Naked objects: a technique for designing more expressive systems

Richard Pawson
Computer Sciences Corporation
and Computer Science Department,
Trinity College, Dublin, IE
rpawson@csc.com

Robert Matthews
NakedObjects.org
rmatthews@nakedobjects.org

ABSTRACT

Naked objects is an approach to systems design in which core business objects show directly through to the user interface, and in which all interaction consists of invoking methods on those objects in the noun-verb style. One advantage of this approach is that it results in systems that are more expressive from the viewpoint of the user: they treat the user like a problem solver, not as merely a process-follower. Another advantage is that the 1:1 mapping between the user's representation and the underlying model means that it is possible to auto-generate the former from the latter , which yields benefits to the development process. The authors have designed a Java-based, open source toolkit called Naked Objects which facilitates this style of development. This paper describes the design and operation of the toolkit and its application to the prototyping of a core business system. Some initial feedback from the project is provided, together with a list of future research directions both for the toolkit and for a methodology to apply the naked objects approach.

1. INTRODUCTION

Advocates of object-oriented user-interfaces (OOUIs) argue that object-orientation is potentially a user concept. In an OOUI the user can identify the objects that he or she is working with, ascertain the behaviours offered by those objects and invoke them directly on those objects. (OOUIs combine well both with graphical representations and with the principles of direct manipulation but these concepts should not be confused. You can certainly have graphical, direct manipulation interfaces that are not object-oriented: the point-and-click calculator is an example, and it is most unfortunate that this is sometimes cited as an example of an OOUI [2]. It is also possible to have a text-based OOUI, as demonstrated in the earliest versions of SmallTalk [7]. However, for the remainder of this paper where we refer to an OOUI, we assume that it supports graphics and direct manipulation.)

OOUIs provide a number of advantages to the user including reduced modality [6], and greater expressiveness for the user [10], and have become the norm for 'creative' applications such as drawing, computer aided design (CAD), page make-up and multimedia editing. OOUIs are far less common in transactional business systems. This may be because the need for expressiveness is less clear. It may also be because the underlying structure of most transactional business applications, comprising scripted procedures and data operations, does not map well onto an OOUI.

Surprisingly, though, even where a transactional business system is based upon a true business object model, the core business objects are typically hidden from the user. The users view of the system, and interaction with the system, continues to be defined by a user interface that has been optimised to a particular set of tasks. The artifacts of this user interface - menus, buttons, forms, reports, dialogue boxes and message windows - are common to most business systems. The user would usually have no indication whether the core of the system was an object model or a conventional composition of functional procedures and data elements. And many software designers would argue that this is right and proper: that one of the purposes of multi-layered architectures is to shield the user from the structure of the underlying software. Object-orientation, this view suggests, is a tool for analysts and/or programmers, offering benefits such as improved development productivity, software maintainability, or overall quality.

A few years ago, we became interested in trying to reconcile these two views. There already exists a body of research on combining OOUIs with object modelling techniques [13], but we wanted to take a more radical step. We wanted to explore the possibility of designing core business systems where there was a 1:1 correspondence between the user's view and the underlying business object model, and where *all* user interactions would take the form of invoking a method upon an instance of a core business object class, or a method on the class itself. In other words the style of interaction would be entirely noun-verb instead of the more common verb-noun style used in most core business systems [11]. Our goal was to eliminate every other construct from the user interface - no forms, no dialogue boxes, no message windows - and get back to just the business objects. We nicknamed this the 'naked objects' approach.

We had two main hypotheses. The first was that the naked objects approach would lead to systems that were more expressive from the perspective of the user. We meant something more than just making it easier to realise the functional benefits of OOUIs in core business systems. Rather we sought to achieve what Hutchins, Hollan and Norman refer to as 'direct engagement': "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."[4]. On a related theme Laurel suggests that "Operating a computer program is all too often a second-person experience: A person makes imperative statements, or pleas, to the system and the system takes action, completely usurping the role of agency . . . Even though we are in fact agents by virtue of making choices and specifying action characteristics, these shadowy forces manage to make us feel that we are patients - those who are done unto rather than those who do."[9].

In other words, our first hypothesis was that systems based on naked objects would empower the user. We recognised, even at the outset, that not everyone would agree that empowering users was important or even desirable in the context of key operational business systems. However, our initial concern was with whether it was possible to design such systems to be user-empowering: if we proved that then we would return to the business arguments for so doing, later.

The second initial hypothesis was that the naked objects approach could offer several benefits to the development process, irrespective of any desire to produce a different experience for the end-user. If the user-interface was envisaged as a 1:1 mapping from the core business object model, then it should be possible to generate one from the other, automatically. This would save time and effort in development. More importantly, perhaps, it would provide a common language between users and developers, and this could change the whole style of requirements gathering, prototyping and design. Many people have discussed the advantages of objects for bridging the semantic gap between programmers and business analysts, but not typically between programmers and the end-users themselves.

In this paper we describe the development of a toolkit to support this approach, our early experiences of applying it, and some of the evidence in support of our initial hypotheses. We then describe our intentions for the future development of the toolkit, some additional hypotheses concerning the benefits, and a more formal programme of research to be conducted.

2. DESIGN AND OPERATION OF THE TOOLKIT

In 1999 we started to specify toolkit, which we now call Naked Objects[1]. The toolkit is open source and can be downloaded from www.nakedobjects.org. Being Java-based, the toolkit will work happily on Windows, Mac, Linux and other client platforms. Java is increasingly popular as a programming language based on its suitability for adopting best-practice design patterns, and its ease of refactoring. Allied to the J2EE enterprise services, Java is well suited to the implementation of transactional business systems. However, our decision to use Java was also influenced by specific features of the language and platform, such as the use of interfaces to support polymorphism, the idea of 'reflection' [1], whereby an object can be interrogated by another object, at run-time, to reveal its methods, and the ability to load new objects or changed object definitions into the running system, dynamically.

Using Naked Objects, the programmer specifies the core business objects (such as Customer, Order, Product) in the same form as the 'model' objects would be specified in a classic model-view-controller [8]arrangement. However, in Naked Objects the programmer does not go on to specify any 'view' and 'controller' objects - these responsibilities are provided, transparently, by the toolkit in the form of a ready-built Object Viewing Mechanism or OVM. When the set of business object (model) classes is loaded into the toolkit, the OVM uses reflection to inspect these business objects, and portray them on the screen. In other words the programmer need give no thought whatever to the design of the user interface. Indeed, a programmer developing a straightforward business application using Naked Objects would not come into direct contact with the OVM. (A programmer would only come into contact with the OVM if they were seeking to extend the capabilities of the toolkit itself). From the perspective of the programmer, the system consists solely of the model, and each business object class effectively inherits the ability to display itself and its business behaviours directly to the user. (In this respect toolkit has some conceptual similarities with the Morphic user interface originally developed within the SELF language [12] and subsequently adopted within Squeak [5]).

The current version of the OVM assumes that the client device has a high-resolution graphical display, a mouse, and a high bandwidth connection to the other tiers of the architecture. It generates a user interface with a very specific, and consistent, look and feel. It represents each business object instance as an icon that can be selected, dragged and dropped, right-clicked to reveal a pop-up menu of business behaviours specific to that object type, or 'opened' to view its attributes and associations. This look and feel will be immediately familiar to anyone who has used a desktop metaphor. By default, the OVM also creates a window containing a set of icons that represent the classes themselves, and this is the means by which the user can perform class methods such as creating a new instance, searching for an existing one, or performing an action across all instances, such as generating a report.

[1] The first version of the toolkit was known as Expressive Objects

However, we expect in future to devise other OVMs to create different kinds of user interfaces, again using reflection. For example, we expect at some future point to provide a separate OVM that will be designed for an HTML-only browser interface, and assumes a low-bandwidth connection. (Our current mechanism could be used via a browser, but only by generating a large Java applet). The HTML-only version could not implement drag-and-drop, but it might represent each object as a page, with the actions or verbs shown as hyperlinks. Separate OVMs could theoretically be generated for PDAs, WAP phones, command-line interfaces, or voice-response systems. The user's interaction gestures would be different in each case, but each would strive to implement the underlying principles of an OOUI (specifically, the noun-verb style of interaction) to the maximum extent possible.

Within a specific OVM, the user may still choose several ways to view a particular object. The OVM uses the common Strategy pattern [3] to create alternative views of an object within a lightweight Component (as done in Swing with pluggable look and feels). For example a Collection (of business objects) can be viewed and manipulated as a single icon, or it can be viewed as a list of the objects it contains - each of which could be dragged out, or expanded *in situ* to view its contents. If the Collection is homogeneous (all the business objects it contains are of the same class e.g. Product), then the user is automatically given the option to view it as a table. These viewing options are shown in the right-click menu, above the business methods.

3. USING THE TOOLKIT

The business objects themselves are written in standard Java. The programmer must follow a few simple conventions to ensure that the objects can work together in a dynamic fashion, and that the OVM can display the object, but where possible, these conventions have been based on existing practice. As with JavaBeans we use reflection to find specific methods that relate to specific attributes and to specific types of behaviours - for example, the OVM looks for method names beginning with 'set' and 'get', and also ones that start with 'action'.

When the OVM observes a 'get' method in a business object, it automatically creates a field to represent the appropriate attribute in the viewer for that object, and if there is a corresponding 'set' method, then the user can change the contents of that field. If the argument of the get and set methods is another business object (i.e. an association) then the OVM will display an icon representing that object, which may itself be right-clicked to view its contents or to invoke a method. The user automatically acquires the ability to drag and drop any object of the specified type into and out of that field. (Attempting to drag in the wrong type of object causes the field to flash red momentarily). In addition to business objects, accessor methods may refer to primitive object types such as text, dates and numbers. Here we have adopted our own small set of primitives based mainly on existing Java types, but which can be interpreted by the OVMs. Thus TextString objects can be portrayed as an editable text field, and Logical objects can be portrayed as a check boxes.

Any method name with the prefix 'action' automatically becomes a command in the pop-up menu for that object. By default the OVM just strips the prefix, so that 'actionCommunicate' will show up in the menu as Communicate. The programmer may also create corresponding methods called 'aboutActionCommunicate' to determine the circumstances in which that menu item will be unavailable (e.g. greyed out) or not displayed at all. These techniques can be used both to enforce universal business rules, such as that you can't create a refund to a customer unless the original purchase has been specified, and also to permit fine-grained levels of user authorisation. At present the system does not indicate to the user why a drop is not possible or why an action cannot take place. In the future an object will be returned by these methods, and the object will contain the reason, very much like the 'Throwable' objects is Java.

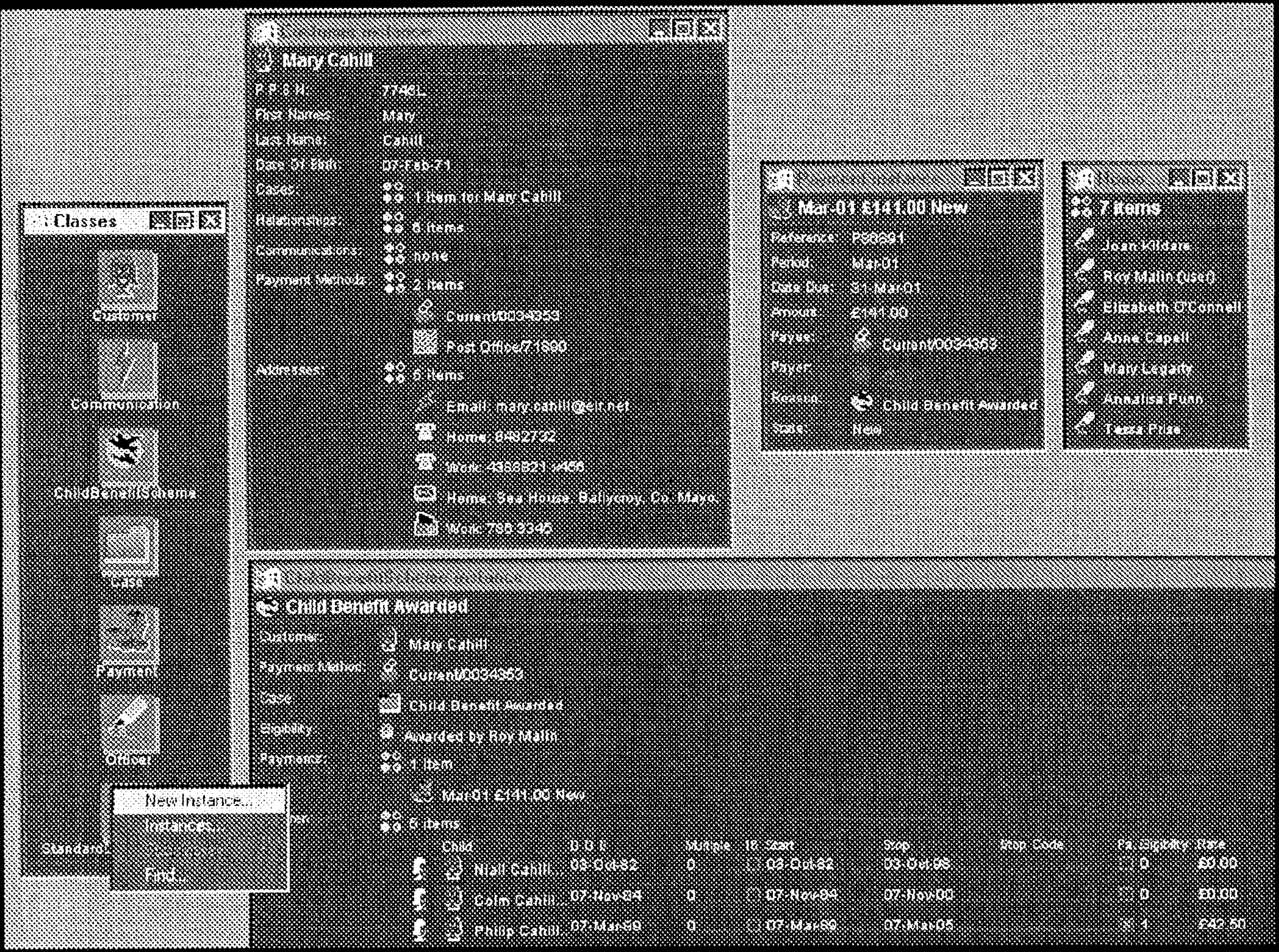
We use the keyword 'process' to provide a standard mechanism for specifying how one object type interacts with another. Our first OVM calls this method whenever one object is dragged over another object (as distinct from being dragged directly into a specific field within that object). However, it is important to understand that, as with the action methods, the process method is not specifically related to the user interface: it may be called within the program itself.

Certain requirements must be met when declaring classes else the system will not be able to use the objects, specifically the OVM will not be able to create views. For example there must be a default (zero parameter) constructor, in order to allow the user to create new object instances. Another is a 'toString' method capable of generating a concise description or summary of each object (for example a concatenation of invoice number and date) which our OVM uses to label the icon. The programmer may optionally declare that the object keeps a 'Status' and provide a method to return a status code for the object. Our OVM uses this to select between different versions of an icon to show an object: such as an open folder or a closed folder.

The programmer must make each business object implement the NakedObject interface. We provide, for convenience, abstract classes that provide the majority of the functionality required by the NakedObject interface - so that the programmer can simply specify their Customer and Product classes as extensions to (i.e. subclasses of) the NakedObject class. In the future, we will add helper classes, so that existing business objects that must inherit from other class hierarchies, can delegate most of these required requests to the helper classes.

4. AN EXAMPLE APPLICATION

Figure 1 shows a typical screen from a system built using Naked Objects. It is taken from a prototype developed in 2000 by the Irish Department of Social, Community and Family Affairs (DSCFA) for a new benefits administration system.



Mary Cahill

P.P.S.N.:	7746L
First Names:	Mary
Last Name:	Cahill
Date Of Birth:	07-Feb-71
Cases:	1 item for Mary Cahill
Relationships:	6 items
Communications:	none
Payment Methods:	2 items
	Current/0034353
	Post Office/71890
Addresses:	5 items
	Email: mary.cahill@eir.net
	Home: 8492732
	Work: 4388821 x456
	Home: Sea House, Ballycroy, Co. Mayo.
	Work: 795 3345

Mar-01 £141.00 New

Reference:	P90091
Period:	Mar-01
Date Due:	31-Mar-01
Amount:	£141.00
Payee:	Current/0034353
Payer:	
Reason:	Child Benefit Awarded
State:	New

7 items

- Joan Kildare
- Roy Mallin (user)
- Elizabeth O'Connell
- Anne Capell
- Mary Legarty
- Annalisa Punn
- Tessa Prise

Child Benefit Awarded

Customer:	Mary Cahill
Payment Method:	Current/0034353
Case:	Child Benefit Awarded
Eligibility:	Awarded by Roy Mallin
Payments:	1 item
	Mar-01 £141.00 New
	5 items

Child	D.O.B.	Multiple	18 Start	Stop	Stop Code	Ea. Eligibility	Rate
Niall Cahill...	03-Oct-82	0	03-Oct-82	03-Oct-98		0	£0.00
Colm Cahill...	07-Nov-84	0	07-Nov-84	07-Nov-00		0	£0.00
Philip Cahill...	07-Mar-89	0	07-Mar-89	07-Mar-05		1	£42.50

Figure 1: The prototype Child Benefit Administration system

A window on the left hand side lists the six primary classes of business object that make up the system. 'Customers' are the individuals involved in a claim. 'Communications' include emails, printed letters, phone calls, or notes from face to face meetings. 'Cases' are folders that hold all the information associated with a claim and which permit the current workload to be monitored and allocated between Officers. A 'Scheme' object can be thought of as a worksheet used by an Officer for collecting the supporting information, recording formal decisions, and for calculating entitlement levels. (The Scheme class will eventually have some forty sub-classes representing the various legislated benefit schemes that the DSCFA administers. This screen shows only the sub-class for the Child Benefit Scheme - which will be the first one to be implemented on the new system.) 'Payment' objects represent actual benefit payments, whether as cheques, electronic funds transfers, or books of vouchers to be cashed at a local Post Office. 'Officers' are the employees of the DSCFA that administer benefits.

Clicking the right-hand mouse button on one of the six icons representing the core classes will bring up a menu of class that the user can invoke. These class methods are mostly generic: create a new instance of this class, retrieve an instance from storage, list instances that match a set of criteria, and show any sub-classes available to the user.

Elsewhere on the screen there are a number of individual object instances, either minimised to an icon, or expanded to view its contents. The contents consist of primitive data such as numbers and text, and other object instances represented as small icons with labels. In addition to instances of the six primary classes you will also see some instances of secondary business object classes such as Payment Method, and Address (for communication).

Right-clicking on any object instance, in any context, reveals the menu of the instance methods available to the user. A few of these methods are generic, including different ways to view the object, but most are specific to that type of business object. For Customer, the instance methods include Authenticate, Communicate, Register new child, and Request an update of basic details. Some of these methods perform simple operations on that object alone; others initiate more complex transactions.

Apart from these right click menu methods, the only other way to invoke any functionality is by dragging and dropping objects onto other objects, or into specific fields within objects. The meaning of any such drag and drop depends upon the objects involved. For example, dragging a Customer instance onto the Child Benefit class would check whether that customer was already claiming child benefit, and if not, initiate a new claim by creating an instance of the Child Benefit Scheme, inserting a reference to the customer object inside it, and inserting the whole within a new instance of Case, with the case owner being the object representing the officer logged onto the system.

Which menu operations and which drag-and-drop operations are permissible depend upon the authorisation levels of the specific user, and on the context. For example, the Communicate Decision (. . to the customer) method on the Scheme Object will be greyed-out until a formal sign-off has been made. Thus, the system does enforce business rules, but these are contained entirely within the business objects, not represented in some external rule base or set of scripts.

5. FEEDBACK

When the prototype Child Benefit Administration system was first shown to the senior management of the DSCFA, three reactions were noted:

'We can see how everyone in the entire organization, right up to [the Government Minister], could use the same system'. This did not mean that all users would be doing the same operations, or indeed have the same levels of authorization - it meant that they saw how everything the organization did could be represented in terms of direct actions on the handful of key business objects representing the departments sphere of responsibility. Such a consistent interface could help to break down some of the divisional barriers, as well as making it easier for individuals to move into new areas of responsibility.

'This interface might be sub-optimal for high volume data entry tasks' There was some debate about whether this was in fact the case, but this evaporated when it was pointed out the DSCFA's commitment to the various e-Government initiatives, including the ability to make enquiries and initiate applications for benefits online, much of the routine data entry work is expected to disappear over the next few years. This led to the third comment.

'This system reinforces the message we have been sending to the workforce about changing the style of working'. The DSCFA is committed to moving away from an old fashioned assembly-line approach to claims processing (where each person performs a small step in the process) towards a model where more of its people can handle a complete claim, and experienced officers could handle all the benefits for one customer. The prototype conveyed a strong message to the users: you are problem solvers, not process followers. This was in marked contrast to the approach that several systems vendors had proposed for the replacement system, using rules-based technology, workflow engines, and software agents to allow the system to automate as much decision making as possible. The system prototyped with our toolkit had nothing in the way of 'artificial intelligence'. Rather, it provided an environment where employees' natural problem-solving skills would be highly leveraged.

Shortly after this, the prototype was shown to a handful of Child Benefit administration officers - the eventual users of the system - excluding those who had been involved in the design process. Some trepidation was observed, but much of this turned out to be because several of those present had no previous experience of using a PC, the current system being green-screen dumb-terminal based. General windows and PC training will obviously be provided before the new system is rolled out. It was striking, though, that within a few minutes of their first exposure to the prototype, these users had picked up the object language and were already asking questions about the system in those terms, such as 'Could I have two Customer objects open at once?' and 'If I dragged a Customer object onto the Payment Class, would that tell me whether we had made any recent payments to that Customer?' The point is that this style of system seems to encourage an exploratory attitude that normal systems do not. In this sense it is very much like the web, but with a bias towards conducting transactions rather than browsing published information.

Whilst this is not sufficient evidence to consider our first hypothesis (that naked objects empower the users) proven, these positive reactions from real business sponsors and users are very encouraging. Similarly, our experience in developing the DSCFA prototype has given us some very positive feedback in regard to the second hypothesis (that designing with naked objects would improve the development process). The ability to translate object ideas into a tangible form that the users could see and manipulate, proved to be a far more effective way to get business representatives involved in the object modelling process than drawing UML diagrams. (In fact, we did not draw a single UML diagram during that entire process). Moreover, the speed with which ideas for new objects, or modifications to existing objects, could be translated into this tangible form, meant that we often found ourselves doing live prototyping in front of the customer. Finally, we found that, compared to previous object modelling exercises that we had been involved with, the concrete form of the naked objects helped to avoid many of the protracted debates about abstraction in general, and inheritance hierarchies in particular, that tend to plague object modelling.

6. FUTURE DIRECTIONS

The DSCFA Child Benefit system has now moved from a prototyping stage to full-scale implementation, with roll-out scheduled for early in 2002. The full-scale implementation is being implemented using different tools (centred on the COM+ object technologies) but both the design ethos and the user interaction remain strongly consistent with the prototype that we described.

We have now also started to work with a handful of other organisations wishing to explore the naked objects approach. One of these, intends to use our Naked Objects toolkit not only for prototyping, but for full-scale implementation, by extending the toolkit to work with their existing J2EE infrastructure. We hope to be able to report on this shortly.

Meantime, we have a lot of work to do on the Naked Objects toolkit itself. Over the next few months we shall be concentrating on the integration of the toolkit with enterprise object services so that we can be confident of building large scale systems using this approach. Then there are many minor improvements needed to the basic look and feel, in which we must draw on best practices from usability engineering and user interface design, yet without getting drawn into the trap of optimising the user interface to a specific task whilst losing the overall expressiveness. We need to improve the toolkit's current handling of associations, with the aim of eventually making it as rich as the UML notation in this area. We would like to extend the Classes window into a more general class browser (but oriented to users, not developers). As previously mentioned, we also want to generate alternative Object Viewing Mechanisms for different user platforms including the web browser.

Our greatest research effort, however, will now be concentrated on the development process. It is not our intent to develop another new methodology. We believe that the naked objects approach is broadly compatible with the bulk of the more modern, or lightweight, development methodologies and may offer some significant advantages to them. At the same time, we suspect that the naked objects approach may suggest, may even require, some subtle changes to those methodologies, and where this is the case, we want to be able to clearly demonstrate the advantage of adopting these changes. We are especially interested in exploring the benefits of using the naked objects approach within the context of Extreme Programming [2]. Our first experiments in this direction are promising, but we need more experience before reporting them.

We are aware that there are many queries, concerns and objections that we will have to address before the naked objects approach will be more widely accepted. A common question from the development side concerns scalability. If scalability is taken to mean number of users, volume of transactions, or the size of relational database tables used to persist the objects, then this is more a function of the underlying object services. The Naked Objects toolkit is primarily concerned with the relationship between the middle tier and the client presentation and makes few assumptions about the connection between the middle and lower tiers.

A more valid concern is the possibility of 'bloated' objects. If all business functionality must be encapsulated in the core business objects, then as these objects start to be shared between different types of user or application, won't they necessarily become bloated with attributes, associations and methods? The toolkit already provides mechanisms for limiting which attributes, associations and methods are shown to the user, according to the role(s) that user fulfils. This removes much of the clutter. Additionally, we have found that this risk of bloated objects forces good partitioning of responsibilities between objects in the model. In several of the prototypes, for example, the Customer object has started to become bloated, but we have easily been able to delegate responsibilities onto associated or aggregated objects, immediately accessible to the user. Users are typically comfortable with this approach.

A common issue from the user-side, or at least from some business managers purporting to represent the users, concerns business controls. How can a system that permits the user to conduct any action in any context provide necessary business controls? In fact, as we have already seen in the DSCFA example, the system does support the enforcement of essential rules and controls. But these are implemented as responsibilities of the objects affected, in other words at the lowest level possible, not by limiting the users interaction to a small number of tightly proscribed scripts or processes. Nevertheless, this is an area where we wish to do more formal evaluation.

7. CONCLUSION

So far, we are aware of approximately 50 software developers whom we have taught, or who have taught themselves, to use the naked objects approach (in most cases using our Naked Objects toolkit). The majority have become very enthusiastic about it. Frequently, they cite its 'flexibility' as a plus point. This is a very interesting response, because the naked objects approach is almost draconian in the constraints that it places on the developers: no, you can't design screens or form layouts; no, you can't write user scripts or dialogue boxes; no, you can't generate a top-level menu. Yet very quickly these constraints seem to become invisible as both the developers, and the users involved in the design, learn to think in terms of pure, naked objects.

8. REFERENCES

[1] *Java Core Reflection.* 1997, Sun Microsystems.

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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(j): p. 26-49.

[9] Laurel, B., *Computers as Theatre.* 1991, Reading, MA: Addison-Wesley.

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

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

[12] 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.

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