional to UP. Therefore it must be concluded that UP and naked objects are not a good match.

8.4.2 Extreme Programming (XP)

The concept of naked objects resonates with a number of the defining principles of XP and the associated practices. Several of these have already been identified in earlier chapters:

- Naked objects provide a ready-made 'system metaphor' to guide the overall design (see page 37), which is an official requirement for an XP project.

- Naked objects support the principle of 'simple design' - by significantly reducing the number of object classes in a business system as well as the amount of code that needs to be written (see page 92).

- The concept of incremental delivery as defined in Chapter 5 (page 72) is highly compatible with the XP concept of incremental delivery, and specifically to the idea of developing a system one user story at a time.

- It has also been demonstrated that naked objects make it easier to write a framework that will support the XP-practice of writing executable user acceptance, even in advance of writing the system (see page 72). Indeed it can be argued that naked objects render this ideal achievable (there being little evidence that this practice is otherwise followed within the XP community [87][20]).

There would appear to be a possible conflict, however, between XP and the idea of an exploration phase as described in Chapter 5. The original definition of XP [9] does prescribe an 'exploration phase' but this is intended as a planning activity - it involves neither prototyping nor modelling. XP is often popularly described as being opposed to up-front modelling at all. This evaluation is refuted by several commentators, however, including [6] and [58], the latter introducing the idea of a 'spike solution' to test an approach to a particular problem. The notion of a naked objects exploration phase could be conceived as a 'spike solution' for the whole system, although this is possibly pushing the definition beyond its intent.

However, although the naked objects concept of an exploration phase is not provided for within XP, it is potentially synergistic with XP. Conducting a naked objects style exploration phase could make it easier to pursue a pure XP approach for the delivery phase: it provides a simple object model on which to build the system, and the concreteness of the model from the perspective of the user makes it easier to articulate specific stories.

8.4.3 Feature Driven Development (FDD)

The 'feature', from which FDD gets its name, is defined as a small, client-valued function expressed in the form <action><result><object> (e.g. "calculate the total of a sale") [81]. The methodology is built on five processes:

---

[20] This paper is reproduced in Appendix IX.

- Develop an overall model
- Build a features list
- Plan by feature
- Design by feature
- Build by feature.

The first three are conducted sequentially, and the last two as a repeating cycle. As an 'agile methodology' its provenance is somewhat curious. The original definitions of the five processes by De Luca in 1998 [30] predated the agile manifesto. According to those definitions one of the 'entry criteria' for the very first process (develop an overall model) is:

'The Requirements have been documented and are stable' [30]

which would certainly not be compatible with the ethos of the agile manifesto. In the latest definition of the processes [31], however, which is described as conforming to the agile manifesto, that particular entry criterion for the first process has disappeared, even though there do not appear to be any other changes that would compensate for such a radical change.

Nevertheless, the feature-by-feature development approach can be seen as very compatible with the incremental approach to the delivery phase argued in Chapter 5 (page 72). FDD has a more explicit commitment to object-oriented approaches than XP, both in the commitment to up-front modelling and in the syntax for defining features. FDD does make reference to using use-cases during the object modelling process but it is not prescriptive in how they should be used. It is therefore possible to use them only in the manner prescribed in Chapter 5.

Coad's object-modelling archetypes (referred to in Chapter 3) including the use of colour to distinguish stereotypes, have become closely associated with FDD [22], although the two are not mutually dependent.

Thus the overall pattern of FDD is strongly compatible with naked objects, and it could again be argued that the latter would make the former easier to adopt.

8.4.4 DSDM

DSDM is the oldest and the most formal of the so-called agile methodologies - and as such has gained a wider acceptance within commercial and governmental organizations than the others, though more so in Europe than in the US.

DSDM demonstrates a very high commitment to prototyping for the purposes of eliciting requirements, and to iterative development, suggesting a good fit naked objects. This compatibility is reinforced by the list of criteria [112] for when to adopt DSDM, which include:

- 'Is the functionality going to be reasonably visible at the user interface?'
- 'Are the requirements flexible and only specified at a high level?'

Taken together, these factors suggest that naked objects is potentially compatible with DSDM for those organizations that want agility but within the context of a reasonably formal methodology.

Safeway (Chapter 6) was already using DSDM on some projects when it started to explore the potential of naked objects. Some effort was devoted by methodologists at Safeway to codifying the way in which the two ideas could best be brought together. Although this might have been a way to gain wider acceptance for naked objects amongst those already committed to DSDM, it was not at all clear that the converse was true. The effort was subsequently abandoned. (However, it should be pointed out that this effort was conducted early in Safeway's experience with naked objects, during which both the naked objects approach and the *Naked Objects* framework were evolving rapidly. Had the work been done at a later stage, the value might have been clearer and the effort sustained.)

8.4.5 Agile Modelling (AM)

Agile Modelling is

> *'a collection of practices, guided by principles and values . . . it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeller. . . you need to use it with another fully-fledged process such as XP, DSDM, SCRUM or UP.' [6]*

Although not explicitly stated by its author, it could be argued that AM has two possible motivations:

- To increase the commitment to modelling in situations where there is already a commitment to agile development approaches
- To increase the commitment to agility in situations where there is already a commitment to modelling.

As such there is a strong resonance with the benefits of naked objects. For example, AM advocates the use of modelling for two purposes:

'To understand what it is you are building or to aid your communication efforts within your team or with your project stakeholders.' [6]

AM advocates modelling 'using the simplest tools', and although it accepts that in some contexts this could mean using CASE tools, there seems to be a strong preference for using whiteboards or Post-It notes. Yet it also argues that modellers should constantly seek to 'prove it with code', to 'obtain rapid feedback' and to 'consider testability as you model'. This combination of advice implies that there is an inherent tension between representing a model directly in code and the need for agility and communication.

Naked objects, supported by an appropriate framework, has the potential to eliminate (or, at least, to substantially reduce) that tension. Not only should all of the AM techniques be compatible with naked objects, but the latter should actually make the former more valuable and/or easier to realize.

8.4.6 Summary of compatibility between naked objects and agile methodologies

The comments on the various agile methodologies are based principally on an evaluation of the literature. The evaluation suggests that the naked objects approach is not only compatible with agile development approaches, but potentially makes some of them more effective and/or easier to realize. Indeed it is possible to envisage naked objects being the key to bringing together the benefits of four of the approaches discussed above:

- Agile modelling to understand the domain and facilitate communication
- Prototyping (in the manner advocated by DSDM) to cover specific user requirements
- Delivery one-feature, or one-story, at a time as advocated by XP and FDD.

Although these synergistic effects have yet to be evaluated rigorously, it is worth noting that, despite the public attention that they have gained, few of the agile methodologies and approaches mentioned above have themselves been subject to rigorous evaluation either.

CHAPTER 9 CONCLUSIONS

In this concluding chapter, the findings from the research are reviewed against the original objectives set out in Chapter 1. This is followed by a summary of the contribution that this research makes to the field of knowledge. The chapter ends with some suggestions for further research.

9.1 Review against the original objectives

Five objectives were stated in Chapter 1 (page 10):

- To identify factors that cause, or reinforce, the tendency to separate procedure and data in the design of systems, even where those systems are intended to use object-oriented approaches.
- To identify and specify an approach to the design of business systems that would help overcome those factors.
- To evaluate the use of this approach for the design of real business systems, and thereby to test its effectiveness in achieving the goal of behaviourally-complete objects.
- To test whether the use of this approach does ultimately lead to more agile systems, and whether there are any other advantages to be gained from it, as well as any disadvantages or limitations.
- To identify types of business system, or types of project, that would potentially benefit most from applying this approach.

The research is now reviewed against each of those five in turn.

Objective 1. To identify factors that cause, or reinforce, the tendency to separate procedure and data in the design of systems, even where those systems are intended to use object-oriented approaches.

The review of the evolution of object-oriented thinking presented in Chapter 2 suggests that the tendency to separate procedure and data in object-oriented designs can be attributed, at least in part, to the idea of separating the concerns of the user interface from the underlying object model.

This idea first materialized in the form of the Model-View-Controller (MVC) architectural pattern. The original thinking that led to MVC was motivated simply by the desire to support multiple visual representations of objects, and multiple client platforms - there was no intent that the View or Controller roles should embody any business behaviour. However, the Controller is commonly

implemented a 'use-case controller', which is a procedure by another name, and its adoption encourages the specification of behaviourally poor entity (Model) objects.

This line of thinking subsequently evolved into the 4-layer architecture - comprising Presentation, Application (or Controller), Domain Model, and Persistence layers - which has now become the dominant design for business systems architecture. Adoption of a 4-layer architecture encourages the separation of procedure and data.

Objective 2: To identify and specify an approach to the design of business systems that would help overcome the those factors.

Chapter 3 introduces a new approach built around the concept of naked objects. This approach enforces a correspondence between the structure of the user interface and the underlying domain object model, such that the domain objects appear to show through directly to the user, and all user actions consist of viewing those objects or directly invoking their encapsulated behaviours.

Enforcing this correspondence between user interface and domain object model implies the use of a framework purpose-designed to support naked objects. Two such frameworks have come into existence as a direct result of this research: the DSFA's proprietary *Naked Object Architecture* and the open-source *Naked Objects* framework.

When used with such a framework, naked objects encourage the design and implementation of behaviourally-complete domain objects in both negative and positive senses. In the negative sense, the only mechanism for making functionality available to the user is to encapsulate it on the appropriate entity object. In the positive sense, the visually-concrete nature of the naked objects makes it much easier for designers and users to envisage the role of objects and the behaviours required of them.

The approach outlined in Chapter 3 is extended in Chapter 5 in the form of seven broad guidelines for designing business systems from naked objects. These guidelines do not represent a complete methodology, but they could be used within a number of existing methodologies, especially the so-called lightweight or 'agile' methodologies. A review of their compatibility with such methodologies was conducted in Chapter 8.

Objective 3: To evaluate the use of this approach for the design of real business systems, and thereby to test its effectiveness in achieving the goal of behaviourally-complete objects.

During the course of this research, naked objects were used at two different organizations (DSFA and Safeway), using two different technical frameworks (The *Naked Object Architecture* and *Naked Objects*) for the design and development of real business systems. In both cases the author played an active role in the early stages of the project, and was able to observe the subsequent

phases closely. The projects, along with their evaluations, are presented in Chapter 4 and Chapter 6 respectively.

In both cases, the goal of achieving behaviourally-complete entity objects was realized in full: all the business functionality of the completed applications was encapsulated in the entity classes, which were exposed to the user, explicitly as objects.

An additional experiment (recorded in Chapter 7) compared two different implementations of the same application: *CarServ1* a conventional 4-layer implementation, and *CarServ2* using naked objects. The significance of this experiment in the context of Objective 3 is that the domain object model used for *CarServ1* (and written before its designer had heard of naked objects) was preserved for *CarServ2*. This meant that the business functionality, which in *CarServ1* was scattered across all four layers, had all to be encapsulated in the domain layer of *CarServ2*, without the addition of any entity classes. This demonstrates that the resulting objects were clearly entity objects - in other words that the need to encapsulate all the business behaviour did not imply some artificial changes to the definition of entity objects.

Objective 4: To test whether the use of this approach does ultimately lead to more agile systems, and whether there are any other advantages to be gained from it, as well as any disadvantages or limitations.

Having achieved behaviourally-complete objects, the evaluation of DSFA project and the *CarServ* experiment provide good evidence that this does in fact lead to greater agility. (The Safeway projects were not taken far enough to attempt to evaluate the agility of the finished system. However, as shown in the response to one of the survey questions - see Question 12 in Appendix VII - the managers involved believed that the resulting solution would prove to be agile if so tested in the future.)

Quantitative evidence for the agility resulting from behavioural-completeness comes from the *CarServ* experiment, when the two implementations were both subjected to several business change scenarios, carefully designed not to be biased towards either implementation. For all of these change scenarios (which are described in detail in Appendix VIII) the naked object implementation (*CarServ2)* proved the easier to modify both in terms of the number of lines of code involved and the time taken.

The evidence from the DSFA is qualitative, but it reflects real business needs for agility:

- Late in the development of the Child Benefit system, significant changes in functionality were specified, and, contrary to the expectations of the managers asking for them, these changes were accommodated without difficulty.

- When the new system was first rolled out, and subsequently, the DSFA was able to introduce significant organizational change and the new system has never prevented or restricted any required change.

- The new system has already been subjected (at a design level) to a radical change in requirements based on unforeseen changes coming from outside the organization (the GRO Online project).

- Having now completed the outline modelling for the Pensions Administration system, the DSFA has been pleasantly surprised at how the behaviourally-complete objects designed in the context of Child Benefit Administration have fulfilled many of the requirements of the pensions system without modification.

In addition to encouraging behavioural-completeness, and thereby achieving greater agility, three other benefits from using naked objects are hypothesized in Chapter 3. The evaluations of the three case studies also provide evidence for the realization of these three benefits:

- **A faster development cycle**. This is confirmed subjectively by the managers surveyed at Safeway (page 85) and objectively by the *CarServ* experiment (page 92). The DSFA did not experience this benefit on their first project because they had to develop the *Naked Object Architecture* in parallel with the first application.

- **Improved communication between developers and users.** This is strongly confirmed by the surveys of both developer and user-participants both at DSFA (page 54) and at Safeway (page 83).

- **A more empowering style of user interface**. This benefit is strongly confirmed by the survey of 15 users of the DSFA's Child Benefit Administration system (page 46). At Safeway this was not a motivation for using naked objects in either of the projects, although some managers did perceive it as a benefit by the end of the project (see Question 10 in Appendix VII).

Three potential limitations from the use of naked objects were identified during the course of the research (page 61). In brief they are :

- **The inability to hand-craft a user interface**. The experience of the case studies presented earlier suggest that this is less of a limitation than might be imagined - and that the advantages of a generic user interface outweigh the disadvantages. Nevertheless there are applications where a hand-crafted user interface is required either for marketing considerations, or because there is a need for a highly specialized visual representation (e.g. safety-critical applications).

- **The lack of explicit user-guidance**. Naked object systems emphasize empowerment over scripted guidance, and for many applications this can be beneficial. However, it is unlikely to be suitable for applications where the usage is only occasional.

- **Batch processing.** While naked objects do not cause problems for batch processing *per se*, the fact that they encourage a very pure approach to object-oriented design can bring some of the inherent difficulties of using objects for batch processing into sharper focus.

Objective 5: To identify types of business system, or types of project, that would potentially benefit most from applying this approach.

As discussed at the beginning of Chapter 5, naked objects are best deployed in systems where any of the following three statements is true:

- There would be benefit from characterizing the role of the user as a problem-solver rather than a process-follower

- Future business agility is a primary concern

- Requirements are uncertain

And all of the following three statements are true:

- There is no real need for a hand-crafted user interface

- The users are likely to be frequent users AND

- Any batch processing is relatively simple in nature, or can be treated as a separate system.

9.2 Contribution

The contribution of this research is the development of the 'naked objects' approach to designing business systems, and the demonstration that the adoption of this approach yields significant benefits both to the developed system and to the development process.

Using the naked objects approach to designing a business system, the domain objects (such as Customer, Product and Order) are exposed explicitly, and automatically, to the user, such that all user actions consist of viewing objects, and invoking behaviours that are encapsulated in those objects.

Four principal benefits have been demonstrated. The first two apply to the resulting product of the approach:

- The naked objects approach encourages the design of business systems from behaviourally-complete domain objects, in contrast to established approaches that encourage, either consciously or unconsciously, the separation of procedure and data. Thinking in terms of naked objects encourages behavioural-completeness in a negative sense - because there is nowhere to place business functionality except on the domain objects - but also positively in the sense that the concreteness of the naked objects makes it easier to identify their natural responsibilities. Others have argued that designing behaviourally-complete objects would lead to greater agility: that is the resulting systems should be more easily modifiable to accommodate unforeseen future changes to business requirements. As well as demonstrating that the naked objects approach leads to behaviourally-complete objects, this thesis has also demonstrated that the resulting systems are more agile.

- The resulting systems provide the user with a more empowering style of user interface. Strictly speaking this is a property not of naked objects *per se* but of object-oriented user interfaces (OOUIs). However, OOUIs are traditionally considered to be difficult to implement. Creating a framework designed to support naked objects implies providing a generic solution to the problem of creating OOUIs. Using such a framework, any system gets a pure OOUI for free, just by the action of defining the domain objects and their responsibilities.

The second pair of benefits relate to the development process:

- The development cycle is significantly shortened. The principal factor here is the elimination of the need (even of the possibility) to hand-craft a user interface - which traditionally accounts for a significant proportion of the effort involved in the development of an interactive business system. (This is not just due to the complexity of coding required, but also because traditional approaches to designing user interfaces encourage too much time to be focused on the detail of the presentation rather than the essential structure of the application). This advantage is shared with other techniques for auto-generating user interfaces, although they are typically not object-oriented. In addition to not writing a presentation layer, when using naked objects it is no longer necessary (or even possible) to write the 'controller' layer of a typical four-layer architecture. This also directly saves coding effort. However, a significant but more subtle factor in reducing the development cycle is that naked objects encourages simplicity of design.

- Naked objects provide a common language between application developers and users, which is invaluable during the early stages of requirements gathering and domain modelling. This is not to suggest that the roles of developer and user become blurred (as advocated by the concept of end-user development, for example). The roles remain

distinct, but both parties focus on the domain objects, their attributes and associations, their high-level responsibilities and/or their specific behaviours (methods). This, combined with the very rapid development cycle mentioned above, makes it possible to undertake the task of requirements gathering and domain modelling in an interactive prototyping style. It has been demonstrated that this form of prototyping is highly effective, and at least as fast as other approaches to prototyping, which are concerned purely with the user-presentation of the system. Using naked objects, it is the domain object model that is prototyped, with the user-interface an automatic by-product.

An additional benefit, introduced on page 74, is the ease with which it is possible to write executable user-acceptance tests for a business scenario, even in advance of developing the functionality itself. The concept of test-driven development of this kind is advocated within Extreme Programming and other agile methodologies, but is seldom practiced due to the technical difficulties arising from the use of graphical user interfaces. Naked objects may help this practice to be more widely adopted.

The genericity of the research has already been demonstrated through its application to three different business problems in three very different business domains: one hypothetical and two real case studies. In addition the concept has led directly to the creation of two independent frameworks to support the creation of systems from naked objects: the DSFA's proprietary *Naked Object Architecture* and the open source *Naked Objects* framework.

Based on the results of its first project using the *Naked Object Architecture*, the DSFA has already announced its intention to redevelop all of its core business systems using this approach. The open source *Naked Objects* framework has already been downloaded by more than 5000 people, and has built up an active community of users, as well as developers who have enhanced or extended the framework. The concept has already attracted significant attention within public discussion forums, and is widely linked on the worldwide web.

It is conceivable that with further work, naked objects could turn into a mainstream approach to business systems design.

9.3 Further research

During the course of the research several potentially interesting and valuable avenues for further research have been identified, but have not yet been pursued either due to constraints of time and resource, or because they require specialist expertise not possessed by the author.

Scalability

Probably the most pressing need is for further research in the area of scalability. The DSFA has implemented an 80-user naked object system, operating over a wide area network, but this is still not a large-scale system by some organizations' standards. As larger scales are attempted additional issues may surface, and it would be desirable to anticipate these, perhaps through some kind of simulation.

It is also clear that naked objects bring some of the known issues in object-oriented implementation into sharper focus. Chief amongst these is persistence. Most large users of information technology have significant investments in relational databases, and though there exist many patterns and tools to support the mapping of objects to a relational database, it is still not considered to be a trivial problem. Several patterns that arise in object modelling (including many-to-many associations, recursive or 'composite' object definitions, sub-classing and polymorphism) can only be satisfactorily handled by a relational database using association tables. These in turn can cause performance problems at very large scales. Naked objects do not make the problem inherently worse; but because they encourage a purer approach to object modelling, the aforementioned patterns are likely to appear more frequently.

Alternative approaches to persistence

One option may be to explore radically different models of persistence. One such is the 'prevalence mechanism' as implemented by the open-source Prevayler[21] tool in which all object instances are held in memory, and continuously serialized onto back-up storage. By encapsulating all updates using the Command pattern [42] it is possible to reconstruct the entire object image in memory in the event of a catastrophe. The concept of an object-oriented prevalence mechanism would appear to be strongly synergistic with naked objects. Given that both concepts are now supported by open source tools, exploring that synergy is an attainable goal.

Alternative viewing mechanisms

In Chapter 3 it is suggested that it would be possible to create viewing mechanisms for naked objects that were tailored to the capabilities of alternative client devices.

Others have started to pick up on this idea already. In June 2003, Franz Acherman released a first version of an HTML-only viewer for the *Naked Objects* framework. This uses the full capabilities of Dynamic HTML (DHTML) to emulate the pop-up menus, and even a form of drag-and-drop, within a native browser [3]. In effect the functionality of the generic viewing mechanism has been

[21] www.prevayler.org

moved from the client to a web-server. Similarly, a very crude version of a viewing mechanism for a Pocket-PC has been developed by Keranen [65].

It would be interesting to push the boundaries further. For example, it might be possible to conceive a purely auditory user interface (i.e. speech synthesis and recognition) for naked objects. This could still preserve the essential object-action (noun-verb) syntax, and arguably that might provide the possibility of more 'expressive' user interfaces for use by the visually impaired.

Further research has also been initiated into more flexible mechanisms for providing alternative visual representations for objects within the same generic viewing mechanism. Mugridge et al [76] describe early results from one such approach. In addition to providing multiple alternative visual representations, their system (built on top of the *Naked Objects* framework) implements the idea of a viewer as a first-class object. In other words, the user can drag objects onto an object that is a generic viewer. The viewer then provides some capabilities for the user to customize the view.

Another potential development is user-customization. Such a capability might allow the user to customize the layout of individual object views, or, perhaps, to repeat a frequently-used menu action as a button on the object viewer. This might make it possible to achieve all the benefits of naked objects, while eliminating one of its limitations.

BIBLIOGRAPHY

1. *What is a Controller, anyway?*, Portland Pattern Repository, http://www.c2.com/cgi/wiki?WhatsaControllerAnyway.

2. Abrahamsson, P., et al., *Agile Software Development Methods: Review and Analysis*. 2002, VTT Publications: Finland.

3. Achermann, F., *An HTML Viewer for Naked Objects*, http://www.nakedobjects.org/discuss/thread_frameset.php?group=nakedobjects.

4. ACM, *A.M. Turing Award*, Association of Computing Machinery. 2002. www.acm.org/announcements/turing_2001.html.

5. Adler, P. and T. Winograd, *Usability*. 1992: Oxford University Press.

6. Ambler, S., *Agile Modelling: Effective Practices for Extreme Programming and the Unified Process*. 2002: John Wiley & Sons.

7. Andersen, D., *Are Use Cases the death of good UI Design?*, UIdesign.net, http://www.uidesign.net/1999/imho/feb_imho.html.

8. Andersen, E., *American Airlines Object-Oriented Flight Despatch Systems (Case study)*. 1994, Harvard Business School.

9. Beck, K., *EXtreme Programming EXplained*. 1999: Addison-Wesley.

10. Beck, K., et al., *The Agile Manifesto*, The Agile Alliance, www.agilemanifesto.org.

11. Beck, K. and W. Cunningham. *A Laboratory for Teaching Object-Oriented Thinking*. in *OOPLSA '89*. 1989: Association of Computing Machinery.

12. Beck, K. and M. Fowler, *Planning Extreme Programming*. 2001: Addison-Wesley.

13. Ben-Nathan, R. and O. Sasson, *IBM San Francisco Developer's Guide*. 2000: McGraw Hill.

14. Berard, E.V., *Be Careful With "Use Cases"*, The Object Agency, http://www.toa.com/pub/use_cases.htm.

15. Beyer, H. and K. Holtzblatt, *Contextual Design*. 1998, San Francisco: Morgan Kaufmann.

16. Brown, J.S. and P. Duguid, *The Social Life of Information*. 2000, Boston, MA: Harvard Business School Press.

17. Brown, K., *Remembrance of Things Past: Layered Architectures for Smalltalk Applications.* The Smalltalk Report, 1995. **4**(9): p. 4-7.

18. Bruner, J., *Toward a Theory of Instruction*. 1966, Cambridge, MA: Belknap Press/Harvard University Press.

19. Buschmann, F., et al., *Pattern-Oriented Software Architecture*. 1996: Wiley.

20. Carmichael, A. and D. Haywood, *Better Software Faster*. 2002: Prentice Hall.

21. Clement, A., *Computing at work: Empowering Action by Low-Level Users*, in *Computerization and Controversy - Value Conflicts and Social Choices*, R. Kling, Editor. 1996, Academic Press: San Diego, CA.

22. Coad, P. and E. Lefebvre, *Modeling in Color*, in *Software Development*. March 1999.

23. Cockburn, A., *Writing Effective Use-Cases,* 2000: Addison-Wesley.

24. Collins, D., *Designing Object-oriented User interfaces*. 1995, Redwood City, CA: Benjamin/Cummings.

25. Constantine, L., *The Emperor Has No Clothes: Naked Objects Meet the Interface*, http://foruse.com/articles/nakedobjects.htm.

26. Constantine, L. and L. Lockwood, *Software for use*. 1999: Addison-Wesley.

27. Crispin, L., T. House, and C. Wade. *The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment*. in *XP2001*. 2001.

28. Cypher, A., ed. *Watch What I do*. 1993, MIT Press: Cambridge, MA.

29. Dahl, O.J. and K. Nygaard, *Simula -- an Algol-based simulation language.* CACM, 1966(9): p. 671-678.

30. De Luca, J., *The original FDD processes*, http://www.nebulon.com/articles/fdd/originalprocesses.html.

31. De Luca, J., *The latest FDD processes*, http://www.nebulon.com/articles/fdd/latestfdd.html.

32. Deligiannis, I., et al., *A Controlled Experiment Investigation of an Object-Oriented Design Heuristic for Maintainability*. May 2002, Bournemouth University, ESERG,.

33. Deutsch, L.P., *Design re-use and frameworks in the Smalltalk-80 system*, in *Software Reusability, Volume II: Applications and Experience*, T.J. Biggerstaff and A. Perliss, Editors. 1989, Addison-Wesley. p. 51-57.

34. DiBona, C., S. Ockman, and M. Stone, eds. *Open Sources*. 1999, O'Reilly.

35. Dubinko, M., et al., *XForms 1.0*. 2003, W3C. p. http://www.w3.org/TR/xforms/.

36. Eeles, P. and O. Sims, *Building Business Objects*. 1998, New York: John Wiley.

37. Englebart, D. and W. English. *A Research Center for Augmenting Human Intellect*. in *Fall Joint Computer Conference*. 1968. San Francisco: AFIPS Conference Proceedings.

38. Finsterwalder, M. *Automating Acceptance Tests for GUI Applications*

in an Extreme Programming Environment. in *XP2001*. 2001. Cagliari.

39. Firesmith, D., *Use Cases: The Pros and Cons*, in *Wisdom of the Gurus*, R. Wiener, Editor. 1996, SIGS books: New York.

40. Fowler, M., *Refactoring*. Object Technology Series. 2000: Addison-Wesley.

41. Fowler, M., *Patterns of Enterprise Application Architecture*. 2003: Addison-Wesley.

42. Gamma, E., et al., *Design Patterns - Elements of Reusable Object Oriented Software*. 1995, Reading, MA: Addison-Wesley.

43. Garson, B., *The Electronic Sweatshop - How Computers are Transforming the Office of the Future into the Factory of the Past*. 1988, New York: Simon and Schuster.

44. Groder, C., *Building Maintainable GUI Tests*, in *Software Test Automation*, M. Fewster and D. Graham, Editors. 1999, ACM Press / Addison-Wesley.

45. Hammer, M. and J. Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*. 1993: Harper Collins.

46. Haywood, D., *Evaluating the Model Driven Architecture*, Application Development Advisor, http://www.appdevadvisor.co.uk/Downloads/ADA7_1/Letters7_1.pdf.

47. Herzum, P. and O. Sims, *Business Component Factory*. 2000: Wiley.

48. Highsmith, J., *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. 2000, New York: Dorest House Publishing.

49. Holub, A., *Building User Interfaces for Object-Oriented Systems (Part 1)*, in *Java World*. 1999.

50. Hunt, A. and D. Thomas, *The Pragmatic Programmer*. 2000: Addison-Wesley.

51. Hutchins, E., J. Hollan, and D. Norman, *Direct Manipulation Interfaces*, in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. Norman and S. Draper, Editors. 1986, Lawrence Erlbaum: Hillsdale, NJ.

52. IBM, *Common User Access - Guide to User Interface Design*. 1991, IBM: Cary, North Carolina.

53. Ingalls, D. *The Smalltalk-76 Programming System Design and Implementation*. in *Fifth Annual ACM Symposium on Principles of Programming Languages*. 1978. Tuscon, AZ: ACM.

54. Ingalls, D., et al. *Back to the Future: The story of Squeak*. in *OOPSLA'97*. 1997: Association of Computing Machinery.

55. Ingalls, D., et al. *Fabrik: A Visual Programming Environment*. in *OOPSLA '88*. 1988: ACM.

56. Jacobson, I., et al., *Object-oriented Software Engineering: A Use Case Driven Approach*. 1992, Reading, MA: Addison-Wesley.

57. Jacobson, I., J. Rumbaugh, and G. Booch, *The Unified Software Development Process*. 1999: Addison-Wesley.

58. Jeffries, R., A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. 2001: Addison-Wesley.

59. Johnson, J. and *e. al*, *The Xerox Star: A Retrospective.* IEEE Computer, 1989(September 1989): p. 11-28.

60. Kaner, C., *Pitfalls and Strategies in Automated Testing.* IEEE Computer, 1997. **30**(4): p. 114-116.

61. Kanigel, R., *The One Best Way - Frederick Winslow Taylor and the Enigma of Efficiency*. 1997, London: Little, Brown and company.

62. Kay, A., *User Interface: A Personal View*, in *The Art of Human-Computer Interface Design*, B. Laurel, Editor. 1990, Addison-Wesley: Reading, MA. p. 191-207.

63. Kay, A., *The early history of SmallTalk*, in *History of Programming Languages*, T. Bergin and R. Gibson, Editors. 1996, Addison-Wesley / ACM Press: Reading, MA. p. 511-.

64. Kay, A., *Is Squeak really object oriented?*, Squeak Developer Newsgroup, http://lists.squeakfoundation.org/pipermail/squeak-dev/2003-May/058830.html.

65. Keranen, H., *PocketPC OVM v. 0.01 DEMO!*, http://www.iie.fi/vhe/noppcdemoapplet/.

66. Krasner, G. and S. Pope, *A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80.* Journal of Object Oriented Programming, 1988. **1**(3): p. 26-49.

67. Kruchten, P., *The Rational Unified Process: An Introduction*. Object Technology Series. 2000, Reading, MA: Addison-Wesley.

68. Lakoff, G., *Women, Fire and Dangerous Things*. 1987: Chicago Press.

69. Laurel, B., *Computers as Theatre*. 1991, Reading, MA: Addison-Wesley.

70. Lieberman, H., ed. *Your Wish is My Command - Programming by Examples*. 2001, Morgan Kaufmann: San Francisco, CA.

71. Madsen, O.L. *The Scandinavian School of Object-Orientation - in memory of Ole-Johan Dahl and Kristen Nygaard*. in *OOPSLA*. 2002. Seattle.

72. Maloney, J. and R. Smith. *Directness and Liveness in the Morphic User Interface Construction Environment*. in *UIST*. 1995. Pittsburgh: ACM.

73. McCullough, M., *Abstracting Craft: The Practiced Digital Hand*. 1996, Cambridge, MA: MIT Press.

74. Meyer, B., *Object-oriented Software Construction*. 1988: Prentice-Hall.

75. Moore, G.E., *Cramming more components onto integrated circuits*, in *Electronics*. 1965. p. 114-117.

76. Mugridge, R., M. Nataraj, and D. Singh. *Emerging User Interfaces through First Class Viewers*. in *CHINZ '03*. 2003. Dunedin.

77. Myers, B.A., *User Interface Software Tools.* ACM Transactions on Computer-Human Interaction, 1995. **2**(1): p. 64-103.

78. Nandhakumar, J. and D. Avison, *The Fiction of Methodolological Development: A Field Study of Information Systems Development.* Information Technology & People, 1999. **12**(2): p. 176-191.

79. Nielsen, J., *Usability Engineering*. 1993, San Francisco: Morgan Kaufmann / Academic Press.

80. OMG, *Model Driven Architecture - A Technical Perspective*, Object Management Group, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

81. Palmer, S. and M. Felsing, *A Practical Guide to Feature Driven Development*. 2002: Prentice Hall.

82. Pawson, R., *Naked Objects*, in *IEEE Software*. 2002. p. 81-83.

83. Pawson, R., J.-L. Bravard, and L. Cameron, *The Case for Expressive Systems.* Sloan Management Review, 1995(Winter 1995): p. 41-48.

84. Pawson, R., F. Hayden, and C. Dale, *The Expressive Object Architecture - A Framework for Designing Agile Business Systems.* CSC Foundation Research Journal, 2000. **1**(4).

85. Pawson, R. and R. Matthews, *Naked objects: a technique for designing more expressive systems.* SIGPLAN Notices, 2001. **36**(12): p. 61-67.

86. Pawson, R. and R. Matthews, *Naked Objects*. 2002: J Wiley.

87. Pawson, R. and V. Wade. *Agile Development with Naked Objects*. in *4th Int. Conf. on Extreme Programming and Agile Methodologies in Software Engineering (XP2003)*. 2003. Genova, Italy: Lecture Notes in Computer Science,

Springer.

88. Phanouriou, C., *UIML: A Device-Independent User Interface Markup Language*, in *Computer Science*. 2000, Virginia Tech: Blacksburg, Va.

89. Porter, M., *Competitive Advantage: Creating an Sustaining Superior Performance*. 1985, New York: Free Press.

90. Puerta, A.R., et al. *Model-Based Automated Generation of User Interfaces*. in *AAAI94*. 1994. Seattle.

91. Raskin, J., *The Humane Interface*. 2000, Reading, MA: Addison-Wesley / ACM Press.

92. Reenskaug, T., *Thing-Model-View-Editor*, Xerox Parc, http://heim.ifi.uio.no/~trygver/mvc/1979-05-MVC.pdf.

93. Reenskaug, T., *Working with Objects in the User Interfaces*, in *ObjectEXPERT*. 1996.

94. Reenskaug, T., *Model View Controller*, Portland Pattern Repository, http://c2.com/cgi/wiki?ModelViewController.

95. Riehle, D., et al. .*The Architecture of a UML Virtual Machine*. in *2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*. 2001: ACM Press.

96. Riel, A., *Object-Oriented Design Heuristics*. 1996: Addison-Wesley.

97. Rosson, M.B. and E. Gold. *Problem-Solution Mapping in Object-Oriented Design*. in *OOPSLA '89*. 1989. New York: ACM.

98. Rumbaugh, J., *Getting started. Using Use Cases to capture requirements.* Journal of Object Oriented Programming, 1994. **7**(5).

99. Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modelling Language Reference Guide*. 1999, Reading, MA: Addison Wesley.

100. Schuler, D. and A. Namioka, eds. *Participatory design: Principles and Practices*. 1993, Lawrence Erlbaum Associates: Hillsdale, NJ.

101. Schwaber, K. and M. Beedle, *Agile Software Development with Scrum*. 2001: Prentice Hall.

102. Shah, S., *Critique of use-cases*, Portland Pattern Repository, http://www.c2.com/cgi/wiki?CritiqueOfUseCases.

103. Sharble, R. and S. Cohen, *The object-oriented brewery: a comparison of two object-oriented development methods.* SIGSOFT Software Engineering Notes, 1993. **18**(2).

104. Sheth, A., W.v.d. Aalst, and I. Arpinar, *Process Driving the Networked Economy.* IEEE Concurrency, 1999. **7**(3): p. 18-31.

105. Shneiderman, B., *The Future of Interactive Systems, and the Emergence of Direct Manipulation.* Behaviour and Information Technology, 1982. **1**: p. 237-256.

106. Shneiderman, B., ed. *Designing the User Interface*. Third ed. 1998, Addison-Wesley: Reading, MA.

107. Sims, O., *Business Objects: Ease of Programming for Client-Server*. 1994: McGraw-Hill.

108. Sims, O., *MDA - The Real Value*, Object Management Group, http://www.omg.org/mda/mda_files/OMG-Information-Day-Sims_01-01.pdf.

109. Smith, H. and P. Fingar, *Business Process Management: The Third Wave*. 2003: Meghan-Kiffer.

110. Smith, R., J. Maloney, and D. Ungar. *The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility*. in *OOPSLA '95*. 1995: Association of Computing Machinery.

111. Stabell, C. and Ø. Fjeldstad, *Configuring Value for Competitive Advantage: On Chains, Shops and Networks.* Strategic Management Journal, 1998(19): p. 413-437.

112. Stapleton, J., *Dynamic Systems Development Method*. 1997, Reading, MA: Addison-Wesley.

113. Sutherland, I. *Sketchpad: A Man-Machine Graphical Communication System*. in *Spring Joint Computer Conference*. 1963.

114. Taylor, F., *The Principles of Scientific Management*. 1911, New York: W.W. Norton & Co.

115. Tesler, L. *Object Oriented User Interfaces and Object Oriented Languages*. in *ACM Conference n Personal and Small Computers*. 1983. New York: ACM.

116. Truex, D., R. Baskerville, and J. Travis, *Amethodical Systems Development: the Deferred Meaning of Systems Development Methods.* Accounting Management and Information Technologies, 2000(10): p. 53-79.

117. Utterback, J., *Mastering the Dynamics of Innovation*. 1994, Boston, MA: Harvard Business School Press.

118. Van Harmelen, M., ed. *Object Modelling and User Interface Design*. 2001, Addison-Wesley: Reading, MA.

119. Wirfs-Brock, R., *Characterizing Your Objects.* SmallTalk Report. **2**(5).

120. Wirfs-Brock, R. and B. Wilkerson. *Object-oriented Design: A Responsibility-Driven Approach*. in *OOPSLA*. 1989. New Orleans.

121. Wirfs-Brock, R., B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*. 1990, Englewood Cliffs, NJ: Prentice Hall.

APPENDIX I. DEFINING PRINCIPLES FOR THE DSFA'S NAKED OBJECT ARCHITECTURE

This appendix specifies the defining principles of the *Naked Object Architecture* as commissioned and deployed by the Department of Social and Family Affairs (DSFA) in Ireland. The text has been extracted from documents that have been publicly issued by the DSFA on several occasions, most recently within its June 2003 'Request For Tender to migrate existing Visual Basic COM+ code base to Microsoft .NET'.

Background

The Department is currently in the process of implementing the Naked Object Architecture (NOA) through which it expects to significantly increase its organisational agility in terms of its ability to cope with change both at macro and micro levels i.e. its ability to adopt new schemes in accordance with Government policy and its ability to change and adapt existing schemes and applications. Organisational agility includes strategic, technical and operational agility.

The Naked Object Architecture implements a Business Object Model of the Department. These high level objects currently encapsulate sufficient functionality to implement the Child Benefit scheme of the Department. It is proposed in this phase to extend the functionality of existing objects in the Business Object Model to include new objects specific to the schemes to be implemented. The Business Object Model is described in terms of both its current set of responsibilities and of additional responsibility requirements at: http://portal.welfare.ie/welfaretopics/dept_tenders/net_mig/index.xml

In the implementation of the Naked Object Architecture the core business objects are seen not just as defining the internal structure of the business systems, but as defining the user-interaction model. Users interact directly with those core business objects. The Department believes that this approach not only results in a more natural user interface, but is also critical to the achievement of the micro-level agility referred to above.

The concept of an Naked Object Architecture grew out of a research conducted by Richard Pawson, a Research Fellow at Computer Sciences Corporation, into the concept of Naked systems.

The Naked Object Architecture is implemented in a modern, multi-tiered, client/server architecture which achieves the clean separation of the front-end, business logic and data layers. It is implemented on a Windows 2000 platform using Java at the front end, COM+ components (written in Visual Basic) in the middle tier, and Microsoft SQL Server in the data tier. Communication between the front-end and the middle tier business objects is achieved using XML over HTTP.

Naked Object Architecture in DSCFA

The Naked Object Architecture defines a set of pre-requisites that must be in place to support it. The Department has implemented these in the following manner:

A componentised software infrastructure that supports distributed software components, with object-style interfaces based on published standards.

The Department has implemented this on the Microsoft COM+ standard. This infrastructure provides a set of services to support the creation and management of the objects including all of the following:

- **Persistence**. Object states are made persistent usually by means of a SQL Server database. Each object has its own set of archive rules. Some of the core data underlying the Customer object is held in the CRS database which is used extensively by the Department's OpenVMS based systems for managing short-term benefit Schemes. The CRS database is built using Oracle RDB. In the Naked Object Architecture, data access from Windows 2000 to CRS is achieved using Attunity Connect.
- **Identification and location.** Each object instance in the system has a unique identifier and the Naked Object Architecture provides directory services to locate them.
- **Message brokering** (including asynchronous message processing) is achieved using a set of proprietary components which implement queuing services to implement a publish and subscribe architecture. XML message formats are used.
- **Distributed transactions** are achieved across the Windows 2000 and OpenVMS platforms using COM+.
- **Access control.** The system can determine which objects a user (or system) can have access to, and which individual methods of those objects they can invoke.
- **Audit trail.** All objects are capable of recording an audit trail of changes made to the object. This audit trail can be read on-line. All future objects will be required to have this capability and to integrate with the online audit access mechanism.

A strong commitment to the concept of object orientation.

In this regard, it delivers the following capabilities to the Business Objects:

- **Instantiability**. Business Objects such as Customer, Officer and Scheme are defined as classes. The principle of instantiability states that an individual object can be created, using the class as a template, for each instance that the user is dealing with: for example, one

instance for each Customer. Each of those instances is separately identifiable and addressable - it has a unique identifier. References to a particular object are held within objects, and through this reference it is possible to access the full capabilities of the object being referenced.

- **Behavioural richness**. In the Naked Object Architecture, a Business Objects encapsulate all of the business behaviours that may be associated with it. Some of these behaviours invoke methods on other objects, and some behaviours will be delegated on to another object, but transparently. There is no separation of data objects and process objects.

- **Substitutability**. The NOA requires that it be easy to substitute one object type for another where this would make sense from a business point of view - for example, the ability to substitute an Agent for a Customer in a transaction. This is achieved by COM+ by means of an "interface inheritance" pattern.

- **Dynamic extensibility.** The infrastructure permits the definition of an object class - meaning both the list of attributes and the behaviours - to be altered or extended, without the loss of any existing instances of that class that are held in persistent storage. For example, if a new method called RegisterNewChild is added to the Customer object, then that method would be automatically acquired by existing instances.

Defining principles for the Naked Object Architecture

Building on these pre-requisite capabilities the Department has created a set of Naked Business Objects that model core business constructs such as Customer, Scheme or Payment. The Naked Business Objects have six defining characteristics:

- **Exposure**. Naked objects are exposed directly to the user -in a form that makes it obvious to the user that they are dealing with an object. This includes the use of an icon to represent the object. Most importantly, though, it includes the exposure of the object's potential behaviours to the user who typically selects an object and then invokes a behaviour upon it. This implements a 'noun-verb' style of user interaction, rather than the more common verb-noun style. In this way, the Naked Object Architecture presents the user with a set of tools with which to operate and allows a business system to be designed that does not dictate the users sequence of actions. This allows the user to be a problem solver rather than simply a process follower.

- **Direct Manipulation**. The NOA provides facilities for the direct manipulation of the Naked objects, such as point-and-click, drag-and-drop.

- **Class methods.** As well as exposing object instances, the NOA provides the user with an explicit representation of the classes of Naked objects. Using these class representations, the user can initiate a set of class methods, including for instance methods for creating a new instance of that class, for retrieving a particular instance of that class from storage using a unique identifier and for finding instances that match specified criteria.

- **Single point of definition.** Naked objects are ideally defined in a single place. The representation and role of the Naked object in multiple tiers of the architecture are all derived from that single definition - including the definition of persistent storage in a database.

- **Auto-generated user interface.** The default user interface is also ideally automatically created from the central definitions of the Naked objects. The Business Object defines for each of its methods the business-related information the Presentation Layer needs to display and to capture from the user. This information is subject to security filtering, to ensure that each user is only shown the information he/she is authorised to see.

- **Channel adaptation** The Departments Naked Object Architecture uses XML as the interface mechanism between the Presentation and Business layers. New channels would need to be able to interpret the XML and determine how it should be displayed.

APPENDIX II. RESPONSIBILITY DEFINITIONS FOR THE DSFA'S BUSINESS OBJECT MODEL

The following description of the business object model used by the Department of Social and Family Affairs (DSFA) in Ireland is extracted from a Request For Tender issued by the Department in July 2003. The business objects are defined primarily on the things that a business object is responsible for knowing (the 'know-what' responsibilities) and the things that a business object is responsible for doing (the 'know-how-to' responsibilities).

For clarity and space, only the definitions for the six core (or 'primary') business classes have been included: Customer, Scheme, Communication, Payment, Officer, and Case. The descriptions include updates that have been added since the Phase I (Child Benefit Administration) application.

Customer Object

A customer is anyone who has dealings with the State and has been assigned a Personal Public Service Number (PPSN). The intent of the Customer object is to provide a single point of access to any and all customer-related information that might be of value in more than one context (e.g. for more than one scheme). Where information is clearly specific to one scheme, such as the recording of a mouth-map for Dental Benefit then this information may be held in the scheme itself - but the guiding principle is to favour the Customer as the repository. The Customer is also the point through which any action that pertains directly to the customer is initiated e.g. authenticate and communicate.

The Customer Object has been implemented in Phase 1 to the extent required for Child Benefit claim processing. The extensions to the Customer object now required relate to the knowledge of social insurance contribution and earnings histories. These will both be implemented as secondary objects in their own right.

The bulk of the data underlying the Customer object, and of some of the new secondary objects such as Contribution History, is held on the Department's existing Central Records database (CRS) which is persisted using Oracle RDB on an OpenVMS platform. However, the initial definition of the Customer object in the implementation of the Child Benefit system required the storage of additional data items and used a separate database to store these. This data is held on an MS SQL Server database on a Windows 2000 platform and connectivity between the two platforms is achieved using Attunity Connect from Attunity Ltd.. The specification below may require the addition of more attributes to the existing Customer Object, which may need to be implemented on either platform.

However, with the advent of the REACH initiative (the framework for E-government in Ireland) the Customer object should also be seen as the interface to 'public service broker' that forms a major facet of REACH. Several of the responsibilities listed below are being explicitly planned for the public service broker.

Know-what responsibilities

Cases in which the customer is cited.

Relationships to other Customers, including 'mother of', 'spouse of', 'nominee for', 'legal guardian of'

Communications to and from the customer (that are not held within a specific Case e.g. advice of change of address)

Payment Methods - methods through which payments can be made to the customer e.g. bank account, Post Office details.

Addresses for communication.

Schemes in which the Customer is cited

Overpayment recovery objects pertaining to the Customer

Whether the Customer is an employee of the DSFA or other civil servant - in which case the object may only be accessed by a special unit of officers.

Know-how-to responsibilities

Find and Retrieve. This method allows the user to find an existing customer instance using any variety of search criteria available. It is implemented by wrapping an OpenVMS based specialised search facility and making it available through the Customer Object.

Communicate. The Customer object provides the ability to Communicate with the customer, using any of the specified media and addresses, currently surface mail, e-mail and telephone. This method creates a new Communication object which looks after the transmission and filing of that Communication. Communication can be in Irish if the Customer desires, so the Customer object knows the preferred language of the customer.

Scheme Object

A Scheme object is responsible for the administration of a particular benefit or set of Benefits. Scheme is an interface. Each benefit scheme that the DSFA administers will be represented by a Scheme object that implements this interface.

There are, broadly, two different forms of Scheme: composite Schemes and component Schemes. Component Schemes model individual benefits. Composite Schemes are containers that hold one or more component schemes. Thus, an instance of a Component Scheme can only exist in the context of a Composite Scheme. For the forthcoming system, we require the implementation of one new composite Scheme class (Pension) and several new component Scheme classes (Retirement Pension, Contributory Pension, Qualified Adult Allowance, Free Travel Allowance etc). Individual Schemes will vary in the kind of support that they provide to the Officer handling the claim. At the simplest level, the Scheme instance merely provides a convenient place for recording the facts and decisions taken. At a more sophisticated level, the Scheme could implement some form of rules engine and/or a spreadsheet-like calculator. However, the underlying philosophy of the design is that the system provides a workbench to leverage the skills, and increase the productivity of the Officer - not to attempt to automate a process that necessarily involves judgement.

A composite Scheme will always exist inside a Case, creating a new one if necessary, in order to be processed by an Officer. However, once the Scheme is 'In Payment', then the Case will usually not play an active role. The batch system will interact directly with the Schemes. Certain Schemes will need the ability to bring themselves up for review after a certain period, or upon certain events - via the Case mechanism.

Any Scheme (composite or component) must implement the following generic responsibilities, plus any additional responsibilities individual to their own need. Responsibilities listed below are the generic responsibilities that any Scheme must implement to conform to the interface.

Know-what responsibilities

The Customer who is claiming the benefit and any other Customers cited in the claim. (Component Schemes do not need the former since they can get it from the composite Scheme they can belong to, but they may need the latter e.g. the Customer object representing the Child or the Qualified Adult)

The Payment Method that the Customer wishes to be paid by (includes nominee payments). (Component Schemes will default to the Payment Method specified in their parent Composite Scheme, but this can be over-ridden if, for example, the customer wishes different components to be paid to different parties or different accounts. Note that for the Free Schemes, the Payment Method specifies the Service Provider and knows how to deal with that Service Provider).

Component Schemes held within this Scheme (if it is composite)

Start and End dates

Status

Any other information specific to this Scheme (or shared by its component Schemes) that cannot be obtained from the relevant Customer object.

The Case within which the Scheme is currently held

All Payments made against this Scheme.

Certificates for various decisions made by the officer, including eligibility. Review date

Know-how-to responsibilities

Request needed information. In line with the Service Delivery Model, this capability could generate a personalised form (paper or electronic) whereby the customer could confirm relevant existing details and supply any missing ones. For some schemes, this request could be going to other agencies (e.g. a school/college or doctor). The request would usually generate a standardised Communication object, filling in the fields as appropriate. Where the missing information should be held within the Customer object, then the responsibility to request the needed information is delegated to the Customer object.

Record the eligibility decision. This is equivalent to ruling the customers eligibility for the Scheme. This involves the creation of a Certificate which represents the legal decision of the Deciding Officer. The claim

cannot proceed until this stage has been passed. If the claim is disallowed, then the Scheme object continues to exist, but the Case that contains it may be closed. Deciding the claim may automatically generate an advice note, using the Customer's communicate capability.

Calculate entitlement for any specified period. This responsibility is carried out by reference to the particular Scheme being processed. It implies that the Scheme must know the rates and rules for previous years, not just the current year. It must also know the payment frequencies which apply to the particular scheme. As new rates and rules come into force, these will be added to the Scheme definition. If the new rules and rates follow the same structure as the previous ones, then this can be thought of as just adding a row to a table. If they introduce new structures then the modifications will be more complex. Note, however, that all changes to scheme rates and rules are contained within the particular Scheme - they do not spill over into the Customer or Payment objects. The calculate entitlement responsibility is used as a prelude to generating a payment for that period, but may also be used just to advise the Customer of how much they are due.

Calculate claim start date. Where there has been a delay in submitting a claim, some backdating of entitlement is permitted. A set of rules exist to calculate the backdated start date.

Generate new payment for a given period. Depending on the Payment Method selected, this capability will typically be invoked by an external batch process running at a range of frequencies e.g. weekly, fortnightly or monthly. Application of taxation rules (with reference to the Customer's taxation status) may be an embedded part of this responsibility.

Generate a schedule of payments. This method will be used when the preferred method of payment is a book (i.e. it is necessary to generate payment vouchers at regular intervals e.g. every 6 months or every year or even on a once-off basis e.g. Christmas Bonus. The frequency of voucher generation and the number of vouchers in a book will vary from scheme to scheme). It may also be used during transition, for compatibility with existing systems.

Generate difference payment for a given period. This method will invoke the Generate New Payment, but will then net the amount against any existing payments for that Scheme for the same period. This will be used when, for example, the customer's circumstances change after a schedule of forward payments has been generated (e.g. new child born during the year). This responsibility can also produce a negative Payment (i.e. an overpayment to be recovered). Note: This is only for making corrections to future payments, and within a single scheme. The general handling of overpayment recovery is handled by a dedicated Overpayment Recovery Scheme.

Calculate compensation. In certain case where arrears are due, it may be decided to pay compensation in respect of loss of purchasing power for the period between the start of entitlement and the date of decision. Compensation is based on a standard formula. (Note: This responsibility may or may not be proper to this object. This will need to be examined in the modelling exercise which will take place early in the project).

Split payments in accordance with individualisation or other legal requirements. This could be achieved by generating separate payments based upon a percentage split agreed with the Customers involved.

Handle Exchequer Cases. Exchequer cases occur where an entitlement exists to a reduced rate contribution based pension but where the Customer concerned would be financially better off claiming a means tested scheme. In these cases, the reduced rate contribution based scheme is set up but is not put into payment. Instead, it is used to calculate entitlement and an accounting adjustment is made between it and the means tested scheme.

Correct an overlap. This means generating an Overlap object that will, effectively, transfer surplus payments made under this Scheme onto another Scheme. There may be several such methods e.g.

Generate an overlap to a value specified by the officer

Generate an overlap to transfer a specified set of Payments

Communication Object

The Communication object models a single communication, for example between an Officer and a Customer, ingoing or outgoing, or internal. The role of the Communication object is not just to allow such communications to be created, but also to allow them to be filed.

Communications may be incoming or outgoing. Transmission options implemented in Phase 1 are surface post and e-mail. In addition, the Communication Object is used to record remarks. A Remark is a Communication that has no recipient. It is typically made within a Scheme, Case or Customer object.

The transmission mechanism for a Communication is achieved through the Address object. The same user interface is used regardless of the transmission channel chosen.

Know-what responsibilities

Recipient's Address (obtained from the list of Addresses contained by the recipient object i.e. a Customer, Officer or any other object that is communicable). The user may choose the particular address, but it will default to the first entry in the customer's list.

Sender's Address (obtained from the sending object). This will default to the first entry in the sender's address list that is the same type as recipient's address. (Although all written communications will list various ways of replying).

Subject. If the Communication was generated inside a Case, then that object will be recorded as the subject. This will not only allow the Communication to be filed in the right place, but will also potentially allow any reply to be matched up. Potentially, this field could hold other context objects.

Date

Status: draft, sent, received, returned, standard letter (read-only) etc.

Content. Text will be held in some generalised mark-up language (e.g. HTML).

Know-how-to responsibilities

Transmit. Execution of this responsibility is fulfilled through Address object

Edit. Allows text contents to be created and edited.

Reply.

Forward. This is done by creating a new communication that has current one as the Subject.

Retrieve (class responsibility). Previous communications will be retrieved from lists held in the Customer, Officer or Case objects. However, in the future, there will need to be a class responsibility to retrieve Communications by content (i.e. text search) but this is not in scope for this Phase.

Attach a file such as a scanned image to a Communication

Confirm successful delivery (also fulfilled in collaboration with Address).

Sign. Create a Certificate digital signature of the sender. Additionally, this responsibility may append a digitised image of a physical signature, if desired.

Copy. This copies a whole Communication

Lock. This turns a Communication into a standard read- communication that can be copied.

Append. Used to create a letter from standard paragraphs.

Officer Object

The Officer object is the single point of contact for information and functionality associated with an individual (employee of the Department or an associate) that may use the information system. There is one instance for each such individual.

Users of the system have their own Officer object readily accessible, as this is the means for logging on and off, and for storing the officer's personal desktop view. In addition the Officer object provides access to current workload.

Officer object may be 'virtual', that is instances that represent roles and/or departmental sections e.g. 'Claims Registration Section'.

Know-what responsibilities

Relationships to other Officers. This includes supervisors, supervised, and peers.

Cases**.** This means cases that are currently assigned to the officer.

Communications to or from the Officer

Addresses for communication

Roles fulfilled.

Know-how-to responsibilities

Find and Retrieve. These responsibilities are broadly similar to those specified for Customer.

Log-on and off.

Capture and recall the user's desktop.

Present caseload. This responsibility can show all cases currently assigned to the Officer broken down by various categories including current status.

Manage in and out boxes

Manage authorisation levels. Authorisation (to perform a specific method on a specific object) will be done by a system-wide authentication and authorisation server. However, the Officer object will be a principal user interface onto this server i.e. the means through which the authorisation levels for specific roles and/or individuals are specified.

Communicate. This works in the same way as the communicate responsibility of the Customer object.

Create a Certificate to record the basis of the Officer's decision.

Payment Object

A Payment object represents a single payment from a payer (by default, the Department) to the payee (usually a Customer). A Payment is in many ways analogous to a Communication and shares some of its structure. Thus, the role of the Address in a Communication, is replaced by a Payment Method, where that may represent a cheque, electronic funds transfer, electronic information transfer or a voucher (the latter usually forming part of a payment book).

The amount of the payment will have been determined by whatever created the Payment object (e.g. a Scheme, or, in rare cases, directly by an authorised Officer), along with the date due. Payment can represent negative amounts for the purposes of recovering an overpayment

Payments are generally created at the lowest level possible to enable them to be posted accurately into the financial accounting system. Thus, a claim may give rise to the creation of several Payment objects representing the different component Schemes such as RP, CP, Qualified Adult Allowance or Child Dependent Allowance. Payments that have been created but not yet executed and which have the same Payee, may be combined or merged with other Payments within the same payment period to form a single net transfer.

Know-what responsibilities

Scheme that caused the Payment to be created

Payee's Payment Method. The descriptive label of the Payment Method includes the name of the Payee, and can provide direct access to the object representing the payee (e.g. a Customer or Agency).

Payment identification. For example, cheque number or PPO voucher number.

Component Payments. This means that any composite payment knows what other payments it has been made up from.

Amount (expressed in a currency).

Status (issued, paid, stopped, reconciled)

Stop Reason, if status indicates that the payment has been stopped

Payment Type. This indicates if the payment is a regular payment, a replacement payment, grant payment etc. This is a free-form field the contents of which are typically determined by the Scheme that creates the Payment.

Payment period that it relates to.

Date due. (It may be that this is a function of the payment period e.g. first Tuesday in the period).

Know-how-to responsibilities

Merge with another payment (subject to rules). Typically Payment instances are created at the level of scheme elements (e.g. child dependent, fuel allowance) and then merged to form a single payment which is transferred to the Customer.

Post into the financial accounting systems.

Authorise. Most payments will be generated within Schemes, which will look after their own levels of authorisation. However, it may be appropriate to put in some additional generic concepts of authorisation into the Payment object itself e.g. for payments over €5,000.

Stop the individual payment or the entire book if the payment method is PPO.

Issue in the manner appropriate to the Payment Method

Case Object

The Case Object is currently the mechanism whereby a Scheme instance can be linked to an Officer. Case can act as a holder for any supporting communications (including Remarks) and could in future hold scanned images of other documents associated with Schemes being processed, but which may not be explicitly held within the Scheme. However, the work contained in a Case does not have to be related to a Scheme. Instead it can be any type of Departmental work from correspondence to investigations. Case provides certain workflow-like characteristics, including the ability to forward the case on to another officer. .

Know-what responsibilities

The Officer currently responsible for the case.

The Officer to whom the Case was previously assigned (if any)

Schemes that form part of the case (which in turn know the Customers)

Communications relating to the case

Other relevant documents (including, potentially, scanned images) and notes.

Current status. This might include: Pending - customer, Pending - other, In payment, Closed etc.

Review Date, the date when the case is to be brought to attention for review. This date will usually be determined by the Officer.

Know-how-to responsibilities

Refer to another Officer. This referral may be temporary (e.g. for authorisation to proceed) or a permanent handoff. The nature of the referral will make that clear. The referral may be initiated merely by dragging the Case object onto the appropriate Officer object. As well as changing the Officer assigned, the referral will generate a standardised Communication to appear in the in-tray of the recipient.

APPENDIX III. SURVEY OF IT MANAGERS AT THE DSFA

Description

In January and February 2003, the author formally interviewed seven managers within the IT function of the DSFA, including the head of the IT, all of whom had had a significant involvement in the conception and commissioning of the *Naked Object Architecture.* All interviews were conducted face to face in Dublin.

The questionnaire is shown overleaf. After the background questions, all questions take the form of a proposition, to which the interviewee was asked to give one of five responses:

1. Agree strongly 2. Agree somewhat 3. Neither agree nor disagree 4. Disagree somewhat 5. Disagree strongly

Questionnaire

Background

Name:

Normal operating role:

Responsibility in relation to the Naked Object Architecture project:

Overall satisfaction with the system as delivered

Q1. Overall, the EOA (not the CB system) as delivered has met our expectations for an EOA that we envisaged when we issued the RFT

Q2. Overall, the CB system as delivered has met our expectations as an application

Q3. The system has already demonstrated the ability to support strategic business agility

Q4. The system has already demonstrated the ability to support operational business agility

Q5. To the extent that either of these forms of agility have not yet been demonstrated, our expectation that they will yet be demonstrated remains as strong as at the start of the project

The development process

Q6. The EOA approach to designing the system directly facilitated communication between the developers and the users

Q7. Our IT staff were able to adapt easily to the fully object-oriented way of thinking

Q8. Specifying the system entirely as business objects and their responsibilities was an effective approach

Q9. The constraints of the EOA approach resulted in a better object model than we would probably have achieved using other approaches to object modelling

Q10. The process could have benefited from greater use of prototyping

Q11. The process could have benefited from a more iterative approach to delivery

Q12. The process could have benefited from a more formal approach to testing the functional completeness of the model

The future

Given that the basic EOA infrastructure now exists, and with suitable modifications to the development process, my expectations are that any subsequent business system developed on the EOA will be:

Q13. Developed faster than achievable using a more conventional approach

Q14. Less expensive than using a more conventional approach

Q15. Achieve more commonality (with existing EOA systems)

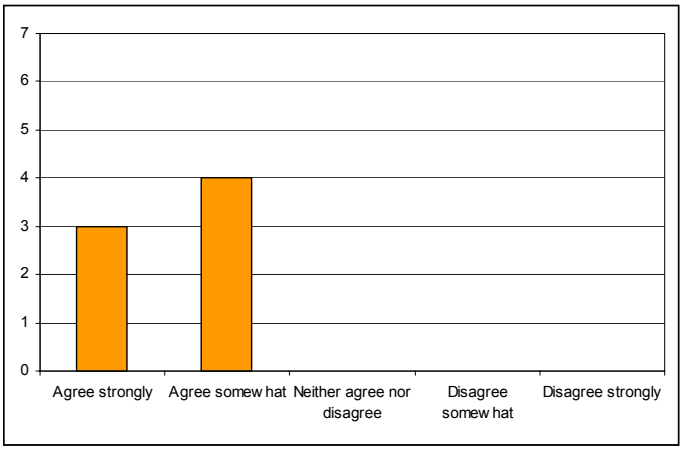
Q16. More comprehensively tested than would using a more conventional approach

Summary of responses

Question No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Individual responses																
IT Manager 1	2	2	3	1	2	2	4	1	2	1	1	1	2	2	1	4
IT Manager 2	2	2	3	1	2	2	4	1	3	1	1	3	2	4	1	1
IT Manager 3	1	1	2	2	2	1	2	1	3	1	3	3	2	2	1	3
IT Manager 4	2	1	1	1	1	2	1	1	3	1	3	2	1	3	1	5
IT Manager 5	1	1	2	1	1	1	2	2	3	1	1	1	5	3	1	3
IT Manager 6	2	2	4	4	3	1	2	1	1	1	3	2	2	3	1	4
IT Manager 7	1	1	1	1	2	1	2	1	1	1	1	3	2	3	3	3
Number of responses	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Frequency																
1. Agree strongly	3	4	2	5	2	4	1	6	2	7	4	2	1	0	6	1
2. Agree somewhat	4	3	2	1	4	3	4	1	1	0	0	2	5	2	0	0
3. Neither agree/disagree	0	0	2	0	1	0	0	0	4	0	3	3	0	4	1	3
4. Disagree somewhat	0	0	1	1	0	0	2	0	0	0	0	0	0	1	0	2
5. Disagree strongly	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
Median response	2	1	2	1	2	1	2	1	3	1	1	2	2	3	1	3

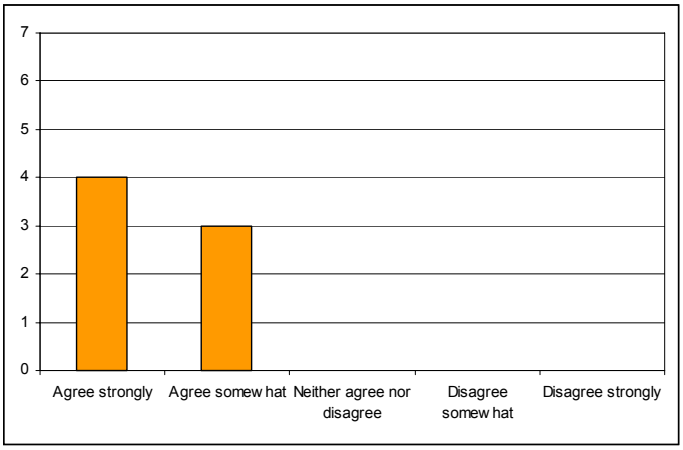
Charts and commentary

Q1. Overall, the NOA (not the Child Benefit Administration system) as delivered has met our expectations for a NOA that we envisaged when we issued the RFT



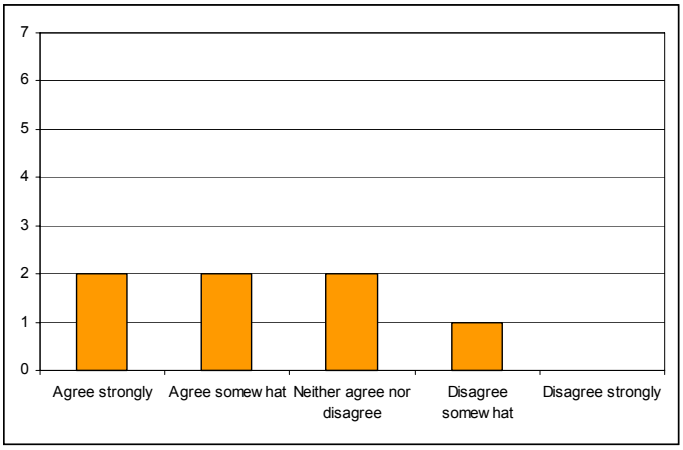
Commentary: The architecture clearly implements both the broad concept and the specific principles of an NOA as defined in the RFT. However there is some concern about how purely the object-oriented design of the CB system has been carried into the coding level.

Q2. Overall, the CB system as delivered has met our expectations as an application



Commentary: This is borne out by the comments of the business customers for the CB system (separate analysis).

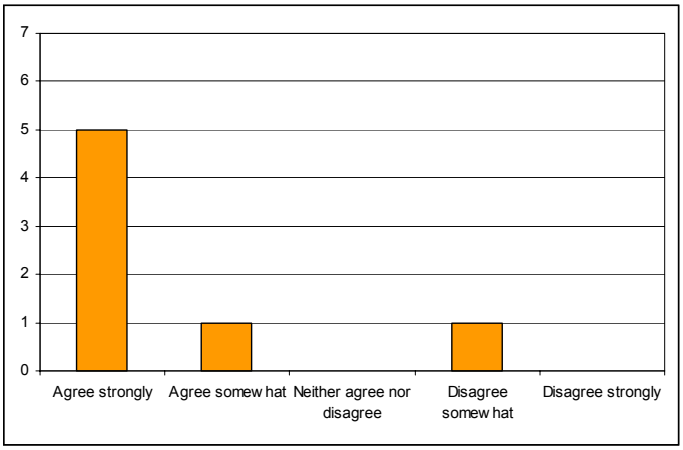
Q3. The system has already demonstrated the ability to support strategic business agility



Commentary: Most people felt that this question was somewhat premature. With the system having only been live for 3 months, the system has yet to be really tested for business agility. Those who agreed with the proposition justified it on the grounds that some of the business

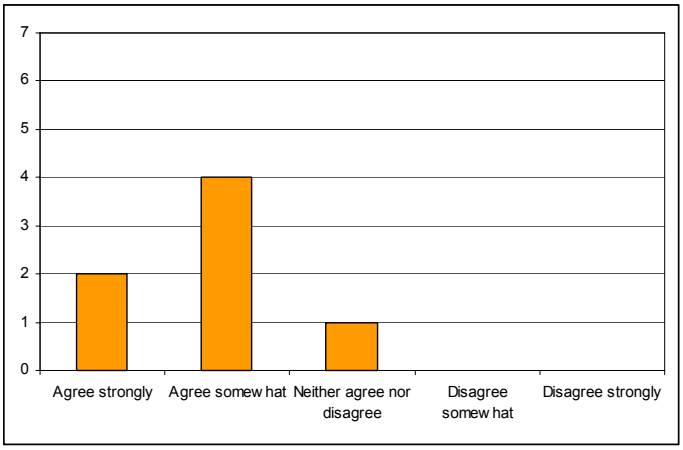
requirements had changed quite late in the project and that this had caused much less of a problem than might have been expected on a more conventional system. Also, some significant new requirements have emerged since the system went live (in particular the GRO online initiative, which will trigger the claims for additional children automatically). Although these requirements have yet to be coded, it is already becoming clear how the business object model and architecture will accommodate them.

Q4. The system has already demonstrated the ability to support operational business agility



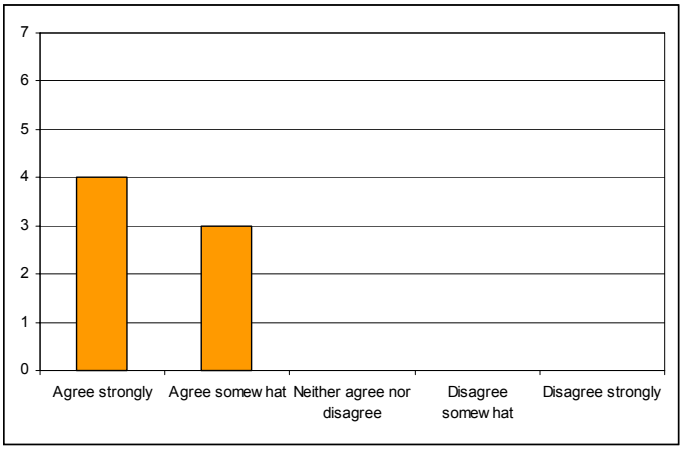
Commentary: The system has already demonstrated the ability to support operational business agility in two senses. First it has permitted a massive organizational change (the movement from a production line into 'once and done' claim processing, and the shift from specialized to multi-skilled teams). Secondly, as brought out strongly by the user survey, the system provides users with a great deal of flexibility in how they manage their day to day work. (Author's note: In this question only, I suspect that the single 'disagree somewhat' answer may reflect a misunderstanding of the specific term 'operational agility').

Q5. To the extent that either of these forms of agility have not yet been demonstrated, our expectation that they will yet be demonstrated remains as strong as at the start of the project



Commentary: Most respondents believe that the agility of the system will show increasingly as future demands are placed on it. The 'neither agree nor disagree' answer in this case reflected a deeper concern that while the object model may be very agile, the way that it has been implemented may turn out to restrict that agility.

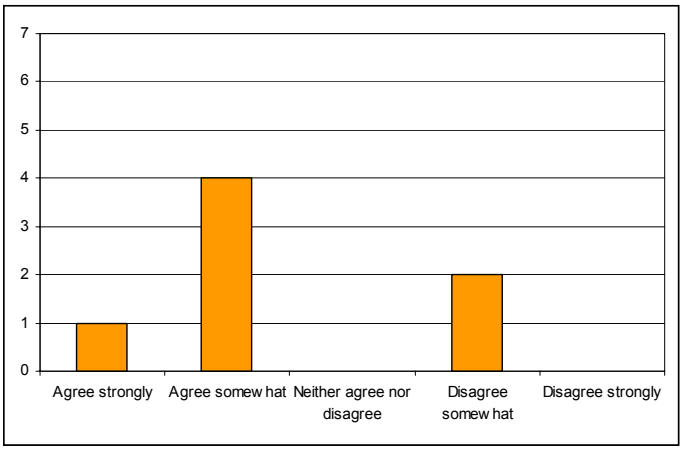
Q6. The NOA approach to designing the system directly facilitated communication between the developers and the users



Commentary: The responses endorse the strength of this approach for requirements analysis, and these are consistent with the answers from the business customer representatives. Most people commented that the early exploratory stages (i.e. before the commencement of the CB project)

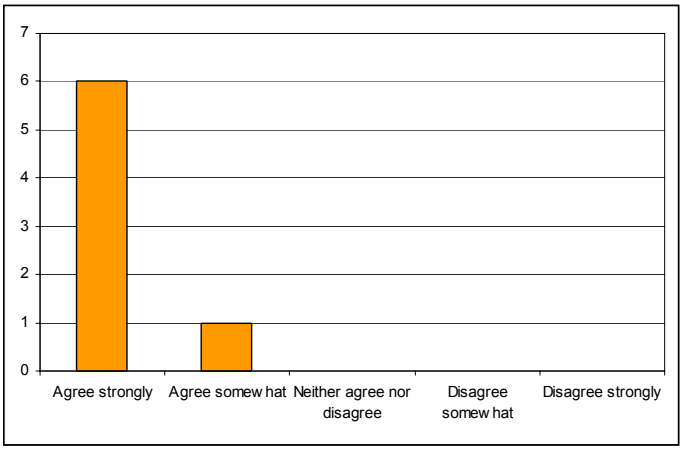
were more effective in this regard than the main analysis of the CB project. The latter could have been as effective had more prototyping been employed,

Q7. Our IT staff were able to adapt easily to the fully object-oriented way of thinking



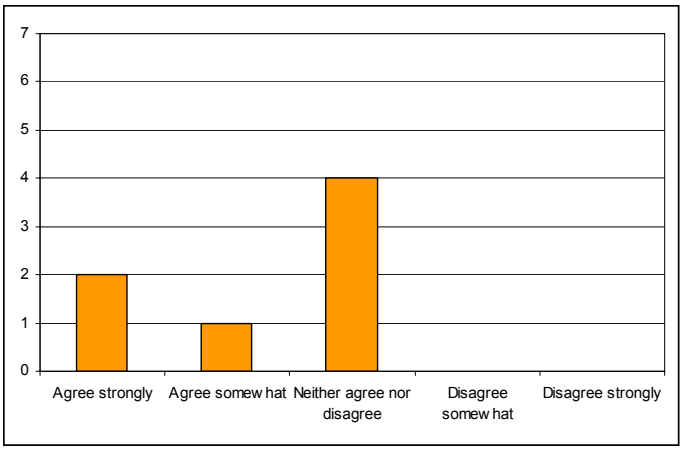
Commentary: General opinion here was that a small management team, which had the benefit of regular mentoring from the author, had managed the transition quite easily. The mistake, if any, was assuming that a broader group within IT could pick up the ideas without the benefit of such mentoring.

Q8. Specifying the system entirely as business objects and their responsibilities was an effective approach



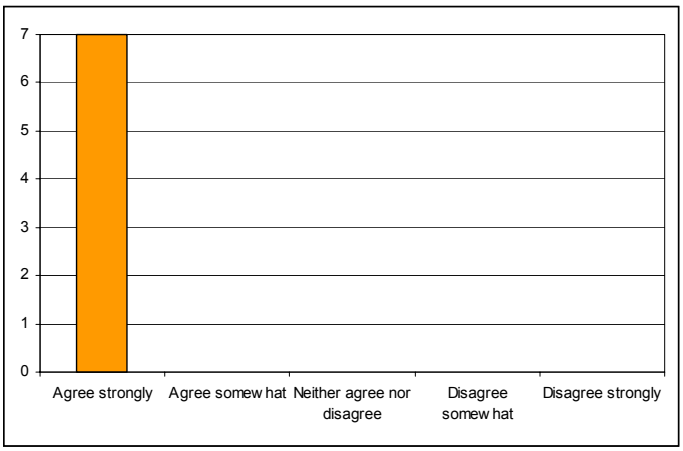
Commentary: Bearing in mind how different this approach was from the Department's previous experience in specifying business systems, this endorsement of the approach is striking.

Q9. The constraints of the NOA approach resulted in a better object model than we would probably have achieved using other approaches to object modelling



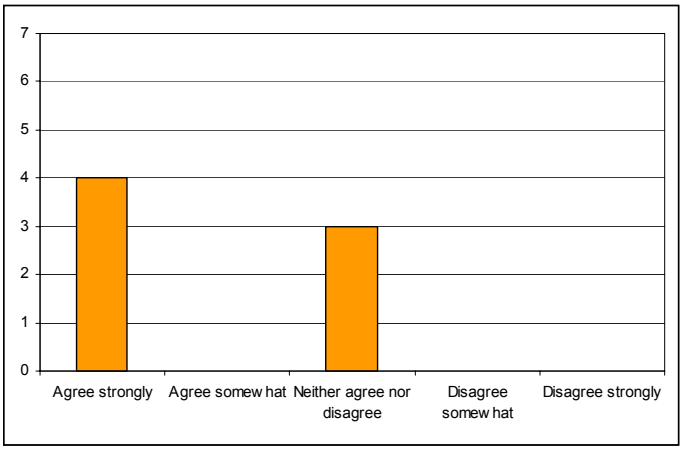
Commentary: The majority of respondents stated that they had no previous experience of object modelling approaches against which to make comparisons. Of the three that answered positively, the greater the experience of conventional OO methods the stronger the endorsement of the new approach.

Q10. The process could have benefited from greater use of prototyping



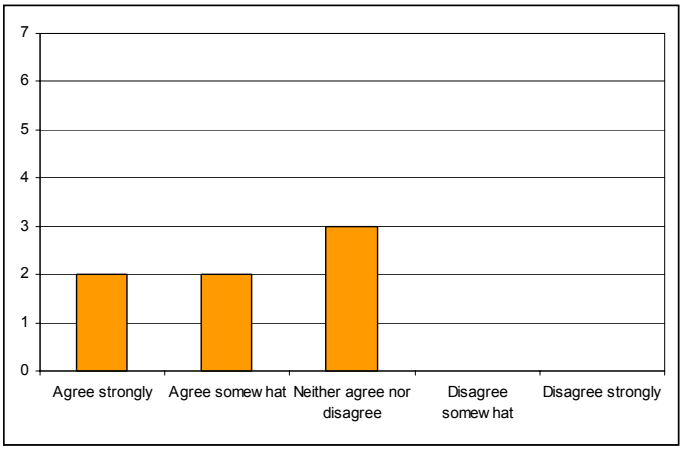
Commentary: None needed.

Q11. The process could have benefited from a more iterative approach to delivery



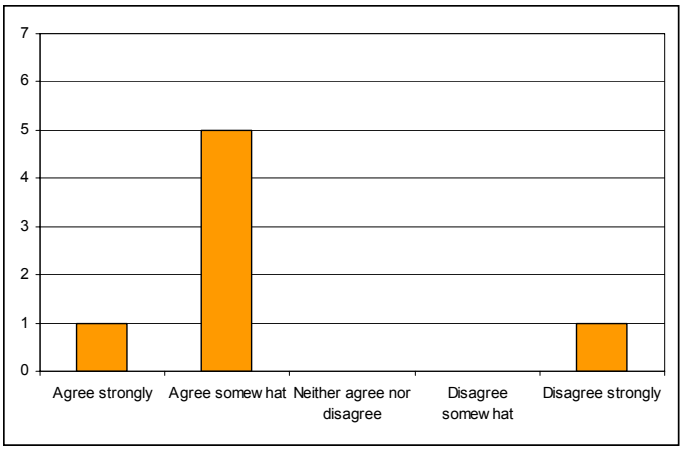
Commentary: While some felt strongly that a more iterative approach to deliver would be beneficial, others were more cautious. The latter group pointed out that while iterative delivery sounds like a good approach, the term is meaningless unless you can state clearly what the nature of the iterations is. One said that they would agree if 'iterative' meant 'incremental'. Another if the iterations meant stronger commitment to proving the batch processing side of the functionality earlier. One also felt that the culture of the Department ('right at all costs') would actually work against true iterative delivery.

Q12. The process could have benefited from a more formal approach to testing the functional completeness of the model



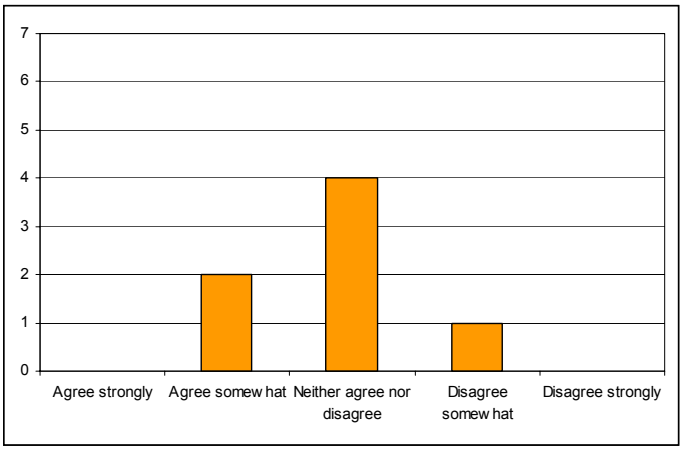
Commentary: About the only clear conclusion that could be drawn from the comments in response to this proposition, was that a different approach to testing was warranted, but not necessarily 'a more formal approach to testing the functional completeness of the model'.

Q13. My expectation is that any subsequent business system developed on the NOA will be developed faster than achievable using a more conventional approach



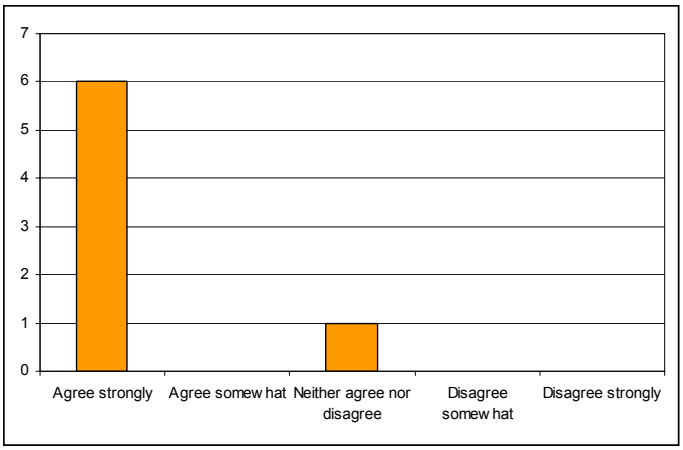
Commentary: The caution here is that the planned Phase II may still be quite early in the learning curve and the real productivity benefits may not show up until Phase III and IV.

Q14. My expectation is that any subsequent business system developed on the NOA will be less expensive than using a more conventional approach



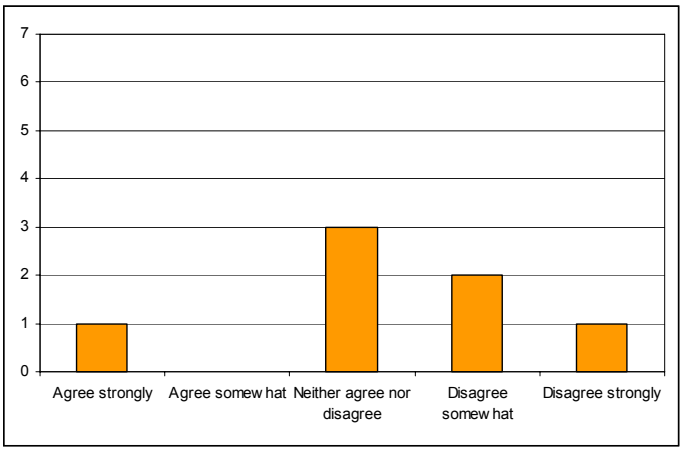
Commentary: No conclusive result on this proposition. Once cause is that the Department does not have a strongly cost-oriented model of its systems, so hard comparison would be difficult.

Q15. My expectation is that any subsequent business system developed on the NOA will be achieve more commonality (with existing NOA systems)



Commentary: Some evidence for this position already exists in the commonality between the business object model for Phase II (the Pensions systems), as described in the RFT issued in August 2002, and the business object model for Phase I (the Child Benefit Administration system).

Q16. My expectation is that any subsequent business system developed on the NOA will be more comprehensively tested than would using a more conventional approach



Commentary: On reflection, this was a poorly phrased proposition. As more than one interviewee pointed out, thorough testing is a historical strength of the DSFA's IT unit. There is nothing in the NOA that would inherently improve this commitment.

APPENDIX IV. SURVEY OF BUSINESS MANAGERS AT THE DSFA

Description

In February 2003, the author formally interviewed three managers within the DSFA who had been directly involved in the commissioning of the Child Benefit Administration system - the first application built using the naked object approach. All of them had been involved in the business object modelling process. All interviews were conducted face to face in Letterkenny.

The questionnaire is shown overleaf. After the background questions, all questions take the form of a proposition, to which the interviewee was asked to give one of five responses:

1. Agree strongly 2. Agree somewhat 3. Neither agree nor disagree 4. Disagree somewhat 5. Disagree strongly

Questionnaire

Name:

Your normal operating role?

Your role/involvement in the business object modelling?

Prior experience, if any, of object modelling?

Prior experience, if any, of specifying new business systems?

Q1. I found it reasonably easy to get involved in the business object modelling process despite having little or no prior experience of this activity

Q2. The actual objects and the responsibilities of those objects, as listed in the BOM, are reasonably easy to understand

Q3. The idea that all business functionality must be 'encapsulated' on business objects was not problematic

Q4. I found it difficult to see how certain requirements could be specified in terms of objects and responsibilities

Q5. During modelling I found it reasonably easy to envisage how the business objects could be used to achieve actual business tasks

Q6. I would like to have seen more use made of prototyping to test out the business scenarios

Q7. The relationship between the CB system as delivered and the business object model is clear

Q8. The business object model has proven to be an effective way to represent the business needs of a new system

Q9. I can envisage how a range of possible future business changes might be realised through the object model

Q10. If I was involved in a specifying an unrelated business system in future (e.g. for another organisation) I would recommend adopting the business object modelling approach

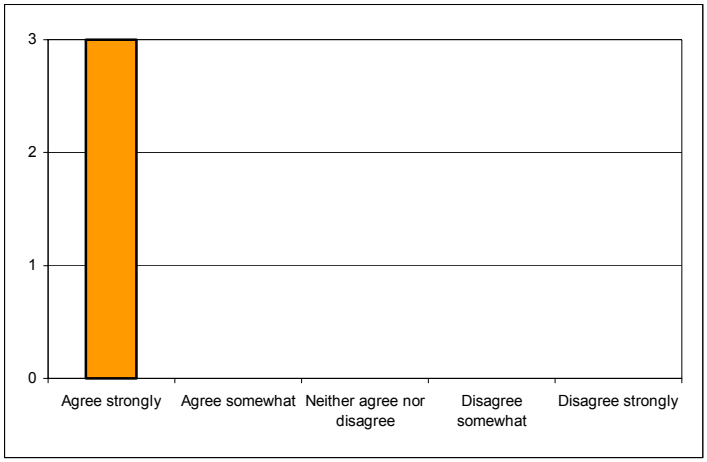
Summary of responses

Question number >>>	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Manager 1	1	2	2	3	2	1	1	1	2	1

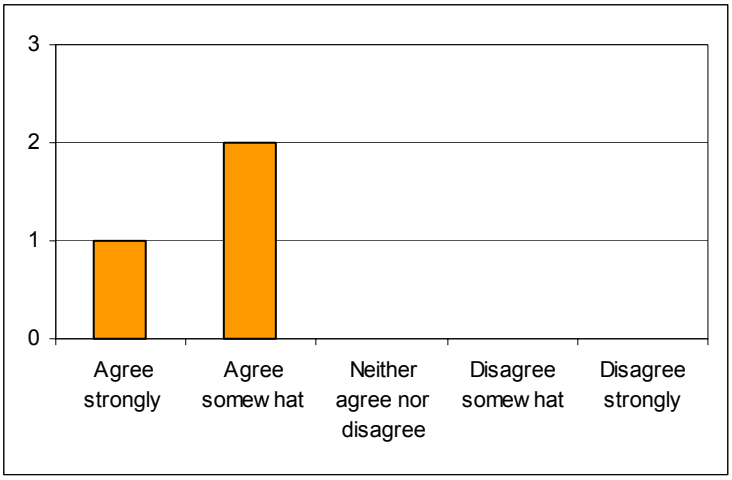
Manager 2	1	2	2	4	3	1	1	1	1	2
Manager 3	1	1	1	5	4	1	1	1	3	1
Number of responses	3	3	3	3	3	3	3	3	3	3
Frequency analysis of responses										
Agree strongly	3	1	1	0	0	3	3	3	1	2
Agree somewhat	0	2	2	0	1	0	0	0	1	1
Neither agree nor disagree	0	0	0	1	1	0	0	0	1	0
Disagree somewhat	0	0	0	1	1	0	0	0	0	0
Disagree strongly	0	0	0	1	0	0	0	0	0	0
Median response	1	2	2	4	3	1	1	1	2	1

Charts

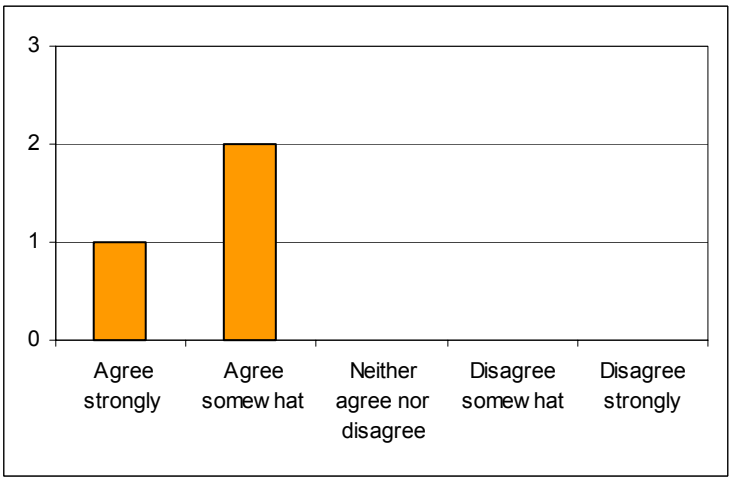
Q1. I found it reasonably easy to get involved in the business object modelling process despite having little or no prior experience of this activity



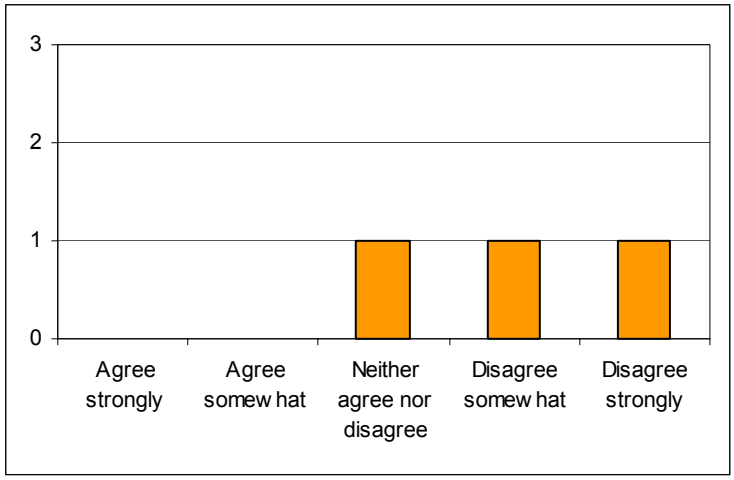
Q2. The actual objects and the responsibilities of those objects, as listed in the BOM, are reasonably easy to understand



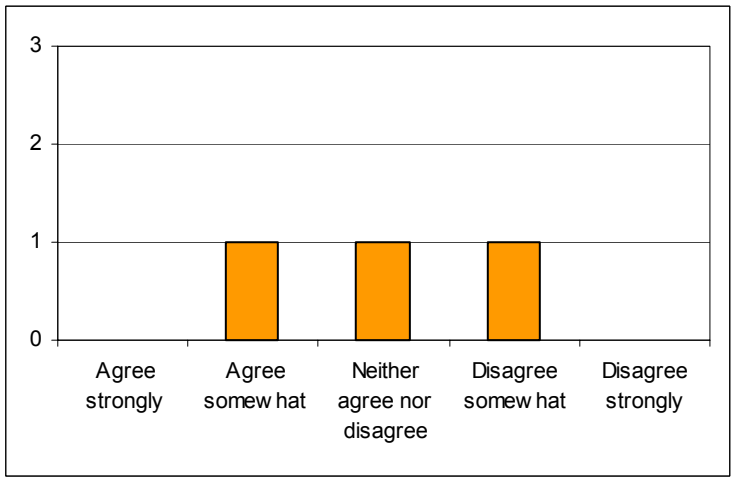
Q3.The idea that all business functionality must be 'encapsulated' on business objects was not problematic



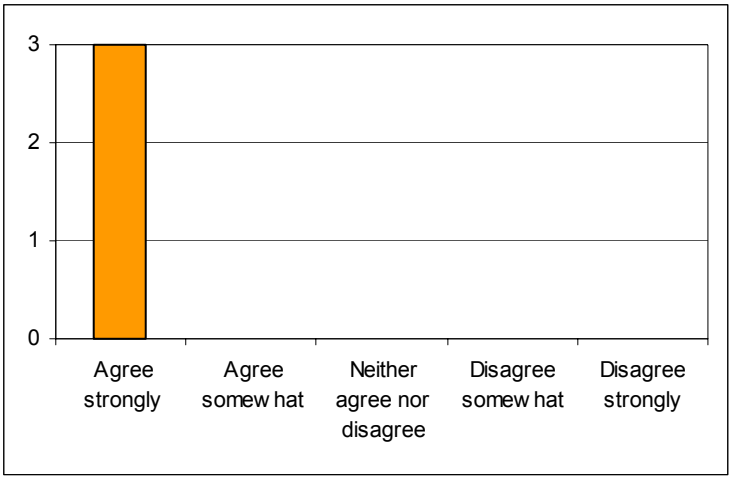
Q4. I found it difficult to see how certain requirements could be specified in terms of objects and responsibilities



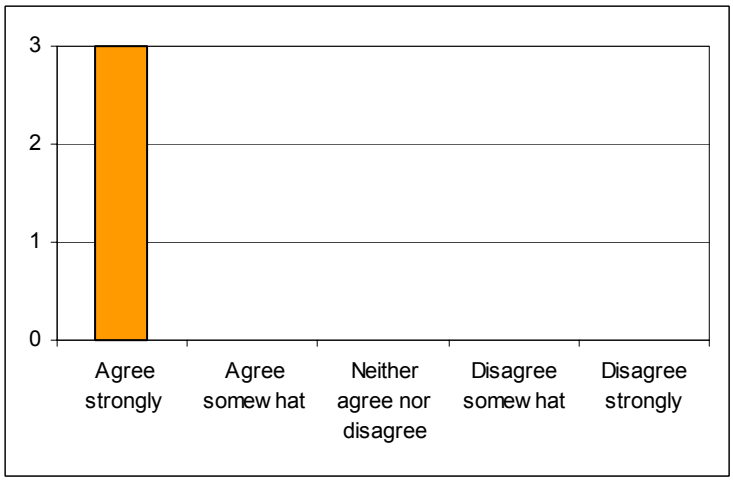
Q5. During modelling I found it reasonably easy to envisage how the business objects could be used to achieve actual business tasks



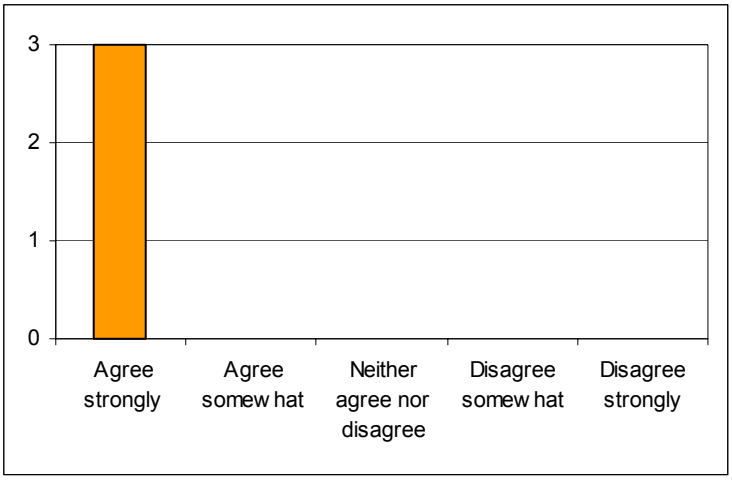
Q6. I would like to have seen more use made of prototyping to test out the business scenarios



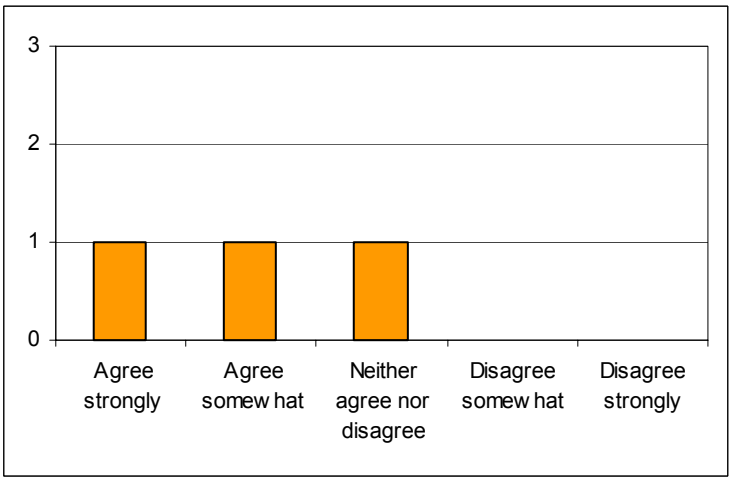
Q7. The relationship between the CB system as delivered and the business object model is clear



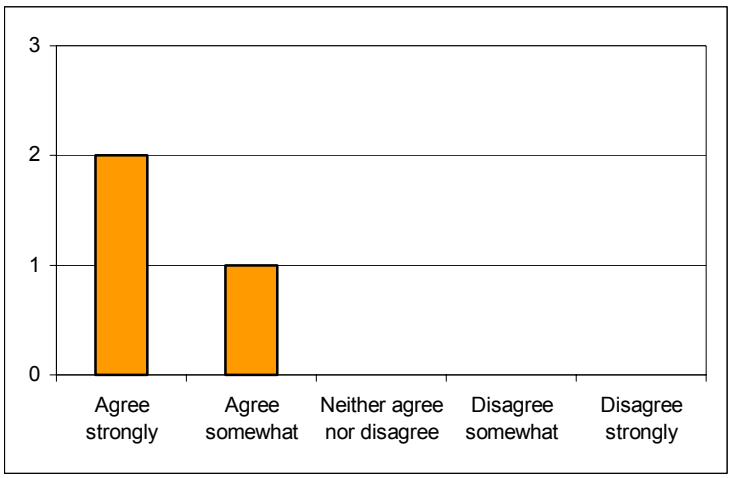
Q8. The business object model has proven to be an effective way to represent the business needs of a new system



Q9. I can envisage how a range of possible future business changes might be realised through the object model



Q10. If I was involved in a specifying an unrelated business system in future (e.g. for another organisation) I would recommend adopting the business object modelling approach



APPENDIX V. SURVEY OF USERS AT THE DSFA

Description

In February 2003, the author conducted interviews with 15 users of the new Child Benefit Administration system running on the *Naked Object Architecture,* which had been live since November 2002. The interviewees comprised 13 Clerical Officers and 2 Supervising Officers, spanning all of the following business responsibilities:

- New Claims
- Additional Child
- 16+ extensions
- Client Maintenance
- General Payments
- Data Clean-up
- Reporting

For some of the interviewees their normal role was in training or user acceptance testing. However, even these individuals had used the system for actual claims processing for several weeks in order to help clear the backlog. Those individuals were asked to respond to the questions as claims processors rather than as trainers or testers.

All interviews were conducted face to face in the Child Benefit office in Letterkenny. In each case they were interviewed at their desk, such that they could point to features of the system to clarify any responses.

The questionnaire is shown overleaf. After the background questions, all questions take the form of a proposition, to which the interviewee was asked to give one of five responses:

1. Agree strongly 2. Agree somewhat 3. Neither agree nor disagree 4. Disagree somewhat 5. Disagree strongly

Questionnaire

Background questions

Name:

i. For how long did you use the previous CB system?

ii. What was your role within the CB organisation prior to the changeover?

iii. What was your role within the CB office now?

iv. What is your civil service grade?

v. Prior to learning the new how much experience did you have of using a PC?

Learning the new system

Q1. Overall, I found learning the new system to be straightforward

Q2. The style and amount of formal training provided on the new system was appropriate to my personal needs

Q3. Since completing the formal training I have learned useful new things about the system just by experimenting

Q4. Since completing the formal training I have learned useful new things about the system from colleagues sharing their tips

Efficiency

Q5. The new system allows me to process most claims and enquiries faster than before

Effectiveness

Q6. I am already handling a broader range of claims/enquiries than before

Q7. I hope to be able to handle a broader range of claims/enquiries in future

Q8. The system enables me to manage my personal workload in a more effective manner

Q9. The system enables our team to manage our workload more effectively

Q10. The new system permits me to better deal with the needs of individual customers

Q11. The new system makes it easier for me to make more ad hoc checks in relation to a claim

Q12. The system makes me feel more empowered as an individual

Flexibility and control

Q13. I value the flexibility that the system provides in choosing how to undertake a task

Q14. I would prefer the system to guide me through the steps of a task

Q15. I would like to see more kinds of 'help' or look-up information on the system

Q16. There are certain kinds of errors that I find myself making repeatedly

The look and feel of the user interface

Q17. I like the look and feel of the user interface on this system

Q18. I often copy objects onto my desktop

Q19. I often end up with too many windows open

Q20. I like the use of 'drag and drop' to initiate actions

Overall satisfaction

Q21. (Aside from teething problems) I am generally satisfied with the new system

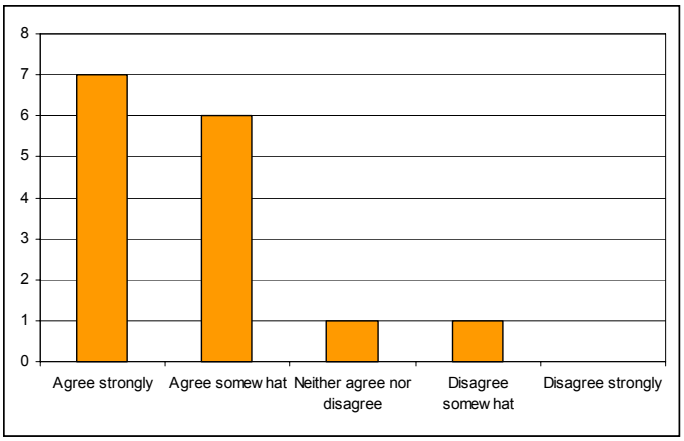
Q22. The new system contributes positively to my job satisfaction

Summary of responses

Question No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
User 1	1	2	1	2	1	1	1	2	3	2	3	2	1	5	5	3	2	1	1	1	1	1
User 2	1	2	3	3	1	3	3	1	2	1	1	2	1	5	2	2	1	2	5	1	1	1
User 3	4	3	4	4	1	1	3	3	2	2	1	3	2	4	5	5	1	2	5	1	1	2
User 4	1	2	1	2	2	1	1	4	3	3	2	3	1	5	4	3	1	1	5	1	1	2
User 5	2	3	1	1	3	1	1	3	1	1	3	3	1	3	3	2	4	5	5	3	1	2
User 6	2	2	2	2	3	5	1	1	3	2	4	2	3	4	2	3	2	1	5	1	2	2
User 7	1	3	1	1	1	1	2	1	1	1	3	1	1	4	2	2	2	1	5	1	1	1
User 8	2	4	1	1	1	1	1	2	3	1	1	3	1	3	3	5	1	1	4	1	1	2
User 9	2	2	1	1	2	1	1	1	3	1	1	2	1	5	5	5	1	5	5	1	1	1
User 10	3	3	1	1	1	1	3	1	1	1	2	1	1	3	3	2	1	4	4	1	1	1
User 11	1	3	1	1	1	3	3	1	3	2	2	1	1	4	4	4	1	5	5	1	1	1
User 12	2	2	5	1	3	4	1	1	1	1	3	1	4	4	2	2	2	5	5	1	2	1
User 13	1	2	4	2	1	5	1	1	3	3	4	3	2	4	5	5	1	1	5	1	1	1
User 14	1	1	1	1	1	1	1	1	1	1	1	1	1	5	5	5	1	4	5	1	1	1
User 15	2	1	1	1	1	5	1	1	3	2	4	3	3	5	5	5	2	5	5	2	1	1
Total responses	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
Frequency																						
1. Agr. strongly	7	2	10	9	10	9	10	10	5	8	5	5	10	0	0	0	9	6	1	13	13	10
2. Agr. somewhat	6	7	1	4	2	0	1	2	2	5	3	4	2	0	4	5	5	2	0	1	2	5
3. Neither agr/dis	1	5	1	1	3	2	4	2	8	2	4	6	2	3	3	3	0	0	0	1	0	0
4. Dis. somewhat	1	1	2	1	0	1	0	1	0	0	3	0	1	6	2	1	1	2	2	0	0	0
5. Dis. strongly	0	0	1	0	0	3	0	0	0	0	0	0	0	6	6	6	0	5	12	0	0	0
Median response	**2**	**2**	**1**	**1**	**1**	**1**	**1**	**1**	**3**	**1**	**2**	**2**	**1**	**4**	**4**	**3**	**1**	**2**	**5**	**1**	**1**	**1**

Charts and commentary

Q1. Overall, I found learning the new system to be straightforward



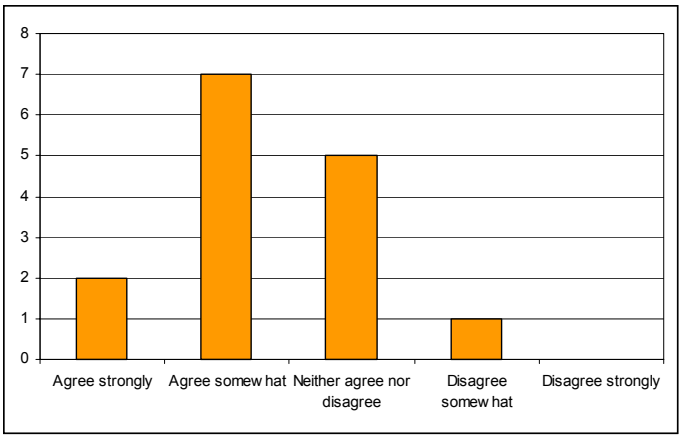
9.3.1.1.1.1 Qualifications or comments made by individual respondents:

Neither agree nor disagree. Being on the SDM team, we were brought into the project quite early

Disagree somewhat. For the first week (advance training in Dublin) I was completely lost.

Commentary: Most people report that they found leaning the system to be straightforward. There was no discernable correlation between the answers to this question and the amount of experience of using a PC prior to the training on this system. This may be surprising to the trainers who reported that those with no prior PC experience found the initial training quite difficult. Probably what this says is that any initial difficulty was quite short lived,

Q2. The style and amount of formal training provided on the new system was appropriate to my personal needs



Qualifications or comments made by individual respondents:

Agree somewhat. I had more training than most: Pilot, Refresher and Top-up.

Agree somewhat. Could have done with a bit more training.

Agree somewhat. The system training was fine, but there was a lot of new work training at the same time.

Agree somewhat. We went straight from the basic training into user testing which was a big jump.

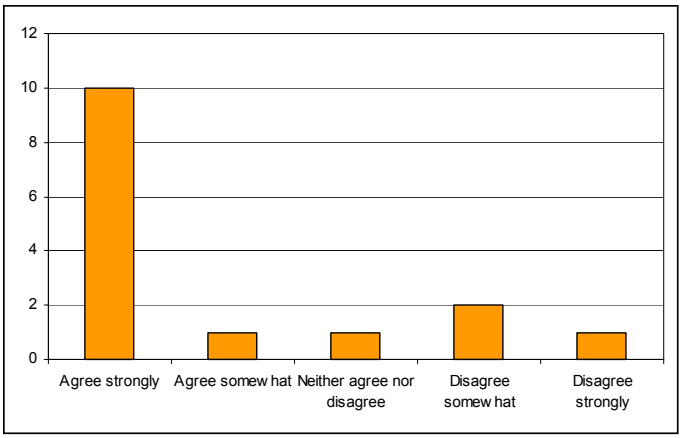
Agree somewhat. Main problem was the long lag between the main training effort and the system going live (though we did get a brief refresher course).

Neither agree nor disagree. I was trained as a CO, but subsequently promoted into a different role.

Disagree somewhat. We [i.e. the trainers] were not shown the new system from a CB point of view - we were shown it in very general terms. The users were shown the system from a CB point of view.

Commentary: Most people would have liked a little more training. However, a big factor in these answers was almost certainly the long delay between the initial training and the final go-live date.

Q3. Since completing the formal training I have learned useful new things about the system just by experimenting



Qualifications or comments made by individual respondents:

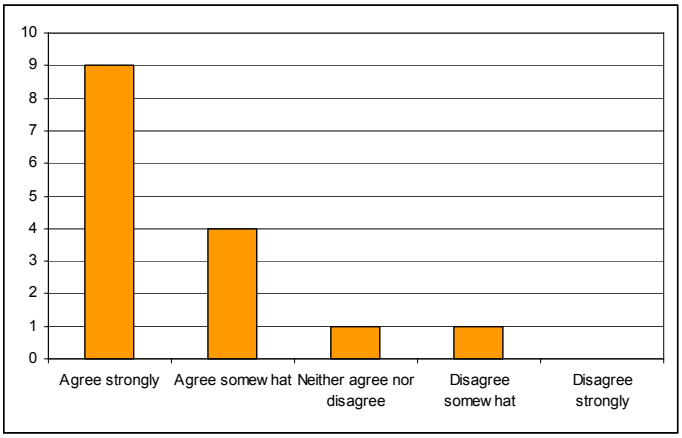
Agree strongly. I have learnt practically everything by experimenting!

Neither agree nor disagree. Yes if this refers to the training I was given in relation to UAT; I did not go through the normal user training.

Disagree strongly. With the backlog to be cleared there is no time to experiment.

Commentary: The system certainly seems to encourage learning by exploration. Not everyone likes to explore, or feels the need to explore, perhaps.

Q4. Since completing the formal training I have learned useful new things about the system from colleagues sharing their tips



Qualifications or comments made by individual respondents:

Agree strongly. The style of the system encourages this.

Agree strongly. But that's part of our job [in SDM team].

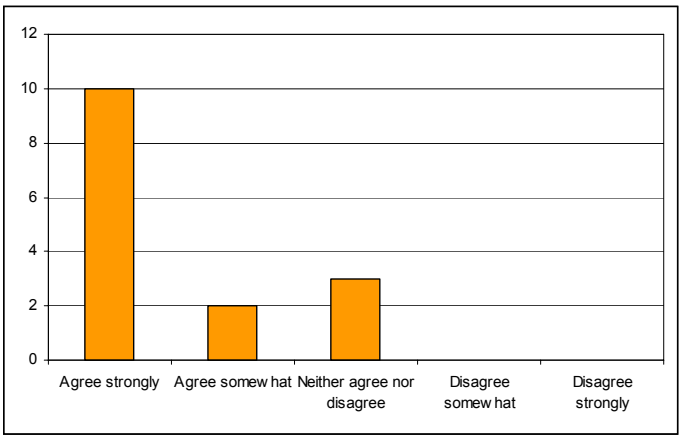
Agree strongly - mostly from the UAT/Trainers.

Agree somewhat. There is no real mechanism for sharing tips between people performing the same role within different teams

9.3.1.1.1.2 Commentary

There seems to be quite a bit of sharing or tips. There should be even more opportunity to do this when the teams move into multi-skilling.

Q5. The new system allows me to process most claims and enquiries faster than before



Qualifications or comments made by individual respondents:

Agree strongly. 16+ claims are much faster. New Claims are a bit slower due to requirement to input more data e.g. Phone numbers. If you have to wait for a PPSN then you have to wait for the overnight batch run and it can sometimes take 2 days. If the claimant knows their PPSN it is probably not slower.

Agree strongly. Especially Claim Maintenance and General Payment.

Agree strongly . On all types.

Agree strongly. On all tasks.

Agree strongly. (Additional child.)

Agree strongly. For all General Payments. We are now up to date on GPs - when we started we had a 6-7 week backlog.

Agree strongly. All types

Agree strongly. You can never catch the system out on applying the rates. It seems to get it right no matter how complicated the case.

Agree somewhat. New Claims and 16+ are quicker. General payments (e.g. separations), no real improvement.

Agree somewhat. For Client Maintenance and 16+: (1). For New Claims: (3). Data clean-up is mostly done on CRS, but the EOA is used for look-up purposes. So (3) also.

Neither agree nor disagree. 16+ is slower but only because of having introduced the School field (and searching for the school takes time). Claim maintenance is very fast.

Neither agree nor disagree. A complex new claim can take longer, due to the need to cycle through each child. [The refresh problem exacerbates this.]

Neither agree nor disagree. Less awkward, but not necessarily faster. However, the new system is more foolproof - we end up making fewer overpayments.

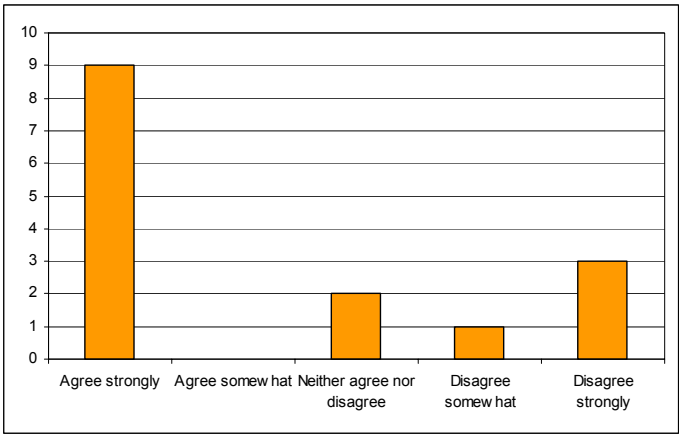
Analysis

General agreement that the new system is faster overall, but disagreement on which types are faster. The following are cited as definitely faster: New claims, Additional Child, 16+, Claim Maintenance, General Payment

However, the following are cited as either slower or no faster than before: New Claims, 16+, General Payments

With then possible exception of complex New Claims, where the refresh problem can slow things down, any other slowing effect is attributable entirely to the fact that the new system asks for more information than the previous one (e.g. School name, Nationality).

Q6. I am already handling a broader range of claims/enquiries than before



Qualifications or comments made by individual respondents:

Agree strongly. I have started to do some 16+, which I had not done previously.

Agree strongly. I previously hadn't done claims processing, now I have been helping clear the backlog.

Agree strongly. I have now done Additional Child, 16+ and Client Maintenance.

Agree strongly. When I was in CB I just did client maintenance. Now I have done all types.

Neither agree nor disagree. Only because I have done all types of claims/enquiries before.

Neither agree nor disagree. I had done all types previously.

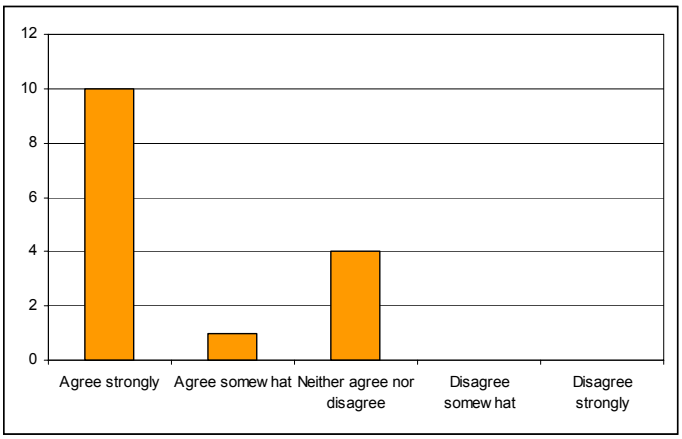
Disagree somewhat. I have added a small amount of Client Maintenance.

Disagree strongly - currently working on the same as before.

Disagree strongly - I am currently doing the same type of work as before.

Commentary: This just seems to vary by individual. Some have already learned new types of work as part of the changeover, some not yet.

Q7. I hope to be able to handle a broader range of claims/enquiries in future



Qualifications or comments made by individual respondents:

Agree strongly. I'd like to get the opportunity to try EU and ECU cases.

Agree strongly. I'd like to be able to do all types.

Agree strongly. I would like to be able to do General Payments.

Agree somewhat. Not very likely given my role (in training).

Neither agree nor disagree. I have done all types.

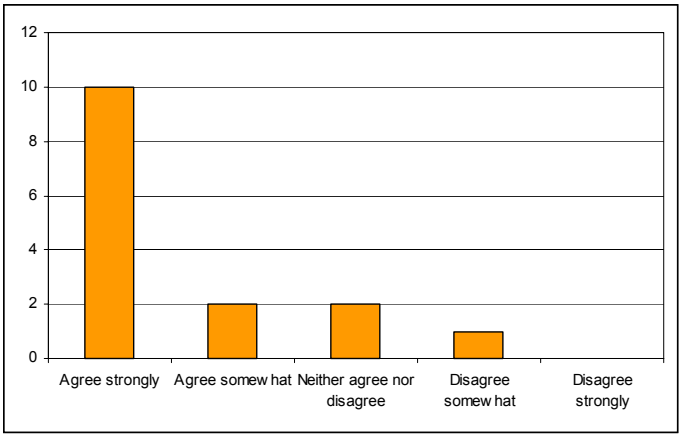
Neither agree nor disagree. Only because I have done all types of claims/enquiries before.

Neither agree nor disagree. N/A. Having done user acceptance testing, we have covered all scenarios.

Neither agree nor disagree. I had done all types previously.

Commentary: Very positive attitude overall in regard to the forthcoming multi-skilling. Those who answered 2 or 3 have either handled all types in the past, or currently function outside the claim teams.

Q8. The system enables me to manage my personal workload in a more effective manner



Qualifications or comments made by individual respondents:

Agree strongly. Because I have total control over getting the task completed.

Agree strongly. Because I am able to complete most tasks [in one pass].

Agree strongly. 'Once and done'.

Agree strongly. I can start and finish a claim without leaving my desk.

Agree strongly. Do all aspects of the task in one go.

Agree strongly. 'Once and done'.

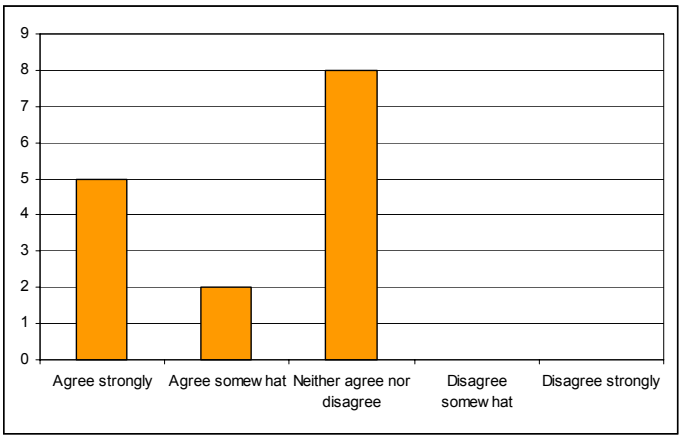
Agree strongly. Due to use of Review Dates.

Agree strongly. Because we are in teams.

Neither agree nor disagree. As a supervisor my principal use of the system is for online audit.

Commentary: Clearly the 'once and done' concept is very popular.

Q9. The system enables our team to manage our workload more effectively



Qualifications or comments made by individual respondents:

Agree strongly. Because it is easier to move work between individuals.

Agree strongly. Because if one of my forms includes something I can't handle I can hand it straight over to someone in my team.

Agree strongly. Partly from sharing tips about the system. Also we are able to share a single claim e.g. if a child is moving home.

Neither agree nor disagree - there is not yet a strong interaction between the members of the team. [That will change with multi-skilling]

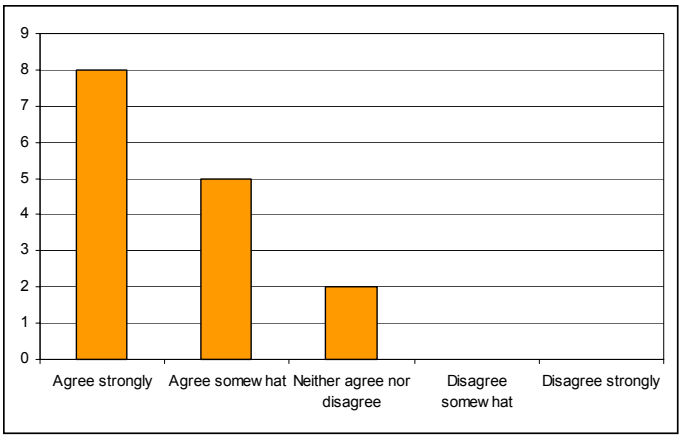
Neither agree nor disagree I am not working in one of the operational teams.

Neither agree nor disagree (Data clean-up)

Neither agree nor disagree. Had the teams been organised by speciality it would. Currently it is hard to find out who owns a particular case.

Commentary: General feeling was that this question was premature as the teams are in transition. When multi-skilling has happened, this will be a more interesting question.

Q10. The new system permits me to better deal with the needs of individual customers



Qualifications or comments made by individual respondents:

Agree strongly. The only restrictions are now not with the system, but with policy - for example, split payments.

Agree strongly. I can change a whole family over to a new form of payment without having to get back all the books.

Agree strongly. The service is much faster.

Agree strongly. Because you don't have to 'beg' someone else to help you sort out a problem.

Agree strongly. Because you are completing a claim.

Agree strongly. We are delivering a better service. Letters now come back to the individual dealing with the case not to someone else.

Agree strongly. Principally by capturing history of actions. Previously we relied heavily on remarks [field].

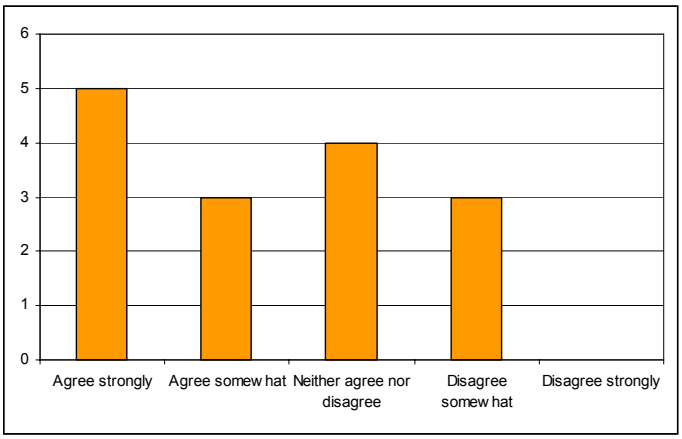
Agree somewhat. E.g. issuing books for twins and triplets.

Agree somewhat. Because it is much quicker.

Agree somewhat e.g. foster cases, where we can see the history of the child more clearly. 3. Neither agree nor disagree. So far we have been concentrating on eliminating the backlog. But in future I could see how we may be able to use the system to give better individual service.

Commentary: Most people related to this question in terms of being able to complete a claim in one sitting, and/or reduction of the backlog. Some also saw it in the broader sense of being able to meet customer's individual needs.

Q11. The new system makes it easier for me to make more ad hoc checks in relation to a claim



Qualifications or comments made by individual respondents:

Agree strongly. In particular, looking at the history of a child [in General Payments].

Agree strongly e.g. checking that a payment went through.

Agree somewhat. Especially checking relationships.

Agree somewhat. Because of the implementation of Review date.

Agree somewhat. Because the layout of the object [tree] acts as a visual prompt to check things.

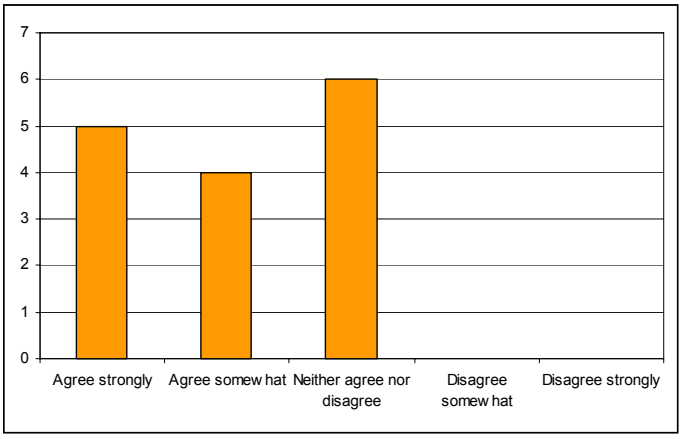
Neither agree nor disagree. I don't have any basis of comparison for this.

Neither agree nor disagree. The level of checking currently being done is only that which is specified in the departmental procedure. When the backlog is cleared and there is more time, I would expect that people will take the opportunity to do more in-depth checking on individual cases.

Neither agree nor disagree. Only thing is we now (on 16+) check that the course if properly qualified.

Commentary: No conclusive agreement on this. A lot of checking is done, but that is part of the process and/or culture. Some felt that when the backlog was cleared the system would permit them to do more in-depth checking and reviewing.

Q12. The system makes me feel more empowered as an individual



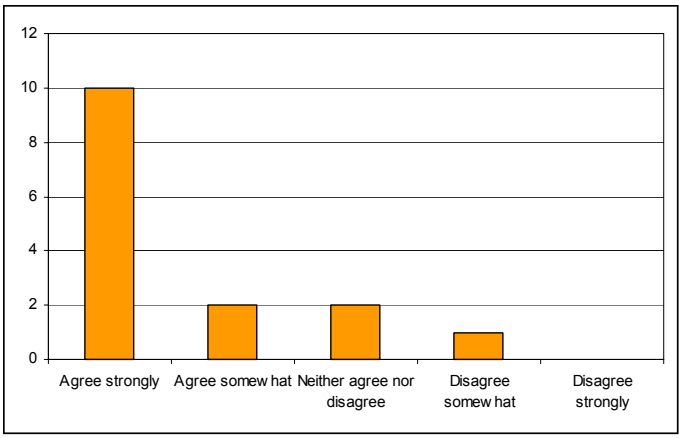
Qualifications or comments made by individual respondents:

Agree strongly. I know I don't have to rely on someone else to do their bit.

Agree somewhat . Because you can see that you've awarded the claim and put it into payment. You're not cleaning up other people's mess!

Commentary: Nine respondents agreed with this. Given the strong positive answers to certain previous questions it is perhaps surprising that this figure wasn't higher. The six who answered 'Neither agree nor disagree' seemed quite puzzled at the phrasing of the question. (The interviewer did not offer any further explanation on this question because to do so would be to lead the respondent.) Possibly this is just a terminology issue.

Q13. I value the flexibility that the system provides in choosing how to undertake a task



Qualifications or comments made by individual respondents:

Agree strongly. I can choose the way of working that I am most comfortable with.

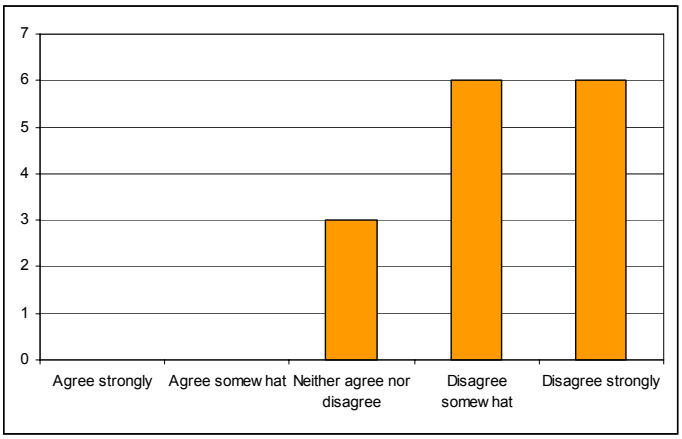
Agree strongly. My doing it differently no longer means that I am doing it wrong.

Agree strongly. You can make a mistake, but then easily go back and fix it.

Disagree somewhat. Doing 16+ extensions there isn't much choice. If I am changing a payment method at the same time I get some choice.

Commentary: Strong endorsement of the 'modelessness' of the system design.

Q14. I would prefer the system to guide me through the steps of a task



Qualifications or comments made by individual respondents:

Neither agree nor disagree. The system does effectively guide me through the task because each object offers me all the actions open to me when I right-click on it.

Disagree somewhat. Although I would occasionally value some prompts.

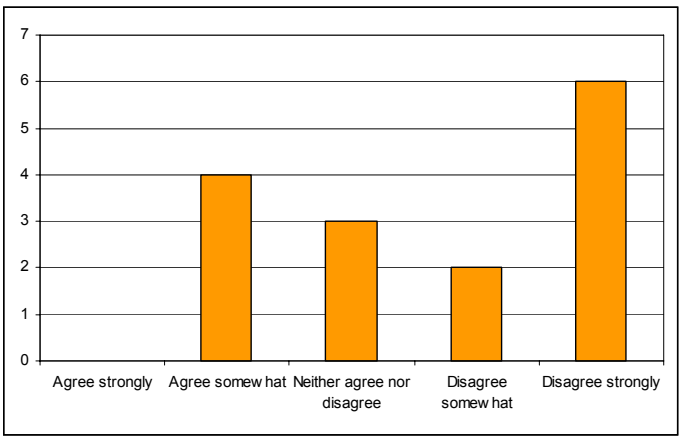
Disagree somewhat. Maybe at first, but not now.

Disagree strongly. Happy with it as it is.

Disagree strongly - that would only slow you down.

Commentary: This question was to test the reverse hypothesis from the previous question. The answers mirror well. This strongly refutes the view stated by many critics (outside the DSFA) of the Expressive Systems concept that 'staff want the system to guide them through a task'.

Q15. I would like to see more kinds of 'help' or look-up information on the system



Qualifications or comments made by individual respondents:

Agree somewhat. I don't like it when the system tells me I can't do something after I've already entered all the information. For example: creating a payment on a Scheme that has already been stopped. [i.e. the system ought not to let the user attempt to create a payment on a Scheme that has been stopped].

Agree somewhat. The School filter could be much better. As it is I quite often have to go to the spreadsheet [containing a full list of schools].

Agree somewhat. I'd like to see more [context-sensitive] help such as 'Why this date is not valid for this field'.

Neither agree nor disagree. As a beginner, yes, but not now

Neither agree nor disagree. The system has all the help and information that I need.

Disagree somewhat. Every thing you need is there.

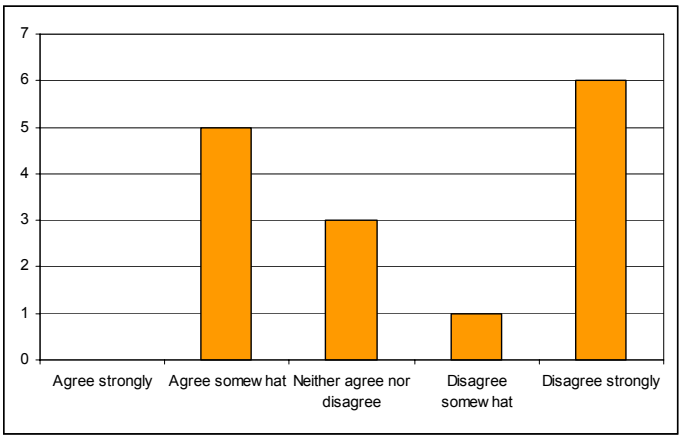
Disagree strongly. It's all there.

Disagree strongly. I find it 'grand'.

Disagree strongly. Happy enough with what there is.

Commentary: The various forms of helper information provided are sufficient for most people. However, many of them could be usefully brought inside the EOA instead of being provided as spreadsheets or documents on the Windows desktop.

Q16. There are certain kinds of errors that I find myself making repeatedly



Qualifications or comments made by individual respondents:

Agree somewhat. Mostly forgetting to complete new fields, such as Residency, that have been introduced with this system.

PaymentMethod for the Customer but then not associating it specifically with the Scheme.

Agree somewhat. Example is failing to associate a new PaymentMethod with the Scheme. However, the system doesn't actually let you do very many things wrong.

Agree somewhat. The system let's you cancel out a screen without saving it.

Agree somewhat. Creating a new payment method and then not associating it with the CB Scheme. Easy to accidentally end up paying for a child even through a gap year i.e system does not warn me that the Scheme is on 'Stop G'.

Neither agree nor disagree. It can be easy to Award each of the children in a claim but forget to award the claim as a whole.

Neither agree nor disagree. It is easy to apply for a PPSN for a child without checking that one has already been applied for or assigned. This can result in creating two PPSNs. It is even possible that the system can conclude that they are twins (with the same name!)

Neither agree nor disagree. One example though [cited by many people] was creating a new

Disagree strongly - first week or so, maybe.

Disagree strongly. At the beginning, maybe, but not now.

Disagree strongly. At the beginning only.

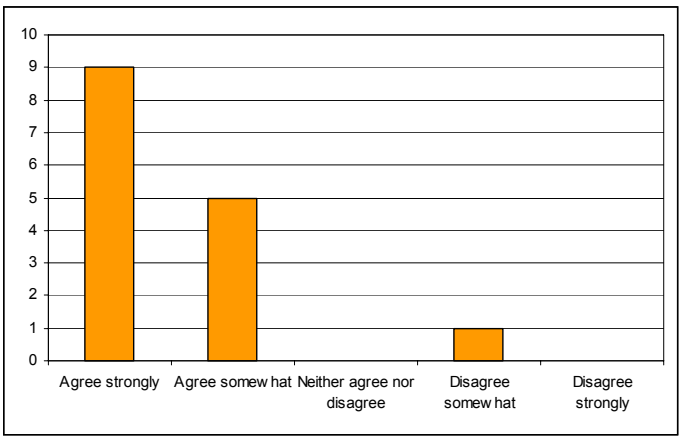
Commentary: Almost everyone stated that their response to this question referred to their early experiences i.e. they are no longer making those same mistakes. Also the mistakes were not made frequently: but those cited were made on more than one occasion.

Some of these arise from changes in the information now being gathered.

Some clearly suggest the need for simple additional business rules of checks. (e.g. not being able to create a Payment on a Scheme that has been stopped.)

The example of failing to associate a new PaymentMethod with the Scheme is a more interesting one (from the point of view of the EOA designers) because this one is caused by the fact that the system I designed for more general usage than just CB. (i.e. you could enter a new PaymentMethod and then assign it to specific Schemes). This one needs further thought on the part of both system designers and change managers / trainers.

Q17. I like the look and feel of the user interface on this system



Qualifications or comments made by individual respondents:

Agree strongly. As both a user and a trainer.

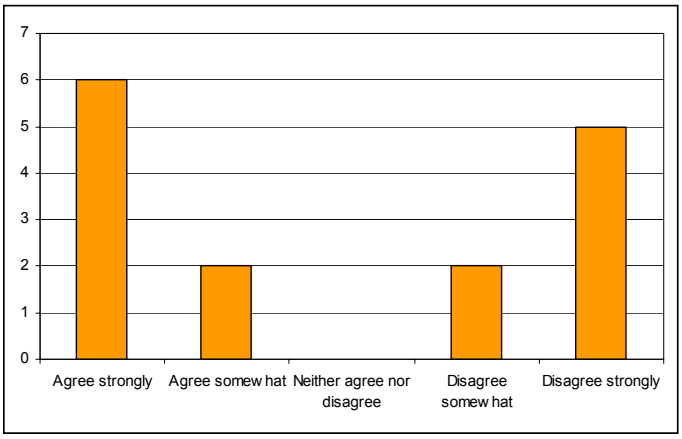
Agree somewhat. Only real dislike is the 'refresh'.

Agree somewhat. I don't like the way that the windows contract again when you save them. [the 'refresh' problem].

Disagree somewhat. Only because I don't like the small text, and especially the blue text.

Commentary: Positive endorsement of the overall design. Only dissent is due to difficulty of reading the small text. This could be solved with a larger screen for such individuals, or possibly reducing the screen resolution.

Q18. I often copy objects onto my desktop



Qualifications or comments made by individual respondents:

Agree strongly. I don't tend to copy Customers as icons onto the desktop, but I do tend to keep Customer windows open between cases if I'm expecting the customer to call back. I keep standard [communication] forms on the desktop [so I can copy them into a communication]. (Although it is annoying that you can't open up the standard forms as such). (Also, I would like to see the standard forms copy more information in from the Scheme e.g. the name of the child).

Agree strongly. All the search options. Not individual customers.

Agree strongly. Templates. However, when the templates are updated (centrally) the shortcuts on the desktop no longer work [because the original objects are deleted and then replaced].

Agree strongly. Customers I know I'll have to come back to next morning. But keeping too many is apparently a systems problem.

Agree somewhat. Mostly the [Communication] templates.

Agree somewhat. But only during the treatment of one case. Then I put the objects away again.

Disagree somewhat. Occasionally I do.

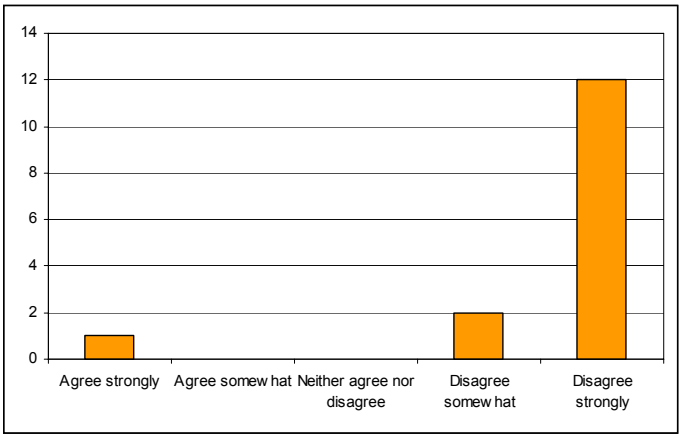
Disagree somewhat. I copy them temporarily, but always clean up after each case.

Disagree strongly. Although I do keep my Officer object on the desktop so I can easily look up which of my actions are being audited.

Disagree strongly. CB Scheme only.

Commentary: The polarisation of answers here is good. Saving objects on the desktop was an optional feature. Clearly some people use it and some don't. Perhaps more will as the further organisational changes are introduced.

Q19. I often end up with too many windows open



Qualifications or comments made by individual respondents:

Disagree somewhat - at the beginning, maybe, but not now.

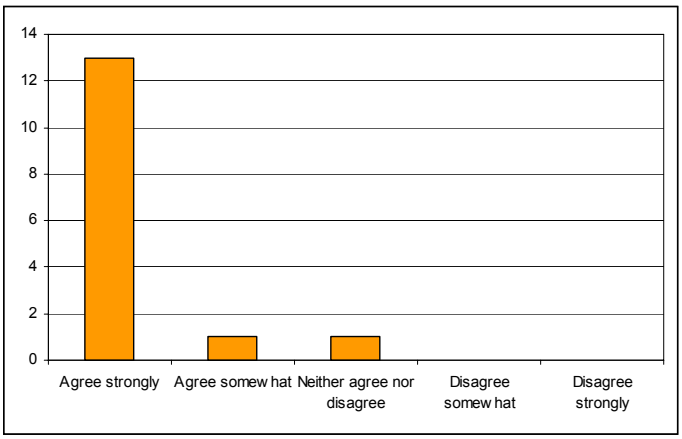
Disagree somewhat - this slows the system down. Anyway you are limited to 10 open windows.

Disagree strongly - I know all the shortcuts [e.g. for tabbing between windows].

Disagree strongly - it has been suggested that leaving too many windows open is one cause of slowing the system down.

Commentary: With one self-confessed exception, who admits to having to close down 'dozens' of windows at the end of the day, most people choose to close up all the windows when they finish a case. The point of this question was to test the prediction by some critics that window management would be a big problem. Clearly it isn't. The rumour that having too many windows open is the cause of systems performance problems should be either clearly verified or quashed.

Q20. I like the use of 'drag and drop' to initiate actions



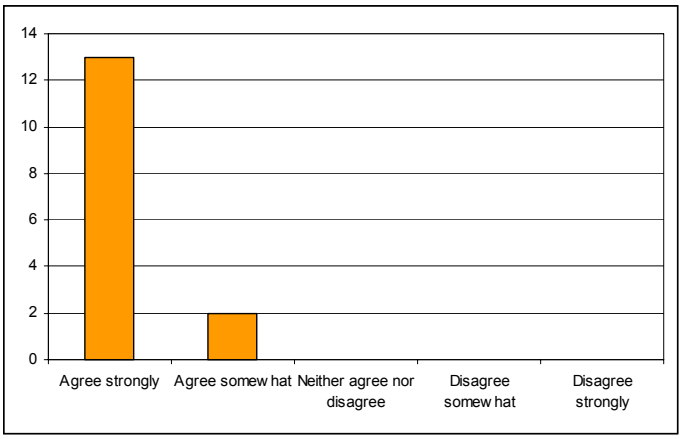
Qualifications or comments made by individual respondents:

Agree strongly - sometimes the system seems to drop the object 'accidentally'. But drag and drop is much better than the alternative.

Neither agree nor disagree - my own role mostly involves look-ups rather than actions through the system.

Commentary: Very strong endorsement for this technique, even from those who had never used a mouse before encountering this system.

Q21. (Aside from teething problems) I am generally satisfied with the new system

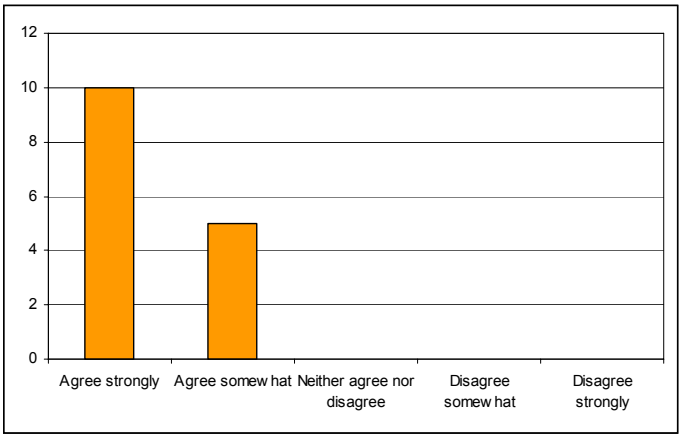


Qualifications or comments made by individual respondents:

Agree strongly - as both a user and a trainer.

Commentary: That's a very good result for any new business system!

Q22. The new system contributes positively to my job satisfaction



Qualifications or comments made by individual respondents:

Agree strongly - as a user. Somewhat as a trainer.

Agree strongly - I'm a happy camper!

Agree strongly - both on UAT and as a user

Agree strongly. - because of getting up to date.

Agree strongly - principally because of the 'once and done' concept and the new organisational structure.

Agree somewhat - to the extent that I am a user of the system.

Agree somewhat - when it is working properly.

Commentary: None needed.

APPENDIX VI. A BRIEF DESCRIPTION OF THE *NAKED OBJECTS* FRAMEWORK

The framework itself (specifically Version 1.0 of the framework) is described in detail in [86]. What follows is a brief description only.

To develop an application using Naked Objects, all that the developer writes are the business objects that model the domain. Each class of business object (for example, Customer, Product, and Order) must be written as a Java class, and must implement an interface called NakedObject. The simplest way to do this is to declare each business class to be a sub-class of the AbstractNakedObject class provided with the framework. The programmer must write the necessary code to specify each business object's attributes, associations with other business objects, and business behaviours. The code must follow a few simple conventions, several of which are similar to JavaBean conventions. In general the business objects are coded in the same way that behaviourally-complete business objects would be written for the business domain model layer of any multi-layered system.

When the set of business classes is compiled and run, the framework's generic 'viewing mechanism' provides the user with a view of the business objects in a form like that shown in figure 5.1. Individual a business object instance will show up as an icon (indicating which class it belongs to), and a unique title (specified by the programmer and usually derived from one or more of the object's primitive attributes, such as the name, date and reference number). Any of these icons can be double-clicked to open a more detailed view of the object's attributes and its associations with other objects, which show up as icons in their own right.

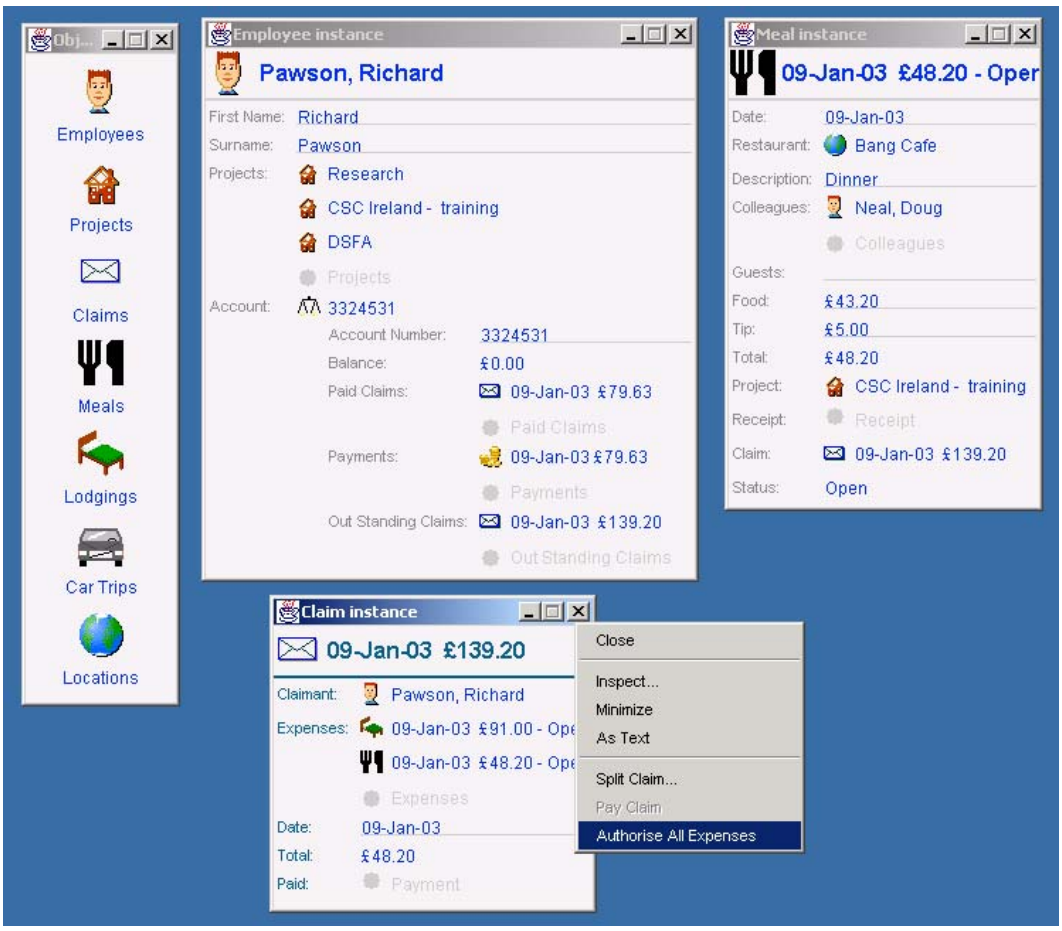


Figure 5.1. An early-stage prototype for an expenses processing system using the *Naked Objects* framework.

Right-clicking on any object will produce a pop-up menu of actions (i.e. instance methods) that users can invoke on that object. Right-click on a Customer object and you might be able to 'communicate' with that customer (via one of the customer's communication addresses) or perhaps 'assess the value' of the customer to the business based on past orders.

As well as via pop-up menu actions, behaviours can also be initiated by user drag-and-drop actions. The user can drag an object onto another object, which will trigger a pre-defined operation involving the two objects; or drag an object into a specific field within another object, usually to specify an association between the two. If the user attempts to drop the wrong type of object, the drop-zone will flash red and the drop will not work. Similarly a menu action may be greyed-out if the action is not permissible.

In addition to the individual instances, and collections of instances, the system presents the user with a direct representation of the classes themselves - shown in Figure 5.1 as the Classes window. This is where the user goes to invoke behaviours that don't belong to a single instance, such as: to create a new instance of that class, to find a specific instance, or to list the instances of that class that match some specified criteria.

The framework relies heavily on the Java capability of ‘reflection’ (a capability known in some other OO languages as ‘introspection’). At runtime the viewing mechanism can ask any business object to list the methods that it can respond to. If this list includes any method that is prefixed with ‘action’ (such as actionNewBooking()) then this method, stripped of the prefix and reformatted, will automatically be added to the list of actions made available to the user via the pop-up menu for that object (‘New Booking. . .’) Similarly, if the reflection reveals a method prefixed with ‘get’, such as a getCustomer() method that returns a Customer object, then this will be rendered as a field containing a Customer object. If there is a corresponding ‘set method (e.g. setCustomer(Customer c)) method then the user will be able to drop any Customer object onto that field.

APPENDIX VII. SURVEY OF PROJECT PARTICIPANTS AT SAFEWAY

Description

Safeway Stores conducted the Exploration phase of two projects using the Naked Objects approach during 2001:

1. The Deal Nominations (DN) project. This was conducted as a learning exercise only. There was no commitment at the time that the system would be implemented, and it has not been - though to the evident disappointment of the business representatives.

2. The Cluster-Based Pricing (CPB) project. After the Exploration phase the team started to develop a full-scale implementation using Naked Objects. Unfortunately, it was not possible to deploy a Java-based system on the available production platform and the system was therefore redeveloped in the CICS-Cobol environment, but using as much as possible of the design work from the Exploration period.

These projects are described more fully in Chapter 6.

In February 2003, the author (who had acted as a part-time consultant and coach on both projects) returned to Safeway to interview 10 individuals who had been involved in either or both of these Explorations. The interview group was split approximately 50:50 between developers and business user representatives. (A couple of the individuals blended both roles). This population of 10 represented almost all those who had had a significant involvement in these projects.

The questionnaire is shown overleaf. After the background questions, all questions take the form of a proposition, to which the interviewee was asked to give one of five responses:

1. Agree strongly 2. Agree somewhat 3. Neither agree nor disagree 4. Disagree somewhat 5. Disagree strongly

Note that not all of the questions were relevant to all participants. In particular some of the later questions are oriented more towards those in a developer role rather than a business role. All participants were given the option to answer all questions if they wished, however. Where the question was deemed by the interviewee not to be relevant a '—' is shown in the table.

Questionnaire

Name:

Normal business role:

Naked Objects project(s): (Cluster-based pricing, Deal Nominations, or both)

Role on project(s):

Degree of involvement:

Previous experience of business systems analysis/design

Previous experience of object-oriented approaches to systems design (if any):

Q1. Using Naked Objects greatly facilitated communication between developers and business representatives, during the discussion of requirements

Q2. Using Naked Objects we were able to prototype the underlying object model at least as rapidly as we could normally have prototyped screenshots alone.

Q3. I found it easy to get into thinking about the business system purely in terms of behaviourally-complete business objects

Q4. The Naked objects approach encouraged us to explore alternative ways to represent or model the business domain

Q5. The period of Exploration revealed specific user requirements that would probably not have been identified using a paper-based approach to requirements specification (or even conventional screen-based prototyping).

Q6. Exploring the business domain using the 'naked' objects, lead to specific business insights or possible business approaches that had not previously been seen

Q7. I did not find it difficult to adopt the object-oriented concepts (such as class, instance, method) used during the exploration. (Question asked of those with no prior knowledge of these concepts - principally the business roles).

Q8. The resulting prototype provided the user with a strong sense of being a problem-solver

Q9. Treating the user as a problem-solver would be an important requirement for the delivered system

Q10. The problem solving style of user interaction made a valuable contribution during the Exploration activity.

Q11. The application we tackled would not have fitted well into a process-oriented approach to business systems analysis and design

Q12. The resulting object model would probably be able to accommodate a broad range of future business changes

Q13. Overall I was satisfied with the activities and conduct of the Exploration phase

Q14. Overall I was satisfied with the results (deliverables) arising from the Exploration phase

Q15. I found it easy to learn the Naked Objects framework and style of development. (Question asked only of developers).

Q16 The restrictions or limitations imposed by the Naked Objects framework did not negatively impact what we wanted to achieve.

Q17. I found that being able to prototype in front of users to be an effective way of working. (Question asked only of those individuals, both developers and business roles, who had direct experience of this live prototyping).

Q18. I found it easy to translate user requests into changes on the object model. (Question asked of developers only)

Q19. The output of the Exploration phase provided a clearer basis for proceeding onto full-scale design and implementation than a conventional requirements specification

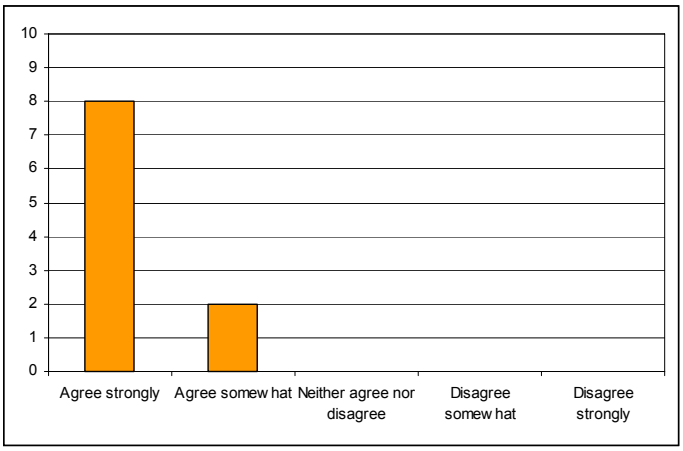
Q20. Using the Naked Objects approach significantly improved my understanding of object-oriented techniques in general.

Summary of responses

Question	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Individual responses																				
Respondent 1	1	1	2	1	1	2	--	1	1	3	2	1	2	1	1	2	1	2	1	1
Respondent 2	1	1	1	2	1	2	--	3	1	1	3	1	2	1	1	1	1	2	2	1
Respondent 3	1	1	2	1	2	1	1	4	4	1	5	1	2	1	--	--	--	--	--	--
Respondent 4	2	1	1	1	2	1	1	2	1	1	1	1	2	2	--	--	--	--	--	--
Respondent 5	1	1	2	1	2	1	1	2	2	1	5	1	1	1	--	--	--	--	--	--
Respondent 6	1	1	1	2	3	2	--	3	5	1	3	2	1	2	1	2	1	1	3	1
Respondent 7	2	1	2	4	2	4	1	1	2	1	4	2	2	1	--	2	1	3	2	2
Respondent 8	1	1	2	2	2	2	2	1	1	2	1	1	1	1	--	1	1	--	3	1
Respondent 9	1	1	2	2	2	2	--	2	1	2	4	2	1	1	2	2	--	2	2	1
Respondent 10	1	1	2	1	1	1	2	3	3	3	2	1	1	1	--	2	1	--	--	--
Total	10	10	10	10	10	10	6	10	10	10	10	10	10	10	4	7	6	5	6	6
Frequency																				
1. Agr. strongly	8	10	3	5	3	4	4	3	5	6	2	7	5	8	3	2	6	1	1	5
2. Agr. somewhat	2	0	7	4	6	5	2	3	2	2	2	3	5	2	1	5	0	3	3	1
3. Neither agr/dis	0	0	0	0	1	0	0	3	1	2	2	0	0	0	0	0	0	1	2	0
4. Dis. somewhat	0	0	0	1	0	1	0	1	1	0	2	0	0	0	0	0	0	0	0	0
5. Dis. strongly	0	0	0	0	0	0	0	0	1	0	2	0	0	0	0	0	0	0	0	0
Median response	1	1	2	1.5	2	2	1	2	1.5	1	3	1	1.5	1	1	2	1	2	2	1

Charts and commentary

Q1. Using Naked Objects greatly facilitated communication between developers and business representatives, during the discussion of requirements

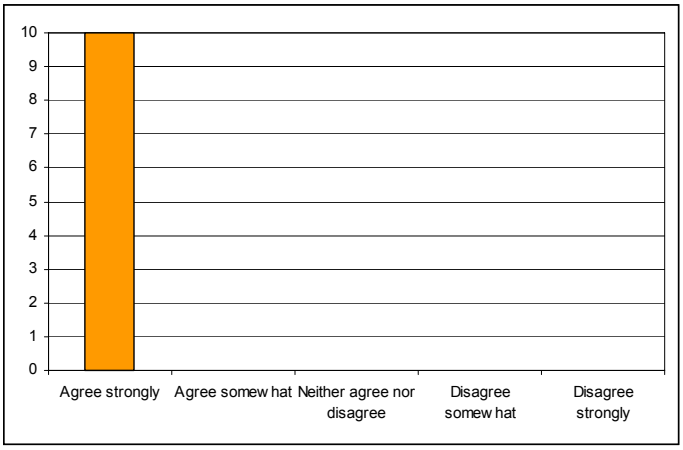


Qualifications or comments made by individual respondents:

Agree somewhat. It was the way it was deployed (e.g. live prototyping) rather than the tool or technique itself. However, Naked Objects did make that approach possible.

Commentary: A pretty strong consensus on this one, from both the developers and business representatives.

Q2. Using Naked Objects we were able to prototype the underlying object model at least as rapidly as we could normally have prototyped screenshots alone.



Qualifications or comments made by individual respondents:

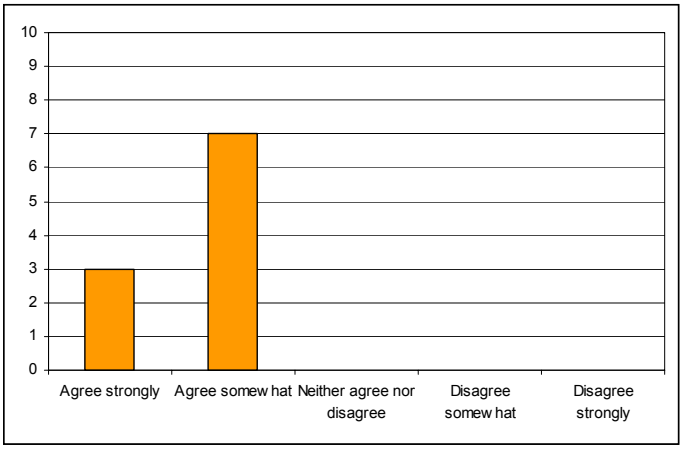
Agree strongly. I had done conventional screenshot prototyping on a couple of projects.

Agree strongly. Naked Objects prototyping was definitely faster than conventional screen-based prototyping

Agree strongly. (though I had no previous personal experience of rapid prototyping)

Agree strongly. We have some talented individuals within the organization who are good at prototyping generally. But the Naked Objects prototyping was faster than anything I have seen.

Q3. I found it easy to get into thinking about the business system purely in terms of behaviourally-complete business objects



Qualifications or comments made by individual respondents:

Agree strongly. In the previous attempt at analyzing the Deal Nominations requirements we had really struggled to understand the business model. Thinking in terms of business objects made it much clearer.

Agree somewhat. For one whole day I found this quite difficult. When I got used to the idea of stripping an application to its bare essentials it started to become a lot easier. Within a short period it became 'blindingly obvious'.

Agree somewhat. We depended on the facilitator (RP) for this. Probably we could have specified our requirements in our own terms and then had them translated offline into object responsibilities.

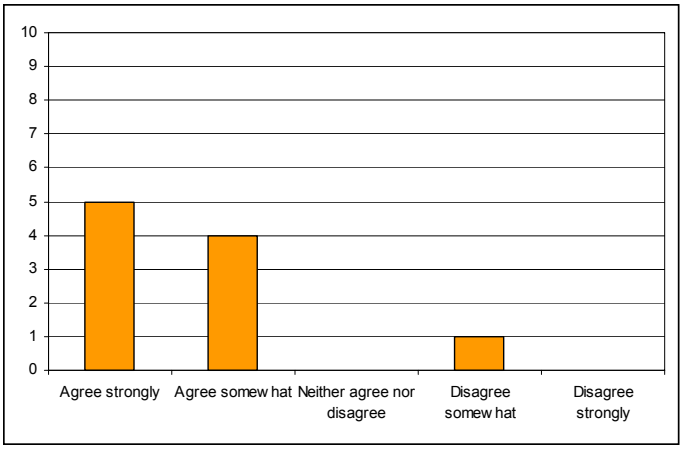
Agree somewhat. There was some confusion and a lot of questions. But most people caught on quite quickly.

Agree somewhat. The approach took some getting into. But with Naked Objects ultimately we could understand the system much better.

Agree somewhat. I was very used to non-OO styles of analysis and found the transition difficult to begin with.

Commentary: Again, strong consensus here. The 'agree somewhat' ratings mostly reflect the fact that there was a steep learning curve.

Q4. The Naked objects approach encouraged us to explore alternative ways to represent or model the business domain



Qualifications or comments made by individual respondents:

Agree strongly. This approach freed up our thinking. It forced us to focus on the end result rather than the detail. There was no barrier to involvement: users could contribute business ideas without the need for technical understanding. Normally, users can't participate unless they understand the technical constraints of the tools/systems involved.

Agree strongly. Example: different ways to structure pricing. Previously we had never questioned the established approach. This approach really forced us to think about the high level process. In previous [CICS-based] approaches we just focus on the screens, not on the broader implications.

Agree strongly. It made us think 'on our feet' about the nature of the business.

Agree strongly. Our normal approach is documentation heavy and this encourages ideas to be fixed early on.

Agree somewhat. Much of the discussion about object names was actually about alternative models.

Agree somewhat. We had explored some alternatives prior to start of Naked Objects project. However the Naked Objects approach greatly facilitated this exploration.

Agree somewhat. The only limitation was the degree to which the business representatives were willing to explore alternatives.

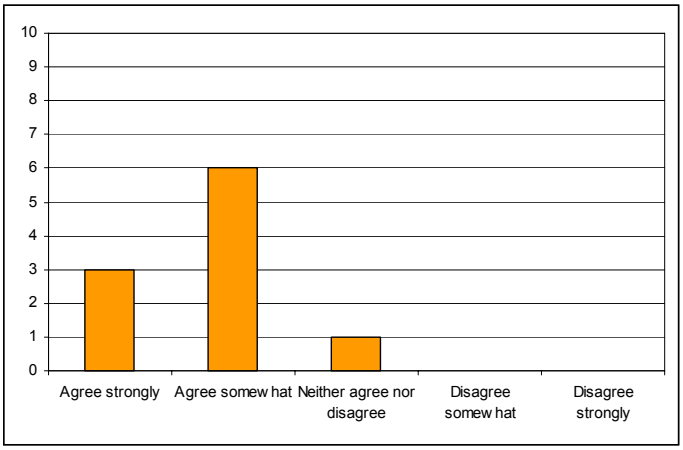
Agree somewhat, but the facilitator (RP) had strong views about which way we should model the business domain.

Disagree somewhat. Certainly Naked Objects allowed alternatives to be explored, but it did not necessarily encourage it.

Commentary: Strong, but not universal support for this message. Note that there was no correlation between the responses and the role of the individual (developer or business).

.

Q5. The period of Exploration revealed specific user requirements that would probably not have been identified using a paper-based approach to requirements specification (or even conventional screen-based prototyping).



Qualifications or comments made by individual respondents:

Agree strongly. The images on the Offering, and the idea of being able to get to all the product details through the Product object.

Agree strongly. This could be seen by comparing the Deal Nominations prototype against the list of user requirements that had resulted from a previous attempt.

Agree strongly. Specific example (on DN): displaying the set of offerings as a collection of product images instead of just a text list

Agree somewhat. There was something (on DN) about being able to find alternative suppliers.

Agree somewhat. We probably would have got all the requirements using a paper-based approach but some of them wouldn't have surfaced until much later in the project. The Naked Objects approach brought them out very early.

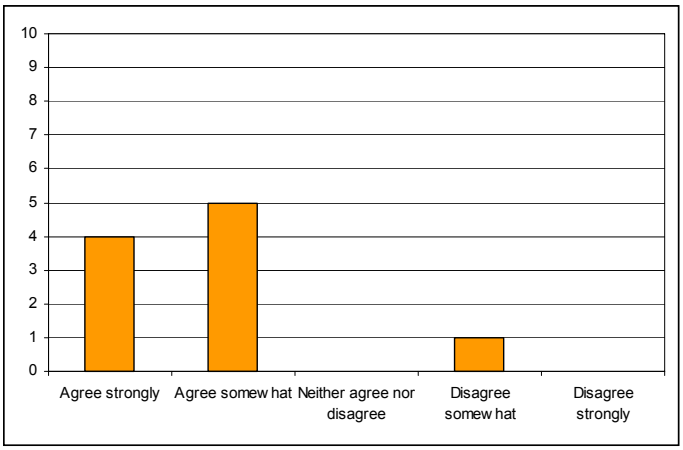
Agree somewhat. There were several cases where the Naked Objects prototype gave us something for nothing [i.e. functionality that was an automatic by-product of creating an object type].

Agree somewhat. One piece of evidence for this is that when we got to the delivery stage, very little extra functionality was added. Normally, at that stage the users would be asking for new things they hadn't thought of before.

Agree somewhat. We didn't have to think about things like where something should appear and on which screen.

Commentary: Those who agreed strongly could all recall specific examples.

Q6. Exploring the business domain using the 'naked' objects, lead to specific business insights or possible business approaches that had not previously been seen



Qualifications or comments made by individual respondents:

Agree strongly. The prototype suggested the possibility of clustering at multiple levels. The analysis went much deeper than in normal projects. During this project we presented the IT team with some tough challenges. Normally we settle for too little. On this project we were never told [by the IT team] that something couldn't be done because it didn't fit the technology.

Agree strongly. Example: the idea of [the user] exploring multiple scenarios [combinations of deals to make up an offering] and then 'making-so' the best scenario.

Agree strongly. One was the realisation that we would see right at the start of putting together a planned Offering whether we had images for the product in the database. Today, we often discover the lack of an image quite late in the day.

Agree somewhat. Example (on DN): Integrating demand forecasting with the deal planning

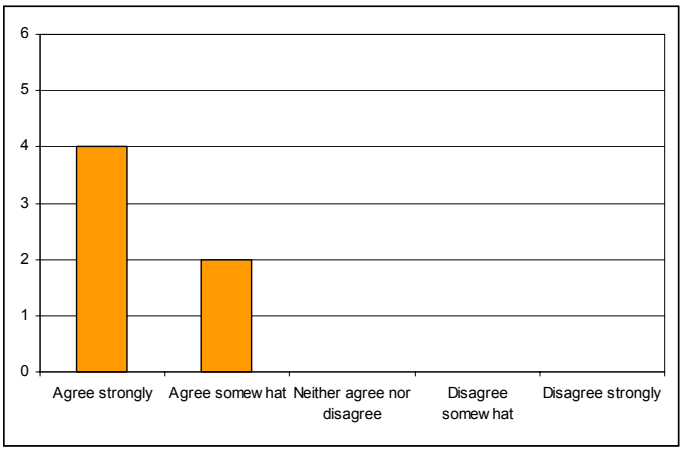
Agree somewhat. In understanding the various different components involved in building up the offering

Agree somewhat. (in Deal Nominations) The idea of modelling the Flyer on the screen.

Disagree somewhat. Most of the discussions were 'IT centric'.

Commentary: The last comment came from an IT person, not a business representative.

Q7. I did not find it difficult to adopt the object-oriented concepts (such as class, instance, method) used during the exploration



Qualifications or comments made by individual respondents:

Agree strongly. I was unconscious (in the role of a user) of having to learn any difficult new concepts.

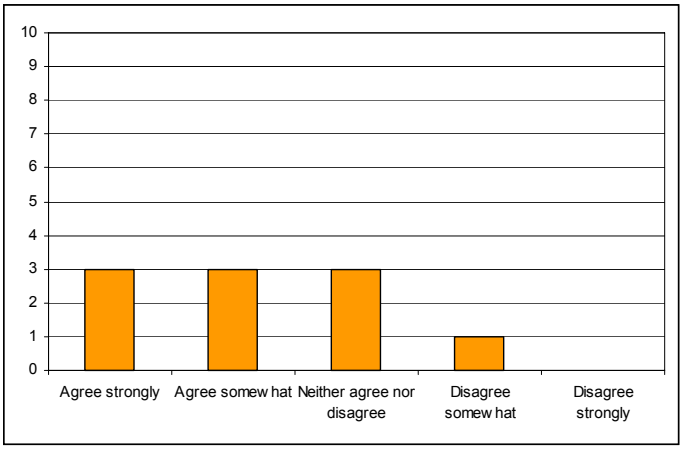
Agree strongly. I now use those concepts all the time.

Agree somewhat. On my UML course I had struggled to understand the difference between class and instance. Naked Objects made that distinction very clear.

Agree somewhat. This was quite a dramatic change from what I was used to.

Commentary: This result counters any suggestion that the Naked Objects approach is forcing users to adopt programming terminology.

Q8. The resulting prototype provided the user with a strong sense of being a problem solver

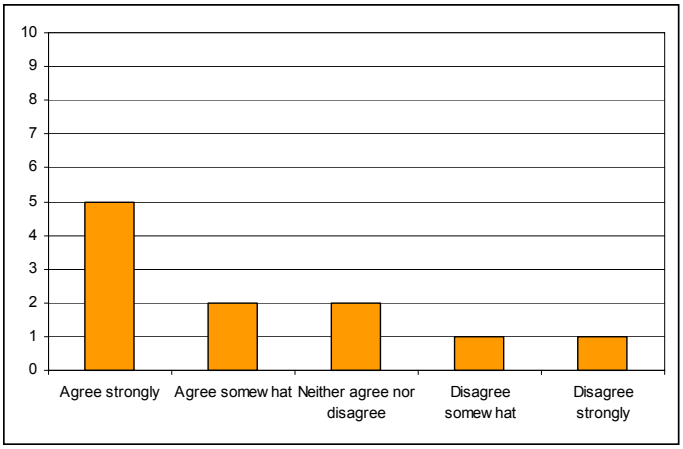


Qualifications or comments made by individual respondents:

Agree somewhat. The prototype allowed the user to choose their own way of working.

Commentary: The concept of the user as a problem solver was certainly not recognized across the board. Interestingly, this did not seem to matter in terms of the outcome. Nor was it seen as necessary to gaining the benefits from the Naked Objects approach.

Q9. Treating the user as a problem solver would be an important requirement for the delivered system



Qualifications or comments made by individual respondents:

Agree strongly - for DN. Neither agree nor disagree - for CBP

Agree strongly - for both DN and CBP

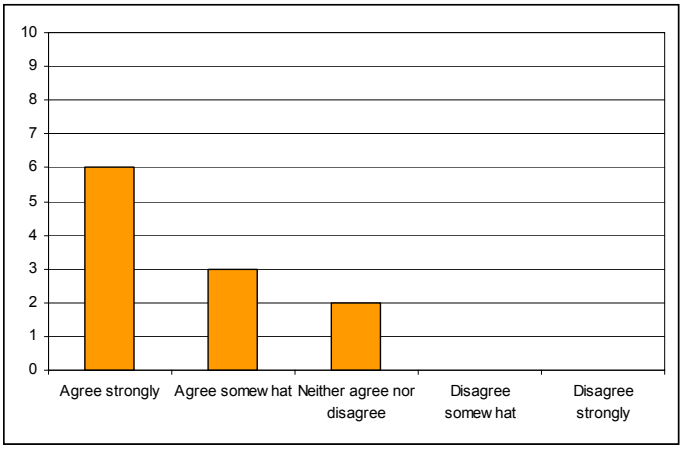
Agree somewhat. There were other ways of doing it. However the Naked Objects approach probably helped. Treating the user as a problem solver is something that we don't do well.

Agree somewhat. Perhaps a better phrase than 'problem solver' would be 'decision maker'. This type of system would help people to make good decisions.

Disagree somewhat. Most of the users would just see CBP as means to achieving a task i.e. doing price adjustments.

Commentary: As previous question.

Q10. The problem solving style of user interaction made a valuable contribution during the Exploration activity.



Qualifications or comments made by individual respondents:

Agree strongly. We all acted as problem solvers.

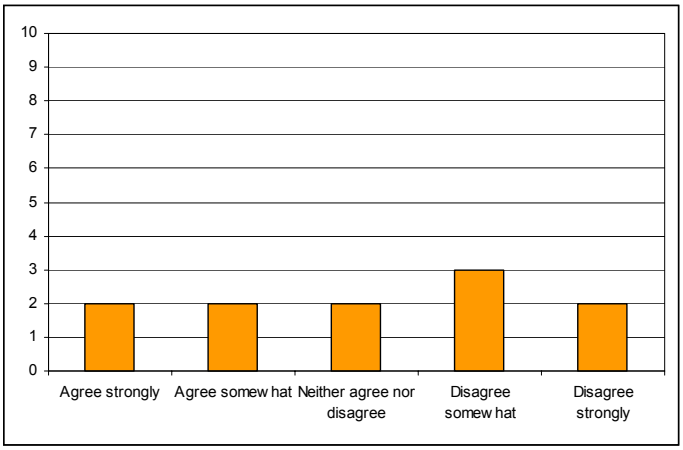
Agree strongly. This is something we don't do enough of in IT.

Agree strongly. The flip side of this is that it got me onto thinking about usage scenarios well beyond the original scope.

Agree somewhat. The free-from discussion did cause us to get stuck in a few ruts.

Commentary: There is clearly stronger recognition of the value of treating users as problem solvers for the purposes of requirements analysis, than as end-users.

Q11. The application we tackled would not have fitted well into a process-oriented approach to business systems analysis and design



Qualifications or comments made by individual respondents:

Agree strongly. The Naked Objects approach felt very different and more natural.

Agree strongly - for DN. Disagree somewhat - for CBP

Agree strongly - for DN Neither agree not disagree - for CBP

Agree somewhat. The process-oriented approach would not have worked well on the Deal Nominations system.

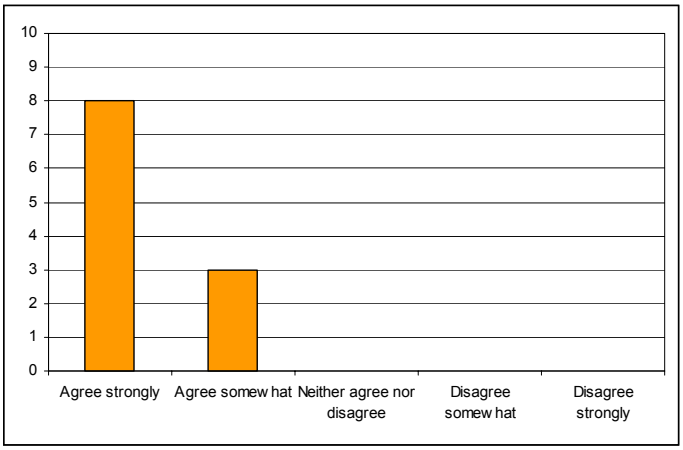
Disagree somewhat. We are getting more strongly into the workflow model (i.e. all activities carefully scripted) as an organization. This application could have been done that way, but I'm not saying that it should have been done that way.

Disagree strongly. CBP was eventually implemented in CICS. It therefore probably could have been designed that way from scratch.

Disagree strongly. The Deal Nominations activity could still be seen as part of a much large process, which could be modelled using more conventional approaches. However, the Naked Objects approach means that you aren't just restricted to process-oriented ways of thinking.

Commentary: No clear result here.

Q12. The resulting object model would probably be able to accommodate a broad range of future business changes



Qualifications or comments made by individual respondents:

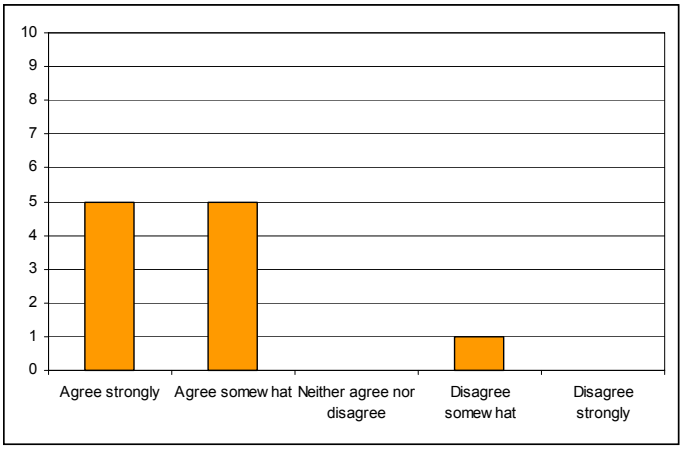
Agree strongly. We could base the entire business model on this approach. It is the ideal solution where we want to differentiate.

Agree strongly. The CICS based systems are very hard to change.

Agree strongly. It already has done. [Even though the DN system wasn't implemented, several of the ideas resulting from that exploration session have survived and been implemented in other ways.]

Commentary: All respondents recognised that this was a speculative answer. But nonetheless, there was a strong belief that the object model resulting from this approach would be more robust to future business changes than a conventionally designed system.

Q13. Overall I was satisfied with the activities and conduct of the Exploration phase



Qualifications or comments made by individual respondents:

Agree strongly. I found that Exploration immensely rewarding. This was the first fundamental intellectual challenge I had experienced for years.

Agree strongly - for DN. Disagree somewhat - for CBP. In CBP we seemed to be driven more in one direction by the facilitator, in DN it seemed to be more of a team effort.

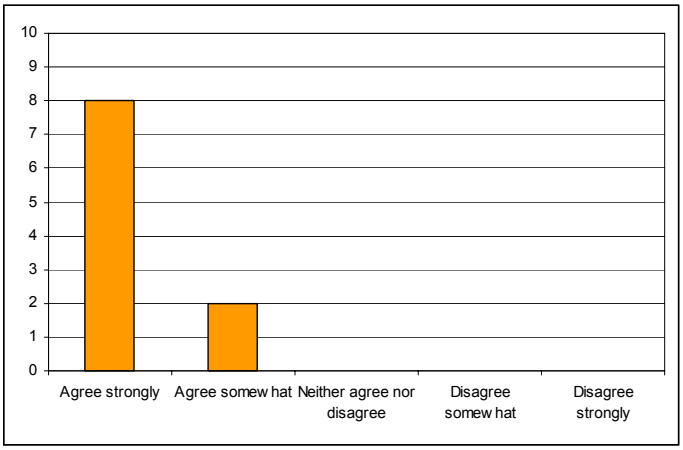
Agree somewhat for DN. Agree strongly for CBP. This was because of the added value of the test-based scenarios in CBP.

Agree somewhat. The Naked Objects approach helped to reduce the 'us and them' culture that plagues many of our projects. But I think we could have gone further.

Agree somewhat. [CBP] RP was a very strong facilitator which may make the process difficult to replicate. There seemed to be some resistance to some ideas/requirements.

Commentary: General satisfaction with the exploration phase, but a warning to the facilitator from one respondent.

Q14. Overall I was satisfied with the results (deliverables) arising from the Exploration phase



Qualifications or comments made by individual respondents:

Agree strongly. The evidence for this was the huge number of managers that heard about the project and wanted a demonstration.

Agree strongly. [CBP] The exercise was not wasted because some of the object model did make it into the CICS-based implementation. But I felt cheated that the delivered system did not adopt the [Naked Objects] design.

Agree strongly. Like Ronseal - 'it does what it says on the tin' ;-)

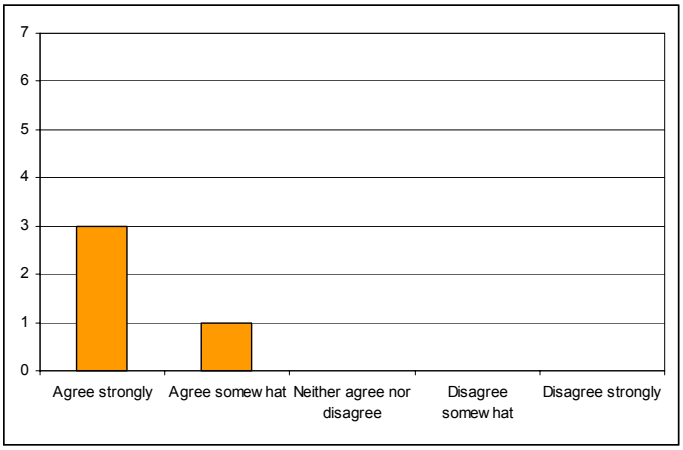
But I felt cheated that this project didn't go forward into implementation.

Agree somewhat. Some of the things that didn't end up in the CICS implementation have resulted in other projects since.

Agree somewhat. There were some issues with technical stability. [The Naked Objects framework was evolving rapidly at that point.] The prototype would not run now [due to changes both in Naked Objects and the VisualAge IDE. However, the real benefit of using Naked Objects in the exploration phase was the understanding of the business requirements, not the product itself.

Commentary: Very strong endorsement of the end value of the Naked Objects exploration phase.

Q15. I found it easy to learn the Naked Objects framework and style of development

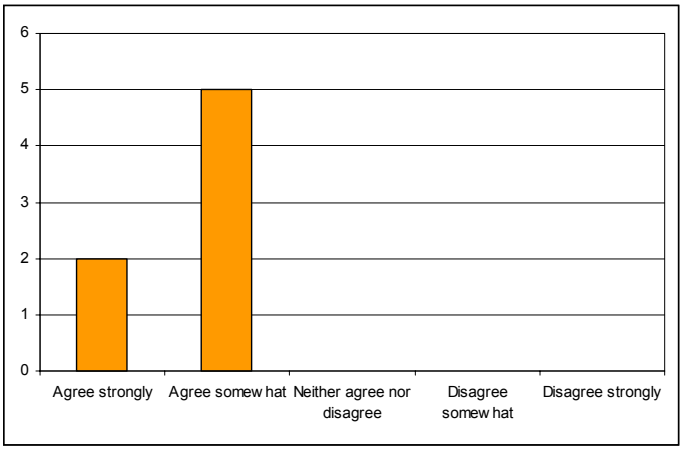


Qualifications or comments made by individual respondents:

Agree strongly. More so than any of the other Java techniques or concepts I have had to learn.

Agree somewhat. Not worrying about windows and GUI programming helped a lot. The visibility of the objects also helped. I had previously never programmed in Java (except for a very short course 1-2 years prior) nor had I developed GUIs.

Q16. The restrictions or limitations imposed by the Naked Objects framework did not negatively impact what we wanted to achieve



Qualifications or comments made by individual respondents:

Agree strongly. The Naked Objects approach stopped us from focusing on all the irrelevant things that we normally focus on in requirements analysis.

Agree strongly. (I learned the development approach, but not the framework itself)

Agree strongly. The framework was still immature at that stage. There could even have been more restrictions, which would have improved its value.

Agree somewhat. No more so than any other technology we've used.

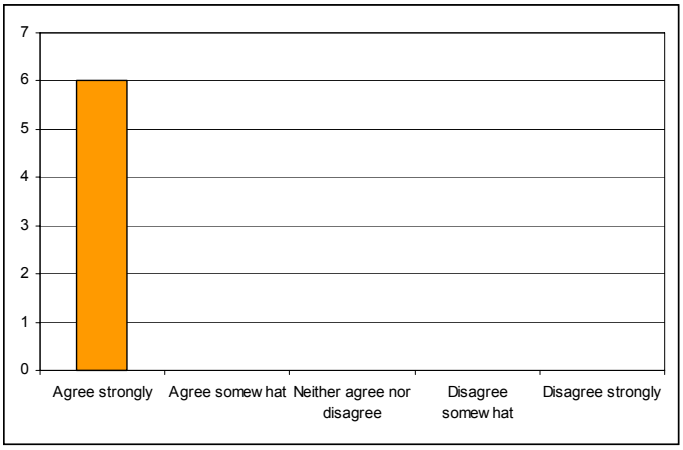
Agree somewhat. There were some debates, but I would have heard about it if the framework was proving to be a restriction.

Agree somewhat. Initially I found some of the restrictions (e.g. no dialog boxes) difficult to work with. Some of the shortcomings of the framework [N.B. interviewee recognized the difference between these shortcomings and deliberate restrictions] were also rectified during the project. I had previously never programmed in Java (except for a very short course 1-2 years prior) nor had I developed GUIs - therefore to some extent I wasn't aware of some of these restrictions.

Agree somewhat. The caveat was that since Naked Objects was still in its infancy we were able to influence the development of the framework [by Robert Matthews] and therefore to eliminate some of the limitations that might have proved to be a problem.

Commentary: This question was directed at developers and those with IT management responsibilities. Some business representatives chose to answer it.

Q17. I found that being able to prototype in front of users to be an effective way of working



Qualifications or comments made by individual respondents:

Agree strongly. We could have done even more of this.

Agree strongly. As a user-representative I found it great to be able to interact with a developer in a non-technical style of systems development.

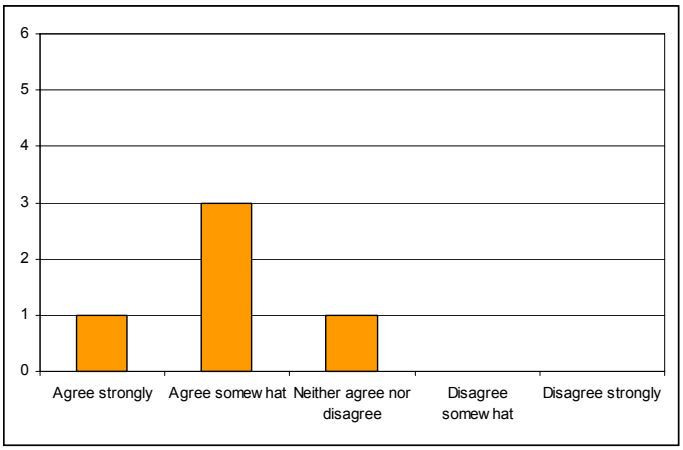
Agree strongly. When it was done. Not sure whether we could have used it more or not.

Agree strongly. This was done in off-line sessions with individual users, not in the plenary sessions. Whenever we did it, it added value

Agree strongly. I observed this rather than participated, but the message came across very strongly about how effective this was.

Commentary: This question was intended only for those developers who had been involved in the live one-on-one prototyping sessions.

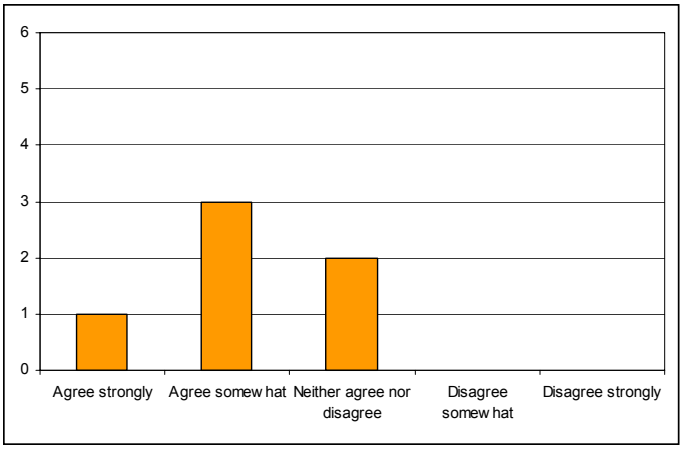
Q18. I found it easy to translate user requests into changes on the object model



Qualifications or comments made by individual respondents:

Agree somewhat. Generally easy, but some more complex user requests were initially difficult to translate.

Q19. The output of the Exploration phase provided a clearer basis for proceeding onto full-scale design and implementation than a conventional requirements specification

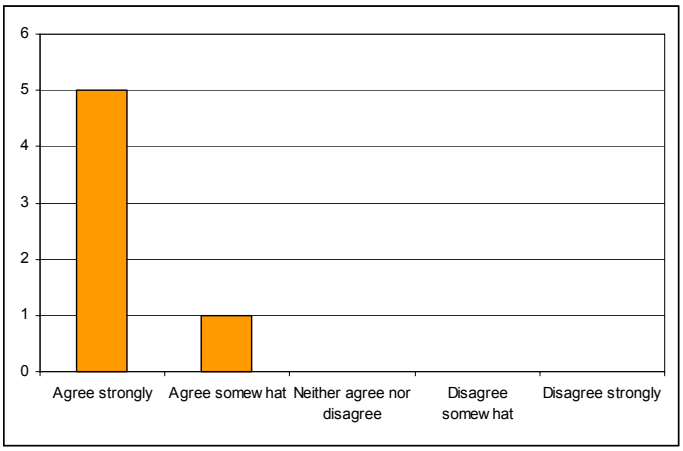


Qualifications or comments made by individual respondents:

Agree somewhat [CBP]. It made it easier to build the system 'right'.

Neither agree not disagree. Not clear on how the functionality provided by the prototype could or should be allocated (or prioritized) for delivery phases.

Q20. (Developers only) Using the Naked Objects approach significantly improved my understanding of object-oriented techniques in general



Qualifications or comments made by individual respondents:

Agree strongly. Could not agree more strongly!

Agree strongly. This was the best thing about it.

APPENDIX VIII. CARSERV - NOTES ON IMPLEMENTING THE CHANGE SCENARIOS

These notes were written entirely by Dan Haywood, and form an Appendix to Chapter 7 of the thesis. They describe the experience of implementing three different business change scenarios to both the implementations of *CarServ*. *CarServ1* refers to the original implementation, which adopts the conventional 4-layer model. *CarServ2* refers to the re-implementation using the *Naked Objects* framework.

Given the hypothesis that it would be easier to implement the changes on *CarServ2* than on *CarServ1*, it was decided each business change scenarios would be implemented on *CarServ2* before they were implemented on *CarServ1*. Given that there could potentially be some learning carried over from one implementation to the other, the order was deliberately chosen so that any bias would be against the hypothesis.

The 'Observations' are extracted from a lab-book, written by Haywood as the change scenarios were being implemented.

The Scenarios

The three change scenarios were conceived and described thus:

Scenario 1

Add some simple attribute to an object. Two examples were chosen:

(a) Add a field to hold an email address, potentially for both Customers and Employees.

(b) Add a derived attribute that calculates the net worth of a Customer (i.e. the sum of the value of all previous services)

Scenario 2

Support new subtype of service, such as a "winter tune-up service". Instead of having the capability to add specific parts and labour charges, the winter tune-up service would simply have a fixed price.

Scenario 3

Support for 'loaner' cars. The garage would hold a pool of potential loan cars (which would be owned by the garage rather than any customer). If a pool car was available when the customer checks his car in, then that pool car would be temporarily associated with the service object.

Changes Required

Scenario 1 (a) & (b)

CarServ2

Person is super class of Customer and Employee, and holds the emailAddress. Customer holds the 'update rating' responsibility, and delegates to Service for it to calculate its own total (makes up part of the rating).

Class	**Notes**
Customer	fieldOrder for "email address" rating field

	actionUpdateRating
Employee	fieldOrder for "email address"
Person	emailAddress field
Service	updateTotal() helper method
DATABASE SCHEMA	(auto generated for SQLObjectStore)

CarServ1

Employee and Person are not mentioned, since they are not fully implemented in *CARSERV1*. The emailAddress appears therefore only in Customer, but note how it appears in three tiers (datamgmt, domain, ui).

The 'netWorth' property is same as 'rating'. However, there is no responsibility assignment to an object, rather it has been "hidden" in the SQL that lists customers.

Class	**Notes**
datamgmt.CustDM	SQL for emailAddress x 2
datamgmt.CustomerData	emailAddress property netWorth property
domain.Customer	emailAddress property netWorth property
ui.CustomerCarPanel	emailAddress field netWorth field
ui.LookupCustomerDialog	email address in table
DATABASE SCHEMA	has the SQL.to do the aggregation of service's total worth

Scenario 2: fixed price service

CarServ2

Introduced new TimeMaterialsService, this now holds the labour and parts charges. Adhoc and Regular Service are subclasses of this new class, rather than directly from Service.

Introduced new FixedPriceService as sibling of TimeMaterialsService, which does not have labour or parts charges, but does have total.

Introduced new SpecialOffer, create FixedPriceService by dragging and dropping.

Car now has responsibility of being able to schedule fixed price service, similar to its support for the other existing service types.

Class	**Notes**
AdhocService RegularService	extends TimeMaterialsService
BeingServicedState PickedUpState ScheduledState ToBeScheduledState	enabling of some fields depends on whether service is a TimeMaterialsService
Car	actionScheduleFixedPriceService
DemoExploration	added SpecialOffer class
DiaryDay	might be a bug fix (2 lines removed)
FixedPriceService	new class
LabourCharge PartsCharge	links to TimeMaterialsService, rather than just Service
Service	some stuff moved down to TimeMaterialsService
SpecialOffer	new class
TimeMaterialsService	new class, superclass of Adhoc & RegularService

DATABASE SCHEMA	(autogenerated for SQLObjectStore)

CarServ1

Car similarly has support to schedule fixed price service, though impacts the datamgmt, domain, app and ui tiers.

There is no explicit support for SpecialOffer, but then it wasn't part of the specification.

Similar approach to TimeMaterialsService, though this is only made explicit in the domain tier.

In all tiers, have explicit case statements for testing the service subtype.

Class	**Notes**
datamgmt.CarDM	scheduleFixedPriceService, support for serviceTypeCode=FIXD listServices – serviceTotal & serviceTypeCode
datamgmt.ServiceData	total property serviceTypeCode property
datamgmt.ServiceDataHelper	support total & serviceTypeCode
datamgmt.ServiceDM	listServices – serviceTotal & serviceTypeCode updateServiceWithLabour – support total & serviceTypeCode updateServiceWithChargeableItem – support total & serviceTypeCode setTotal method
domain.Car	scheduleFixedPriceService
domain.Service	reset(), getServiceDM(), data accessible to subclass moved stuff to TimeMaterialsService
domain.ServiceHelper	serviceFor() – support FIXD type
domain.TimeMaterialsService	new class

	moved stuff from Service
app.BookInCommand	support FIXD type
app.CancelCommand	support FIXD type
app.ScheduleFixedPriceServiceCommand	new class
ui.CarServAppUI	mnemonic for scheduleServicePanel
ui.DiaryServicePanel	support FIXD type within insertService (service table)
ui.ScheduleServicePanel	mnemonic for option button scheduleFixedPriceService option button userSignedOn / userLoggedOut methods
ui.ServiceDetailPanel ui.ServiceLabourPanel ui.ServicePartsPanel	use TimeMaterialsService, not Service
DATABASE SCHEMA	*** NOT COMPLETED

note how the Car.actionScheduleFixedPriceService (in NakedObject implementation) hits four different tiers in *CARSERV1*.

conversely, there's no need to do any state management here (unlike NO), since the *CARSERV1* just doesn't expose it.

Scenario 3: Loan Car

CarServ2

Class	**Notes**
AbstractCar	new class
BeingServicedState PickedUpState ScheduledState	state control for action "show available cars"

ToBeScheduledState	
Car	extends AbstractCar moved model to AbstractCar
DemoExploration	added PoolCar class
DiaryDay	might be a bug fix (2 lines removed)
PoolCar	new class
DATABASE SCHEMA	(autogenerated for SQLObjectStore)

CarServ1

Class	**Notes**
datamgmt.CarDM	listAvailablePoolCars listServices – loanLicensePlate
datamgmt.CustDM	findCustomers – filter out garage as owner
datamgmt.ServiceData	loanLicensePlate property
datamgmt.ServiceDataHelper	support for loanLicensePlate property
datamgmt.ServiceDM	updateServiceWith – loanLicensePlate bookInCar, takes loanLicensePlate listServices – loanLicensePlate
domain.Service	bookInCar, takes loanLicensePlate loanLicensePlate property
app.BookInCommand	doExecute, pass in loanLicensePlate as arg
ui.BookInDialog	new class
ui.DiaryServicePanel	use BookInDialog comment out support for in-situ specifying of mileage.

DATABASE SCHEMA	*** NOT COMPLETED

Metrics

The following metrics were captured:

LOC: Lines of Code

NOC: Number of Classes

NOO: Number of Operations

NOA: Number of Attributes

PS: Program Size

VS1: Vocabulary Size 1 (# distinct classes used)

classes in `java.util` and `java.lang` ignored

VS2: Vocabulary Size 2 (# distinct method calls)

classes in `java.util` and `java.lang` ignored

The metrics can crudely be categorized as either measuring a "benefit" – a larger number implies some level of value - or as a "cost" – a larger number implies some level of cost incurred within the implementation:

	Benefit	**Cost**
LOC NOC NOO NOA	These metrics measure the size of the code; more code means more features. (Of course, the feature "density" depends upon the implementation platform).	
PS VS1 VS2		These metrics measure the cost of implementation, in terms of programmer productivity / training.

		LOC	NOA	NOC	NOO	PS	VS1	VS2
CarServ1	**orig**	7304	289	190	788	3180	142	411
	scenario 1	7385	295	190	794	3257	142	415
	scenario 2	7498	297	194	801	3271	144	430
	scenario 3	7450	295	191	798	3289	143	432
CarServ2	**reimpl**	1726	62	27	230	747	18	56
	scenario 1	1770	64	27	234	775	18	58
	scenario 2	1906	68	30	249	795	18	58
	scenario 3	1907	65	30	251	829	20	62

Timings

Scenario	***CarServ2***	***CarServ1***	**% Time**
1 (a)	18 mins	56 mins	32%
1 (b)	41 mins	58 mins	71%
2	131 mins	116 mins + 60 mins* (estimate for GUI) = 176 mins	75%
3	220 mins	276 mins	80%

*The work actually done did not include implementing a GUI. With appropriate scaffolding in place (none such exists in the original implementation), a very optimistic estimate of a further one hour to implement the GUI has been made.

Observations

Scenario 1 (a): Person.emailAddress

CarServ2

18 minutes

CarServ1

56 minutes

Finished the implementation after 45 minutes, but then thought further:

> *" the problem here is that there is nowhere to show the customer's data. Have decided to show it in the rhs panel on the diary tab..."*

I also added …

> *"the fact that there is a five minute gap is that I'm having to figure out what the correct representation on the UI should be. No such issues arise for a CarServ2objects implementation, of course (other than figuring out the field order...)"*

There was another 11 minutes making these additional changes, bringing the whole up to 56 minutes.

Scenario 1 (b): Person.netWorth

CarServ2

41 minutes

Had problems with an infinite loop, to do with lifecycle issues. (Still not convinced that I have fixed correctly).

CarServ1

58 minutes

The multiple-tier architecture for *CARSERV1* caused me to have to make some decisions:

> *"decided to implement the net worth directly within datamgmt layer. Doesn't really expose the business logic (there s a similar remark in scenario 3, which I did first). Could*

argue legimately that I have done a premature optimization here (one that is not really properly documented)."

Scenario 2: FixedPriceService

CarServ2

131 minutes

Following observation on NakedObjects design:

*"we considered an enhancement whereby the ServiceDefinition could define both parts and labour charges. However, the problem is that our labour charges are expressed in terms of "green" employees, not "blue" labour rates. Contrast this with the parts charges where these *do* refer to "blue" parts. So, basically, the situation - even though it looks like it is symmetricl - is not. This is much easier to see when the colours are applied. We decided to reject this scenario because adding in labour rates would be just too big a change"*

CarServ1

116 minutes.

This time is less than the *CarServ2* Objects equivalent, but there are some important points to make:

First, no GUI was implemented (since the original *CARSERV1* implementation was incomplete in this respect anyway; it would have been unfair to include the time to build "scaffolding"); highly optimistic estimate for a GUI if appropriate scaffolding had been in place, 60 minutes.

Second, I did implement the *CARSERV1* changes after the *CarServ2* changes, also this was the very last *CARSERV1* scenario that I did. Therefore, probably getting faster.

Towards the end of the day, I had this to say…

"although the CARSERV1 implementation of carserv is not as fully featured as CarServ2objects, its additive (rather than subtractive) nature means that adding features and enhancements is more "controlled". That is, since the entire behaviour of the app is under the developer's control, it is possible to stop adding features at any given point. With nakedobjects, on the other hand, adding a new feature (e.g. a field) may have a ripple effect to some extent. Hmmm - not a very convincing paragraph when I read it back. Oh well, brain is starting to tire a little ..."

In other words, it can be hard to control the increments under NakedObjects.

I also made the following observation:

> *"under the CarServ1.carserv, feel it is much easier to cobble it together; put another way, naked.carserv requires a more "honest" approach to OO, can't really cheat"*

For example, in this scenario there are lots of places (in every tier) where the subtype is explicitly tested, either with the `instanceof` operator, or by checking the value of a type column. Don't get this in the NakedObjects implementation.

Scenario 3: Loan Car

CarServ2

220 minutes

This time was inflated somewhat by learning about NakedObject lifecycle methods (about methods, and when they are called). There was also a change in design half-way through, when decided not to model the garage as an explicit entity after all.

Also, the SQLObjectStore does not support resolving 1:1 references to abstract class. So, unless have already resolved the services, attempting to view them from PoolCar will fail.

CarServ1

276 minutes.

This scenario would have taken longer to code, but I decided not to implement in-situ entry of mileage/loan car, and also decided not to worry about stale lists of available pool cars.

I had some observations on the design of state management in the original implementation:

> *"The CarDM & ServiceDM's scheduleService, bookInCar and pickUpCar methods control the state lifecycle of the service … see how distributed this logic is... :-("*

And again:

> *" the business logic in the SQL to determine which cars are available. This is probably defensible on performance grounds, but the design of carserv makes it the natural place to put it, rather than putting it there as an optimization (moved out of the domain layer)"*

Later on in the day (getting somewhat tired with the work, as a developer would), I noted:

"somewhat hacky, in that the UI talks directly to datamgmt layer; should really (I guess) have put in some intermediary functionality in the 'app' layer. However, the fact that I didn't could be a discussion point (its easy with multiple layers to "not bother" ... so long as the direction of dependencies is still correct ... etc etc)

APPENDIX IX. PUBLISHED PAPERS

Two published papers are reproduced here in their original formats.

The first, entitled 'Naked Objects: A Technique for Designing More Expressive Systems' was selected by the program committee as one of four 'Intriguing Technologies' to be presented at the OOPSLA 2001 Conference in Tampa, Florida. The paper was subsequently published, along with the other papers from that track, in SIGPLAN Notices December 2001 (Volume 36, Issue 12, pages 61-67).

The second, entitled 'Agile Development with Naked Objects' was presented at the 4th International Conference on Extreme Programming and Agile Methodologies in Software Engineering (XP2003) in Genoa, Italy, published by Springer Lecture Notes in Computer Science.