-+

METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1023-4918**

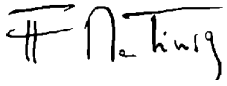
Spring 2003 (Volume 11 - number 1)

Software depollution

One of the articles of this issue explains the concept of refactoring. Refactoring is defined as "improving the design of existing code without changing its observable behaviour". This issue is related to an old problem of software development: maintenance. The management of existing programs to correct or improve them could be considered as the most neglected area of software development, even if studies indicate that it constitutes the majority of the costs of software applications. Developers love to "create" new programs, but they dislike modifying existing ones, even more if the program has been written by someone else. Methodologists will dedicate most of their effort to describe how to build a system, but few or no words on how to handle it after it is delivered to the end-users. Mostly, maintenance is considered has "a kind of another development project", setting aside the special tasks implied by the understanding and modification of an existing application.

It is initially a little bit amazing to see that the "management of existing programs" is considered as important by the new agile methodologies where development is iterative and incremental, but it is logical. When approaches like eXtreme Programming advocate limiting initial design and building the system in short iterations that last one to three weeks, it is important to be able to work efficiently on existing programs. You face the same issues than in the evolution of legacy applications, even if it should be easier, because the elapsed time since the last modification should be shorter than in classic maintenance project.

In the end, and regardless of the development method or programming language that has been used, the main responsibility of producing maintainable code lies on the shoulder of the programmer. Usage of comments, explicit language constructs and meaningful naming of variables are some of the simple actions that can make a program easier to understand and modify. At a time where many are trying to respect ecological principles in their life, let's try to do it also in our work and to leave our code as clean as we would have like to find it.



Inside

The Case for Naked Objects ..... page 2

Refactoring..... page 20

How to Select a QA Collaboration Tool ..... page 26

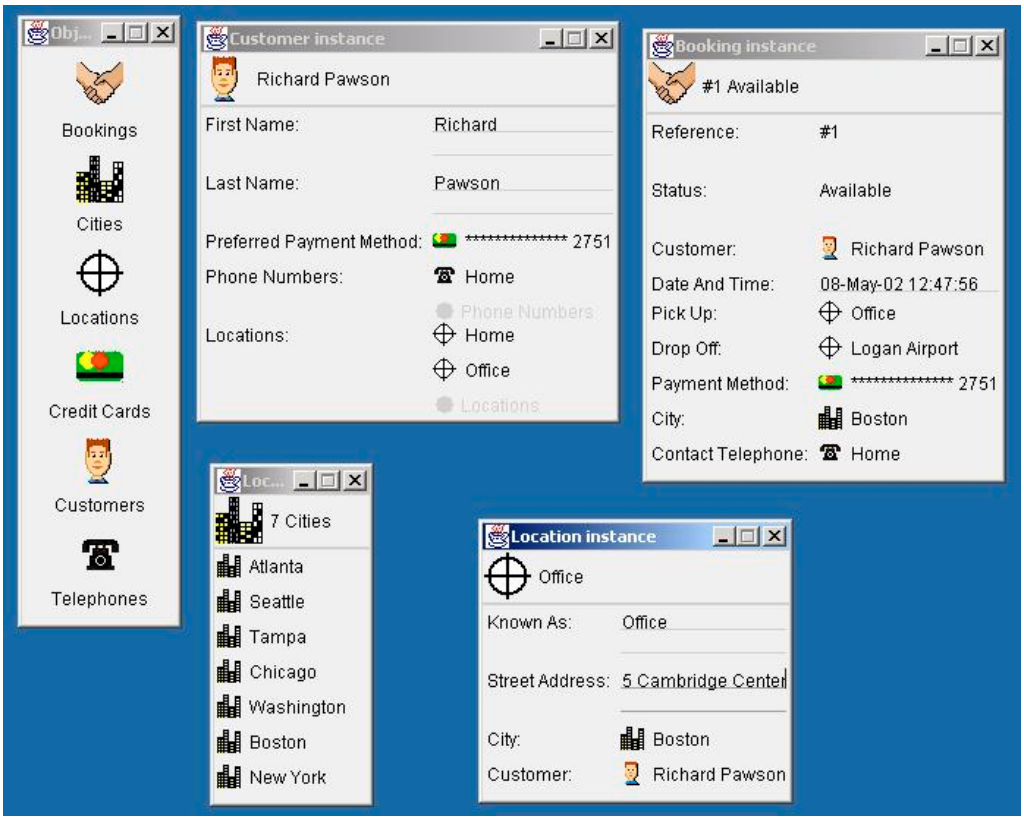
The Case for Naked Objects

Richard Pawson, rpawson@csc.com
Computer Sciences Corporation, co-founder of www.nakedobjects.org

Introduction

Naked objects[9] is an architectural pattern such that the core business objects (such as *Customer*, *Product* and *Order*) are rendered directly visible to the user rather than hidden behind the menus, forms, process scripts and dialog boxes that make up most user interfaces. In a system built according to the naked objects pattern, all operations are performed by selecting a particular object (i.e. a noun) and then invoking a behaviour (i.e. a verb) that is an inherent property of that object.

The screenshot below is taken from an application designed using naked objects. The 'Classes' window lists the core classes of business object available to the user. Right-clicking on any of these icons reveals a pop-up menu of 'class methods', which typically include creating a new instance of that class, retrieving an existing instance from storage, listing all the instances that fit a set of criteria, and showing any sub-classes that are accessible to the user. Elsewhere on the screen you will see icons representing individual instances of these classes, some of which have been 'opened' to show their attributes and associations with other objects. Right-clicking on any of these instances reveals a pop-up menu of 'instance methods', which include some generic methods for deleting, copying and viewing the object in different ways, and the business behaviours specific to that object.



The idea of object-oriented user interfaces is not new. What makes the naked objects idea unique is that it requires a 1:1 correspondence between the objects that the user is manipulating on the screen and the business objects in the business logic layer of a multi-layered architecture. Most approaches to systems design not only permit the presentation layer to use different constructs from the business logic layer, but actively encourage it. That philosophy is now so ingrained that it is hard for many systems designers to conceive an alternative. The best way to overcome this tendency is to use a toolkit that enforces the 1:1 correspondence. The best known example of such a framework was written by Robert Matthews and is known simply as *Naked Objects*. It is Java-based and fully open source - you can download it from www.nakedobjects.org.

For most applications involving a graphical user interface, especially those with direct manipulation gestures such as drag and drop, a high proportion of the effort usually goes into coding the user interface. Yet the programmer who generated the prototype shown in Figure 1 did not have to write one single line of code to do with the user interface. Using the *Naked Objects* framework, all that the programmer specifies is the business objects: their attributes, their associations with other business objects, and the business methods that they can fulfil. When the object definitions are compiled and run, the framework's 'viewing mechanism' automatically translates each object's specification into a screen representation. Individual objects show up as icons that can be dragged and dropped, or 'opened'. In the opened view, attributes – such as text strings, numbers and dates – show up as user-editable fields (unless the programmer has specified them as read only). Business methods translate into menu actions that appear when you right-click on an object icon. Associations are shown as icons representing the associated objects. Importantly, wherever you see an icon, in any context, it provides full access to the behaviour of that object. So, instead of an Order object having a field for the 'Name of the Customer', it will have a field containing the appropriate Customer object, and if the user wanted to communicate with that customer there and then, she could right-click on the icon and select the 'Communicate…' action.

Business rules are written into the business objects themselves, and, where appropriate, the viewing mechanism interprets and enforces them. When you drag an object onto another object, or onto a particular field within an object, it will instantly flash red or green to indicate whether you are permitted to drop it there. Hold it there for a moment and you will be told either what the consequences of that action will be, or why it is not permissible. Right-click on an object to see its pop-up action menu, and it is likely that several of the actions will be greyed-out, indicating that they are not permissible with the object in its current state. Again, hold the mouse there for a moment and you will receive further explanation if you need it. It is also possible to customise the view of the system according to the user's role(s): for a given user, certain object classes, menu actions, attributes or associations may not show up at all. Again, all this is specified within the business object itself.

The idea of auto-generating the user interface from the business model definition is not new, but most of the resulting user interfaces have been hard to use. Critics say that this approach makes life easier for the programmer at the expense of the user. The naked objects approach overcomes such criticism. Naked objects systems won't necessarily score highly on every aspect of usability, but overall they are more usable than most traditional systems, even those where a great deal of attention has been paid to usability. This is because object-oriented user interfaces better match the way that most people think about the tasks they are performing.

The traditional user interface is seen as a way to mediate between the computer's representation of a domain and the user's understanding of it. Much of the effort involved in designing and implementing a business system is spent translating between these different representations, and

ensuring that the various documented versions of both remain synchronised. Naked objects eliminate this notion of mediation. This has the curious consequence that the concept of the user interface effectively disappears. The programmers no longer think about there being a user interface because they do not have to design and code one, and the users no longer think about there being a user interface because they perceive themselves as engaging directly with the business model.

Naked objects give you less control than conventional techniques over the detailed layout, typography and visual style of the interface. However, this can be surprisingly liberating. Many people who have tried the naked objects approach, users and developers alike, praise its 'flexibility' – curious for an approach that removes a familiar form of control. Perhaps this is because conventional user interface design tools encourage design teams to spend too much time and effort worrying about superficial forms of flexibility, whilst failing to see its deeper forms.

The resulting systems are more flexible to future business change, because the business functionality is truly encapsulated by the core business objects. We call this 'strategic agility'. Most attempts at object-oriented systems design unwittingly encourage the separation of process and data, thereby undoing much of the flexibility provided by object-oriented approaches. The resulting systems are also more agile in the day-to-day operations of the business, because the 'direct engagement' style of interaction treats the user as an empowered problem solver rather than a mere process-follower. We call this 'operational agility'.

Naked objects also help to make the development process itself more agile, because they provide a common language for communication between the developer and the customer. This yields strong benefits in situations where the requirements are uncertain.

We will now explore these claims in greater detail.

Naked objects are better able to accommodate future business change

Systems built from naked objects are more agile, in the sense that they can more easily accommodate changes to the business that will be needed in the future, such as changes to product specifications, the organisation chart, internal rules and external regulations, business processes and relationships with other organisations. This is significant because most businesses find that these changes are becoming increasingly frequent: what used to change every decade now changes every year; what used to change every year now changes every month; and so forth. We call this kind of agility ‘strategic agility’.

The most effective of the techniques for building strategic agility into business systems is object-oriented design. Object-orientation was originally invented to support the simulation and modelling of real-world systems: oil refining processes, car suspension systems, the spreading of diseases. Most computer systems, then and now, are designed on the assumption that a stable process will be applied to many different data. Simulation, however, frequently involves working with fixed data, and varying the functionality in order to design a system that behaves in the desired fashion. The designers of the first object-oriented programming language, Simula [3], concluded that to make data and functionality equally changeable, the system should be constructed from units (objects) that were a complete representation of the real-world artefacts being modelled. Thus an object representing the wheel of a truck would know not just the dimensions and engineering characteristics of that wheel, but how to turn, how to bounce, and how to model friction. If the truck’s designer wanted to see the effect of adding another pair of wheels, he could instantiate the wheel object twice and specify the links to other elements, knowing that all the functionality would already be incorporated. This principle is known as ‘behavioural completeness’. This does not mean complete in the sense of 'nothing more could possible be added'. It means complete in the sense that, if there is any functionality in the system concerned with the behaviour of a wheel, it will exist on the wheel object itself and nowhere else.

The characteristics of today’s businesses are closer to those of a simulation and modelling exercise than to the classic notion of a stable process. In recent years a great deal of effort has been directed at making business processes more editable, through simple workflow scripts or complex business process modelling languages. Their success depends not just upon the form of the process script, but upon the nature of the elements being linked together by the script. Most forms of software componentisation draw upon object-oriented concepts, such as ‘encapsulation’, but seem to have lost the most important idea – behavioural completeness. One clue to this state of affairs may lie in the two meanings of the English word ‘encapsulate’. The first meaning – to *enclose* (as in a capsule) – is the one adopted by most forms of componentised software and business process scripting techniques. The components are treated as black boxes, their functionality accessible only through a tightly-defined interface. Object-oriented systems embrace this idea, but they also embrace the second meaning – *to exemplify* (as in ‘this document encapsulates our business strategy’). This is what leads to behavioural completeness.

Many people who purport to be doing object-oriented design seem not to understand this point. As a result, most object modelling exercises result in business objects that are behaviourally poor. The objects are defined principally in terms of their attributes, and relationships. They have methods for ‘getting’ and ‘setting’ (that is, reading and writing) those attributes and relationships, and perhaps some methods for validating the data, but most of the real business functionality is implemented as some kind of process script or controller that sits on top of the objects. These process or task scripts may themselves be implemented using object-inspired

technology, but the separation of data and process is something that object-orientation was originally intended to avoid.

In other cases, the software designers intend to specify behaviourally complete objects, but the design just seems to drift back into a separation of process and data. Usually this is caused by the methodology. Most so-called object-oriented methodologies require that you start the exercise by gathering business requirements in the form of 'use-cases', and then use these to identify candidates for objects. But use-cases have an unfortunate tendency to get turned into procedures or 'controller' objects which end up holding most of the business functionality. The business objects that are subsequently identified as being common to those tasks end up as little more than data structures with some minimal behaviours for checking consistency and applying constraints.

Another important factor that reinforces this trend is the Model-View-Controller or MVC pattern [7]. The intent of MVC is that the business object model should have no knowledge of the user interface: you should be able to change the latter without changing the former. MVC is so well established within object-oriented methodologies that to question it is considered heretical. We suggest, however, that in practice significant amounts of business logic (such as testing that inputs lie within tolerable range) end up in the view and controller objects (which are sometimes combined, anyway, in the so-called Model-View pattern). The net effect is that almost any business change will require alternations to all three categories.

We suggest that not only should the business logic layer know nothing about the presentation layer, but that the presentation layer should also know nothing about the business logic layer. The idea of using the web-browser as the universal client has achieved this in a physical sense: you can update your application on the server without altering the software on the client, and this has significantly reduced the cost of desktop support and maintenance. But in another sense, this has merely pushed the presentation layer into the server: the same problem still exists between the multiple tiers on the server.

Naked objects make it easier to achieve true object-orientation

In the naked objects approach, the viewing mechanism is truly agnostic with respect to the business model. The programmer does not alter the viewing mechanism in the course of developing a system. Indeed, they need not even be aware of the existence of the viewing mechanism. Conversely, the business object definitions are agnostic with respect to the presentation layer. Although the programmer writing the object definitions must adopt a few very simple conventions, there is nothing in any of these conventions (at least for the *Naked Objects* framework) that suggests anything to do with a user interface, let alone a particular style of user interface. It would be quite possible to write alternative viewing mechanisms that produced different user interface styles, or that utilised the different interaction 'gestures' offered by different client platforms (for example, a viewing mechanism for a Palm Pilot). It would even be possible to write a conventional task-oriented user interface that invoked the functionality that it needed from the business objects.

Similarly the naked objects approach avoids the need for application-specific 'controller' objects. With the *Naked Objects* framework there is no way to provide the user with any form of business functionality, except as an object behaviour – in other words, by invoking a method (verb) on a core business object (noun). If you want something done, you have to think about which object the behaviour starts with.

Trying to enforce such a constraint on a paper-based object modelling exercise would prove daunting, except to experienced object modellers. But naked objects are immediately rendered visible and tangible. Specifying the behavioural responsibilities of an object means asking questions such as 'What actions would you expect to see when you right-click on this object?' and 'Where might you want to drop this object, and why?'

For less experienced modellers, this concreteness of naked objects helps to avoid some of the common pitfalls in object modelling. Even for more advanced modellers, the visibility and concreteness of naked objects means that standard object design patterns, previously accessible only to programmers, can be identified and used by all those involved in the business object modelling exercise.

Naked objects empower the user to be a problem-solver

In most core business systems the computer runs the business process, delegating to the user only those tasks that it cannot do for itself. This reduces the scope for error, but it also minimises the user's scope for decision making. In other words, most core business systems are dis-empowering. This is exemplified in the call centre, where every action of the customer service representatives is tightly scripted by a computer system, often designed to minimise the average call length.

Technology is not to blame for this state of affairs. The concept of removing all decision rights from the worker in the interests of efficiency dates back to Frederick Taylor and his 'Principles of Scientific Management'[10]. However, technology has served Taylorism well. The last couple of decades have seen a significant number of technology innovations that make it easier to remove the decision rights from the worker. Paradoxically, many of these innovations were originally sold as doing the opposite. Expert systems, for example, were supposed to make specialised expertise more broadly accessible within an organisation: to turn more people into 'knowledge workers'. In reality, expert systems were mostly deployed to deskill existing jobs. Similarly, workflow systems were initially trumpeted as a form of collaborative technology. The reality is that workflow systems treat the user as just another service on the network to be invoked when the system needs it.

More important than the technology innovations is the context into which they are emerging. As John Seeley Brown points out the business process reengineering movement of the early 1990s has left many organisations unable to see anything that cannot be described in the language of formal processes [2]. Barbara Garson suggests an even 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 centralised 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' [4].

Treating the user as purely a process follower can have negative consequences. We can see this whenever the scripting of the system does not fit something that we want to do. We have all encountered this at the customer interface, when the customer service representative cannot deal effectively with our problem because it does not fit any of the standard scripts (perhaps just because the order in which we 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 their schedules. But when significant disruption (for example, a major storm) occurs, those tools provide very limited support for simulating and then executing live workarounds.

There is a second and more subtle negative consequence. As Brenda Laurel states '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.'[8] Over time this has a subtle but cumulative impact on the user's sense of his own self-worth and motivation.

How do you go about designing a system to empower the user?

Hutchins, Hollan and Norman state: '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.'[6]
The reason why the naked objects approach makes extensive use of drag and drop has nothing to do with efficiency, but is rather because physical gestures aid this sense of direct engagement. Similarly, instead of waiting until the user makes an error and popping up an intrusive error message, the naked objects approach is to prevent the user from making the error. Dragging an object across a series of fields will cause each field to colour red or green to indicate if the object can be dropped there.

Working around constraints embedded into scripted tasks

'Sorry, sir, the system is showing that we have no cars available at your requested time of 3.15. However, let me just try booking it for 3.45pm. Good, that's accepted. Now I'll just confirm the booking. Now, we'll pretend that you've just called up five minutes later and requested to bring forward your pick-up time – the system will allow me to alter the time by up to 30 minutes without re-checking for availability. There – all set!'

Treating the user as a problem solver rather than a process follower does not imply the elimination of all rules and constraints – that would be impracticable for many business activities – but it does entail rethinking how and where they are applied. Most business systems incorporate the rules within the process or task scripts, but this often forces the user to conduct that task one way.

Naked objects embed the necessary rules directly into the objects that the user is manipulating. This makes it impossible to work around the important rules, but localises their impact. Such an approach also makes it easy for the users to specify their own constraints when solving a problem, for example 'this wheel must remain attached to that axle no matter where it moves', or 'the total cost should be kept below $150'.

Another way in which naked objects aid operational agility is by helping to eliminate modality. The oft-told story of an early word processor illustrates how frustrating modality can be for users: type the word 'EDIT' whilst in the wrong mode and it would be interpreted as 'select Everything; Delete it, then Insert the letter T'[5]. A more commonplace form of the problem

occurs in most transactional business systems when the user is forced to complete one task script before initiating another. At the highest level, discrete applications that force the user to switch applications just to invoke a particular function or piece of information are another form of modality.

Modes often cut across the way in which the user wants to think about the problem, or the order in which the information becomes available. It is not possible, or indeed desirable, to eliminate all forms of modality, but naked objects do eliminate many of its most undesirable forms. Any legitimate action is available on any object in any context, giving the user a great deal of freedom in the order in which the elements of a task are fulfilled.

Naked objects also eliminate the concept of discrete applications. Think of a single object model in which different users have access rights to different objects within that model, and/or different methods or attributes of those objects.

Naked objects provide a common language between developers and users

Not only are naked objects systems more agile than conventionally designed systems, but the development process itself is more agile as well. The first key to this agility is the improved communication between user and developer, both during the initial exploration of business requirements and possibilities, and when specifying the exact functionality that is to be delivered.

It has long been claimed that object-orientation bridges the communication gap (sometimes called the ‘semantic gap’) between business and technology, but in practice this has meant bridging the gap between programmers and analysts. Usually, the end-user representation of a system developed using object-oriented techniques is not itself object-oriented. Conversely, even where an object-oriented user interface has been deployed, it does not necessarily mirror the underlying business object model.

Naked objects provide a common language between user and developer. This makes it a lot easier for users to become involved in the design of the system, which is one of the tenets of most modern development methodologies.

In all the projects where we have seen this approach deployed, users have been very comfortable adopting the language of objects. This is not limited to the idea of business objects themselves (such as Customers, Products, Orders, Payments, and so forth) but also with concepts such as ‘instance’, ‘class’ and ‘subclass’, ‘instance method’ and ‘class method’, ‘attribute’ and ‘association’. Very quickly the users start to express their ideas and requests for new functionality in those terms. They ask questions like: ‘Would it be possible to have a method on the Promotion object to visualise the leaflet?’; ‘We need another sub-class of Store to represent our Petrol Filling Stations’; and ‘I want to be able to associate an individual Payment Method (object) with each different Benefit (object), not just with the Scheme (object) overall’. Naked objects bridge the gap because they render the core business objects visible to the users; they are therefore the most obvious things to talk about.

Contrast this with the normal state of affairs, where users specify requirements in terms of alterations to processes: new menu items, new reports, alterations to forms. Someone (maybe the developers, or maybe an intermediary such as a systems analyst) must translate such requests not only into the changes needed to the underlying data and functions, but into all the screens and user interaction controllers that are affected.

The following two case studies describe the application of the Naked objects pattern to business systems at two different organizations: the Irish government's Department of Social and Family Affairs, and Safeway Stores in the UK. The nature of the two applications were very different, as was the technology framework used to implement the naked objects.

Case study 1: Government Benefits Processing

Background

The Department of Social and Family Affairs (DSFA) is the Irish government agency responsible for social security administration, (formerly known as the Department of Social Welfare). It depends heavily upon information technology to fulfill its tasks. Although it manages more than 2000 PCs, the 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 are increasingly expensive to maintain. Currently, there is a separate system for each major type of benefit - Pensions, Unemployment, Child Benefit, Disability etc. Although there is a Central Records System (effectively a customer database), there is much less sharing of both information and functionality than there could be. Many systems have their own separate mechanisms for generating payments, for example.

In 1999 the DSFA conceived a new Service Delivery Model (SDM) that emphasises electronic commerce, organizational agility and customer-responsiveness. The SDM highlighted the need for a complete new architecture for the core systems, that would not only support the specific needs of the new operating model, but would be more adaptable to future, as yet unforeseen, business changes. Early on it was decided that new architecture should be multi-tiered, and object-oriented. The DSFA's IT department had experimented with object-oriented techniques some 3-4 years previously, focusing on the idea that object-orientation could improve development productivity through re-use, but this initiative had produced little in the way of tangible output or results. In retrospect, the management felt that that initiative had been focused much too inwardly on the IT department itself. This time, the motivation for thinking about objects would be to improve business agility.

Early experimentation

The IT management became aware of the author's work on the naked objects pattern in 1999 (although that name had not been coined at the time). The emphasis on designing behaviourally-complete entity objects, and thereby to improve the agility of the resulting systems, meshed well with the DSFA's own objectives. The IT management was also attracted to the visual concreteness of the pattern. They felt that this would make it easier to get non-IT managers more directly involved in the object modeling.

A single one-day workshop was arranged to explore the idea of designing a new system, using the naked objects pattern, that might ultimately replace the entire current application portfolio. In other words the objective was to conceive an enterprise-level business object model, which if exposed to users (each user probably having only having access to a subset of the objects or their behaviours) would fulfil all the DSFA's application requirements. The workshop involved six senior managers from within the DSFA, two representing the business units and four from the IT department. A couple of the participants had a small amount of previous exposure to object-oriented techniques; the others had none. The group therefore depended upon the OO experience of the author who acted as the lead object modeller. However, the group did have very broad knowledge of the department's business at both strategic and operational levels, and this was critical to the success of the workshop.

Following a brief explanation of the naked objects pattern, the group was asked to start naming candidates for possible objects immediately. No explicit process or methodology was followed and no attempt was made to specify use-cases, either formally or informally. The group did not find this task difficult, and within an hour had generated a list of approximately 30 candidate objects. The stated objective of the day was to end up with a list of no more than 10 core business object classes, that between them provided the maximum value (i.e. functionality offered to the user) and versatility (i.e. would be used in a broad range of business tasks).

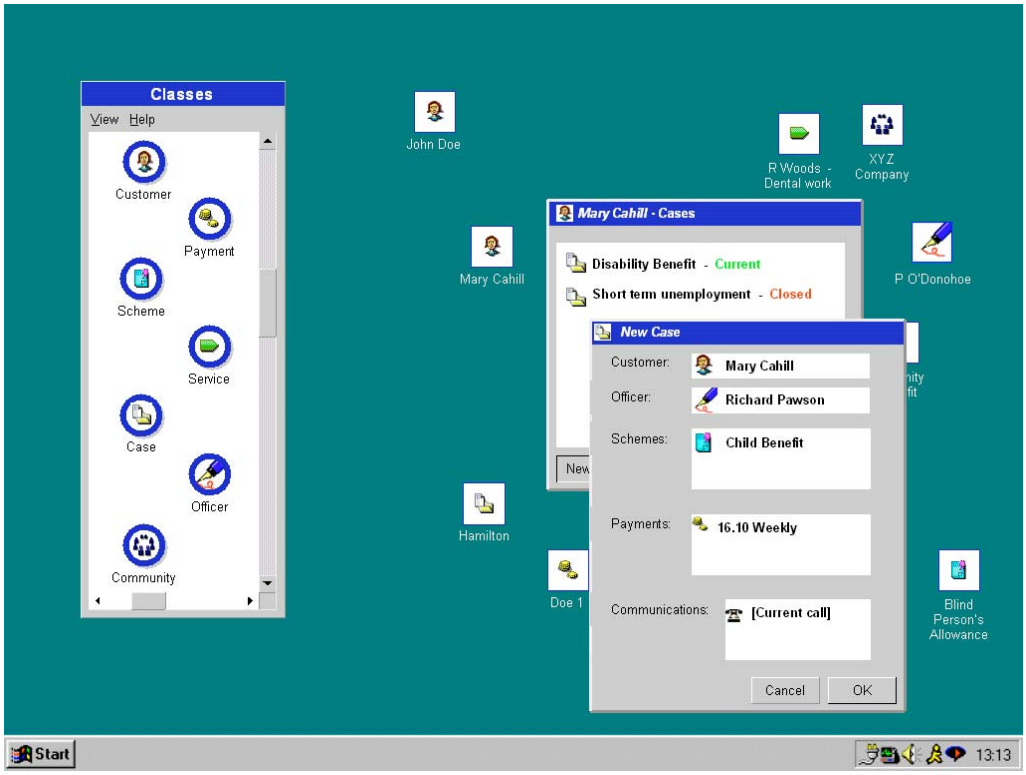
Reducing the list of candidates involved looking for synonyms or overlapping definitions, specialization relationships (that could be modeled through inheritance) and aggregation (i.e. objects that could satisfactorily be treated as elements of other entities rather than entities in their own right). However, by far the most important technique used in reducing the list and defining the responsibilities of the object classes, was visualisation. The group was repeatedly asked to envision a candidate object from the perspective of the user, even to the point of specifying an icon to represent that class of object. In addition the group was asked questions such as:

- *'Why might the user want to point to a particular instance of this object class (i.e. this one, not that one)?'* This was useful for weeding out several suggestions that were not instantiable classes. In some cases they were singleton objects, but more commonly these candidates were really names of services that could be better restated as instantiable entities encapsulating the behaviour (e.g. replacing InterestRateCalculator with an instantiable InterestRate that encapsulated responsibility for calculation).
- '*How would the user distinguish one instance from another?'* In other words: how might the icon be titled?
- *'What other icons would you expect to see if you opened up this icon (i.e. double clicked on it)?'* In other words, what are its associations?
- *'Where might the user drag and drop this icon and what might be their intent in doing so?'*
- *'If the user right-clicked on the icon what actions would they expect to be offered on the pop-up menu?'*

In parallel with these visualization questions, the group was also encouraged to specify the responsibilities of the objects, broadly in the manner described by Wirfs-Brock [11]. In particular they were encouraged to specify responsibilities in the form of 'know-how-to' statements. These were captured in an informal textual form, similar but not identical to Class-Responsibility-Collaborator (CRC) cards [1].

By the end of that one-day workshop, the group had identified six core business object classes (Customer, Officer, Scheme, Case, Communication, and Payment/Service) and defined the key business 'know-how-to' responsibilities of each. It had also 'walked through' a number of very simple operational business scenarios to see how a user might invoke the objects and their responsibilities to fulfill some representative business tasks such as a simple benefits claim.

At the time of this workshop there was no technological framework to implement the naked objects pattern. The only tools available to the group were flip charts and pens. Immediately after the workshop, however, the business object definitions were translated into a crude visual mock-up of a system as used by a DSFA officer handling benefit claims. The mock-up (shown below) was actually just a series of hand-drawn screenshots held as full-screen PowerPoint slides. However, a well-rehearsed demonstration created the impression of icons being dragged around the screen and menus popping up in response to a right mouse-click.



The visual mock-up was demonstrated to a group of senior managers most of whom had not been involved in the one-day modeling workshop. The following responses summarise the impact of the demonstration:

- '*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 will 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 divisional barriers, 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 future commitment to electronic access (via the web, smartcards and telephone call centres), plus a more integrated approach to the systems themselves, would mean that most of the data entry work will eventually 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 the appropriately-trained officers might in future handle all the benefits for one customer. The managers present felt that even this simple mock-up could help to convey to users the message that they are problem solvers, not process followers - because it 'felt' more like using a drawing package rather than a conventional transactional system. The latter was in marked contrast to the approach that had been proposed by some package vendors, which had emphasized using rules-based technology or 'intelligent software agents' to automate as much decision making as possible. Instead, the new mock-up suggested an environment where their natural problem skills would be highly leveraged.

In addition to this positive reaction from the user-representatives, the IT management were also impressed with speed of this exercise compared to all previous attempts at department-wide modeling (whether as objects, data or processes).

It was subsequently decided to trial the naked objects pattern on a replacement for the current Child Benefit Administration system. The Government had indicated possible future changes to Child Benefit that the existing system simply could not be modified to address. Child Benefit is one of the simpler schemes that the DSFA administers and is relatively small in scale: the existing system had just 50 users. Yet it also has much in common with the administration of larger or more complex benefit schemes.

The outline object model was now extended to address the specific needs of Child Benefit Administration. Object responsibilities were refined and new responsibilities identified. New sub-classes and secondary (aggregated) objects were also added. And the whole model was then tested against a number of more detailed business scenarios – use-cases, in effect. In addition the model was also tested against a range of 'strategic' or 'what-if' scenarios that speculated on possible future changes to the DSFA's operating model or even its scope of responsibilities. Those managers involved in this process were impressed at how well the initial model from the one-day workshop stood up to these subsequent demands and tests.

Technology demonstrators

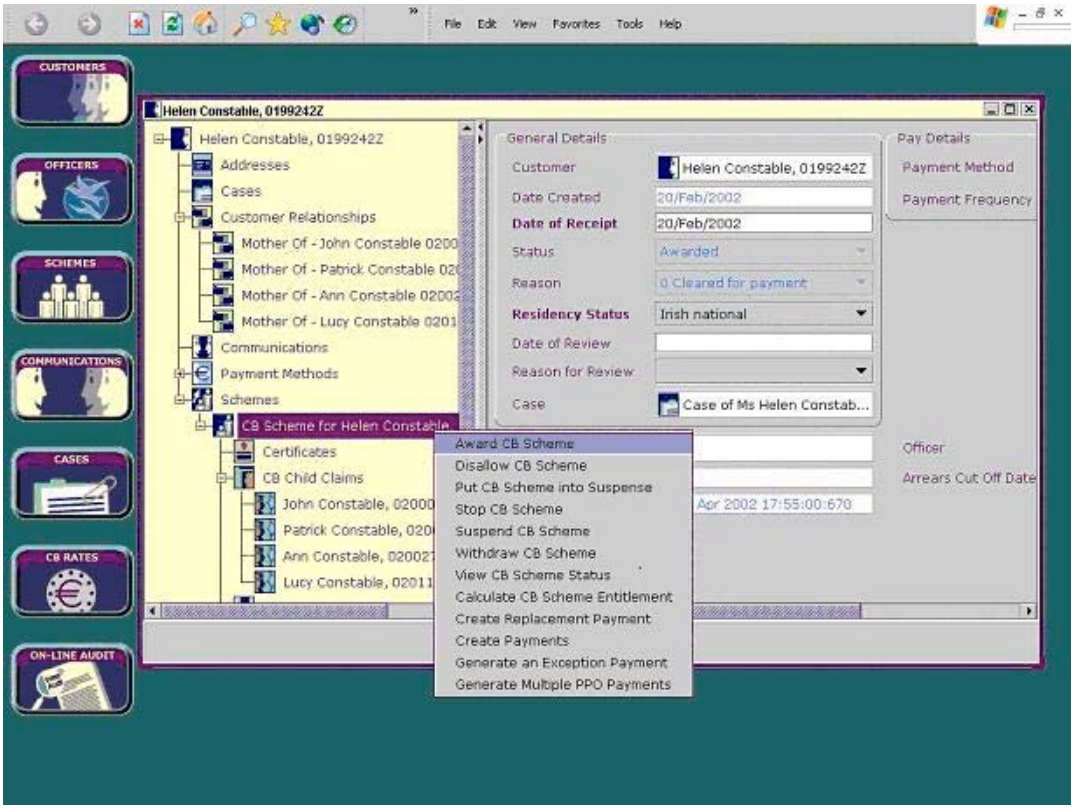
In parallel with this modeling activity the DSFA commissioned three different vendors to build and demonstrate a complete technology framework that would support the naked objects pattern. (The term used by the author at that time, and retained by the DSFA, is an 'Expressive Object Architecture' or EOA). The three demonstrators were based on different distributed object technologies: EJB/J2EE, COM+ and a proprietary object-oriented package respectively. This exercise satisfied the DSFA that the pattern could be successfully implemented, including the requirement that an object-oriented user interface could be 100% auto-generated from the business entity object definitions.

Implementation

At the beginning of 2001 a contract was issued for the full-scale development of the EOA, and the implementation of a subset of the business object model necessary to support the administration of the Child Benefit scheme, sufficient to replace the existing systems.

In the chosen architecture, the business objects are defined in VisualBasic 6.0 supported by COM+ (now scheduled to be upgraded to .Net) services for persistence, transaction monitoring and so forth.

The client side of the system consists of a generic viewing mechanism, written in Java, that will run either as a standalone application from the desktop, or as a Applet within a browser. A typical screen generated by that viewing mechanism is shown in below. Although the graphical design of the elements has changed from original PowerPoint mockup the structure and style of interaction is almost identical.



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 business 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 is used to generate the class icons with their pop-up menus of class methods.

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, wants to create a new association between instances, or to invoke a business method from a pop-up menu, the viewing mechanism will generate a concise XML message back to the appropriate object, specifying the action and any necessary parameters.

Evaluation.

The Child Benefit System went live in November 2002. Despite some fears about performance, this proved to be well within the specified criteria - at least for on-line use. Batch processing (for example to manage the issue of monthly electronic payments, or annual payment voucher books) did generate some performance problems. It had been hoped that batch processing could also be done purely by invoking the behaviours of the entity objects. In the end it was necessary for some of the batch processes to duplicate some of the business functionality in the form of dedicated batch scripts that communicated directly with the data in the persistence layer.

From the users' perspective, the new system represented a radical departure from the system they had known. A substantial effort had to be devoted to training, including a basic introduction to using PC and Windows for some users. However, users responded very positively to the look, and more importantly to the 'feel' of the new system. Many users quite happily adopted the object-oriented terminology that was reflected in the user interface including the idea of classes, instances, collections, and methods. They particularly liked the fact that the new system permits them to address a particular task in a variety of different ways, according to the information that is available to them at the start of the task, rather than being forced to go through a standard procedure. This concept was characterized within the DSFA as 'operational' agility.

The original claims made for adopting the naked objects pattern had included the promise of more flexibility for the user, but it was recognized up front that for purely standard tasks there might be some loss in efficiency. However, one early user of the new system reported that the time it took him to process a straightforward Child Benefit application had fallen from 20 minutes using the old system, to six minutes on the new one, and the time to process a '16+ extension' (for children taking further education) had fallen from five minutes to one minute. This result was a positive surprise to many of those involved in the design.

The naked objects pattern had also promised more 'strategic agility', meaning the ability to accommodate future changes to the business. With the Child Benefit system having only just gone live, it has yet to cope with any significant business changes, so this benefit cannot yet be explicitly evaluated. However, the DSFA has now decided to replace the systems responsible for administering the state pensions with the new architecture: a much bigger project than the Child Benefit system. The outline business object model required for the pensions system has already been completed, and this has identified the need both for new classes and new attributes or methods on the existing business objects. However, the managers responsible for the modeling have reported their delight at the fact that the objects developed in the context of the Child Benefit Administration system will in fact provide by far the bulk of the functionality needed for Pensions administration. Thus, it could be said that the model exhibits a high degree of re-use. However, the DSFA is concerned less with measuring the degree of re-use than with the complementary concept: how easily the model can be extended to accommodate a new requirement.

While this 'Phase 2' modeling was happening, the Irish prime minister announced via the Sunday papers that if re-elected he would introduce the radical new concept of a 'stakeholder pension'. Such announcements are often dreaded by government departments, since they frequently imply massive changes to the existing information systems in a short timescale. By Monday morning, however, the IT manager responsible for the business object model had already determined that if enacted, the new pension would require only very small extensions to the object model.

The DSFA attributes this strategic business agility to the behavioural completeness of the business objects, as directly encouraged through the use of the naked objects pattern. For example, for several of the benefit schemes it administers the DSFA offers the option of payment directly into a bank account or via a book of payment vouchers that can be cashed at a nominated Post Office. In the legacy systems, code responsible for dealing with these two different forms of payment is scattered throughout the systems portfolio. In the new architecture the details of a customer's bank account or local post office are held in an object that implements the interface 'PaymentMethod' and which encapsulates the knowledge of how to interface to the appropriate book-printing or bank-transfer systems. Introduction of any new form of payment, such as a smart-card, would merely require the development of a new

SmartCard object that implements the PaymentMethod interface. That new object could then, by default, be used anywhere that the other payment methods can be used - there is no need to update all the various Scheme objects to tell them of the new option. There are alternative ways of implementing the idea of a shared payment service other than by using behaviourally-complete approaches, but for most of those approaches introducing a new form of payment would require not only a change to the shared payment service, but also to the code of each of the applications or processes that invoked the service.

Case study 2: Retail marketing and promotions

The second case study is in the area of retail marketing and promotions. Safeway Stores is the fourth largest supermarket chain in the UK, with 450 stores ranging from hypermarkets to local convenience outlets. Safeway has developed two exploratory prototypes using an open source Java-based framework called *Naked Objects* [9], which was designed specifically to implement the naked objects pattern. Using the *Naked Objects* framework, each business object class is implemented as a single Java class, inheriting from an abstract class supplied with the framework. At run-time, the framework renders the objects and their classes visible to the user. It uses Java's reflection capability to identify the methods available on any business object (including any public accessors to private fields) and to make these available to the user. It is quite true to say that in writing a business application using the *Naked Objects* framework, the programmer does not need to write a single line of code having to do with the user interface, and indeed is not given any opportunity to do so.

Although many of Safeway's existing systems must be maintained in Cobol, Java is the preferred language for new development. The management of the Java services team became interested in the naked objects pattern, and the *Naked Objects* framework, not initially as a development approach, but rather as a way to train some of the Java developers to think in more object-oriented terms. They felt that more commitment to object-oriented principles would help them see greater benefit from the investment in Java technology and skills. More than 30 developers were trained to use the pattern and framework over the next 3 months - with the majority reporting that it had significantly improved their understanding of object-orientation. Moreover, the experience had created considerable enthusiasm for undertaking a realistic development exercise using the naked objects pattern.

Background

Still under the auspices of training, a candidate project was identified in the area of promotions management, known as 'deal nominations'. Safeway competes 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. To prevent the competition from matching these offers, the set of promotions in each geographic cluster of stores is changed each week.

Implementing these promotions involves managing the supply chain to cope with sudden uplifts in sales volume on the promoted items, communicating the price changes to the point of sale systems in the stores, printing and distributing the promotional flyers as well as in-store banners and shelf-edge labels. Systems exist to manage each of these activities individually, but the coordination of these activities is intensely manual, as is the planning process. Promotions managers are constantly exploring packages of special offers that will attract 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.

Ideally, these managers needed a purpose-designed system to nominate new deals, simulate their roll-out through the store clusters, and then coordinate their execution through the supply chain and price management systems. Previous attempts by the systems department to analyze 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 supply chain management systems. 'Deal nominations' was much more of a problem-solving activity: any particular deal might start with a proposal from a supplier, or it might be initiated to fill a 'hole' in a partly-assembled offering.

Approach

A team consisting of both developers and user representatives from the promotions unit was assembled and given just four weeks to design a proof-of-concept using the *Naked Objects* framework. To put this into perspective, previous attempts had taken much longer than that just to do a paper based requirements-gathering exercise, only to be abandoned because the user's were unconvinced that the resulting document captured what they needed.

The new exercise made no use of that previous work, and started deliberately from scratch. On the first day a couple of hours was spent discussing the dynamics of the business, in order to give the developers some familiarity with the domain. The team got straight down to identifying the set of business objects that would best model the deal nominations area. Around twenty candidates were suggested but by the end of the first day this had been reduced to below ten.

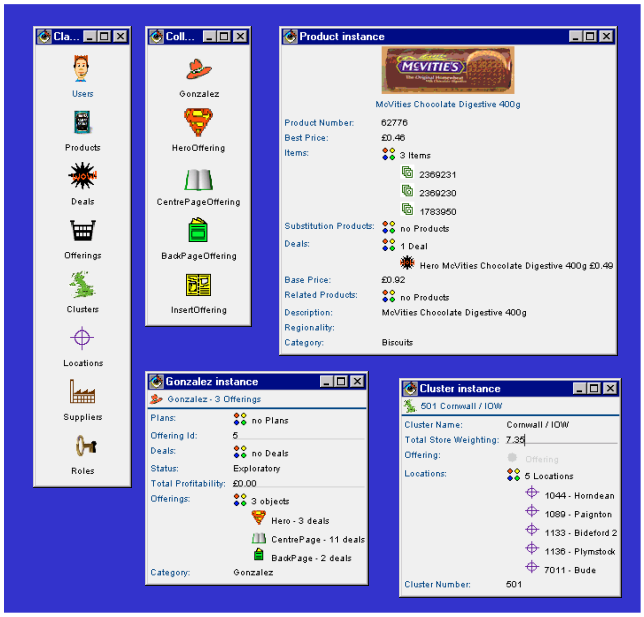
By the second morning the developers were already translating the surviving object candidate descriptions into Java using the *Naked Objects* framework, drawing icons suggested by the business representatives, and assembling some realistic data for Products, Stores and so forth.

The next four weeks followed an iterative pattern. The whole team met formally 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 would be 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 those that involved searching collections of objects or navigating long chains of command) then the developers would work alone, or in pairs.

Throughout this period there was almost constant demand for demonstrations, both from the project's business sponsors, 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 corresponding to specific use-case scenarios. Apart from engaging the sponsors, the demonstrations thus served the important task of continuously validating the object model.

Results

A screenshot from the prototype at the end of the four week exploration is shown below.



Evaluation

All those involved with the exploratory project expressed delight at what had been achieved in the short time. The user representatives said that the prototype had the right ‘feel’: it gave them a great deal of operational flexibility to nominate deals and assemble Offerings in different ways: matching the reality of how they work.

All commented positively on the way in which the naked objects pattern facilitated the dialogue between developers and business representatives: the latter having no difficulty adopting much of the object terminology.

At the end of the exercise the capabilities of the prototype were compared to the prioritized requirements produced by the previous (unsuccessful) attempt - and which had not been consulted during the exercise. It transpired that all the high priority requirements from that original list had been modeled in the new prototype, and much more besides. In fact, when asked to list which of the prototype’s features they would now consider to be the highest priority were the project to proceed to implementation, many of them had not originally been identified as requirements in the original document. For example, an Offering object contains a number of Deals - viewed as a collection of icons. During the first couple of weeks of exploration, someone had asked whether the generic icon representing a Deal could be replaced with an individual icon which was a photograph of the actual Product to be discounted, and whether these icons could be expanded. These were both generic capabilities of the framework. The net result was that any Offering could be viewed as a crude form of the colour flyer that would eventually be printed. The marketing people reported that this early visualization was very helpful in evaluating the attractiveness of the Offering as a whole.

At the time of writing the Deal Nominations system is now awaiting a decision to proceed to full implementation, but there seems no doubt that if it does proceed, the naked objects pattern is recognized by both developers and users as the best way to implement it.

References

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

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

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

4. 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.

5. Hiltzik, M., *Dealers of Lightning - Xerox PARC and the Dawn of the Computer Age*. 1999, New York: HarperCollins.

6. 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.

7. 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.

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

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

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

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

Refactoring

Oliver Whiler, enquiries@refactorit.com
Agris Software - www.refactorit.com

Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. This paper describes the refactoring process in general and some of the benefits of using automated tools to reliably enhance code quality by safely performing refactoring tasks.

Overview

When a system's source code is easily understandable, the system is more maintainable, leading to reduced costs and allowing precious development resources to be used elsewhere. At the same time, if the code is well structured, new requirements can be introduced more efficiently and with fewer problems. These two development tasks, maintenance and enhancement, often conflict since new features, especially those that do not fit cleanly within the original design, result in an increased maintenance effort. The refactoring process aims to reduce this conflict, by aiding non destructive changes to the structure of the source code, in order to increase code clarity and maintainability.

However many developers and managers are hesitant to use refactoring. The most obvious reasons for this are the amount of effort required for even a minor change, and a fear of introducing bugs. Both of these problems can be solved by using an automated refactoring tool

Refactoring – an emerging software development activity

Who needs refactoring?

Software starts on a small scale, and most of it well-designed. Over time, software size and complexity increases, with that bugs creep in, and thus code reliability decreases. Software developers, particularly when they are not the original authors, are finding it increasingly difficult to maintain the code, and even harder to extend. The code base, which in any software company should be a valuable asset, at some point may become a liability.

What is needed to prevent the software from ageing prematurely? Strategically, attention of management and software developers is the most important factor. On the practical side, application of sound development methods will slow this ageing down. However refactoring can reverse this ageing when applied properly, preferably with good software tools that aid in the detection, analysis, and characterisation of the problems, and ultimately allow fixing them

Well trained software developers who are intimately familiar with their code, are often acutely aware of the lurking code ageing problems. However, most developers would be reluctant to make changes to the structure of the code, especially if the changes may take some time. If developers find it easy to apply refactoring operations to their code, they will show less resistance to such restructuring work.

Rewriting a component is often seen as easier by the developer, or at least less confusing. The current source code may have changed over time from the original design, and may not be immediately clear to a developer who is seeing the code for the first time. Alternatively, the original developer may regret certain design decisions, and now believes there is a better way.

However this ignores the fact that the source code has a lot of hidden value. The bug fixes contained in the source code, may not all be documented, however they are very valuable. The component has been previously tested thoroughly in the production environment, and this is not something that should be thrown away. Refactoring retains this hidden value, ensuring the behaviour of the system does not change.

Management are often unwilling to allow changes that will not give any immediate visible benefit, “If it ain't broke, don't fix it”. As well, management may worry about the problem of introducing bugs to a system which has previously been thoroughly tested. This could happen, for example, when manually renaming a method where similarly named methods can be changed as well or methods overridden in a subclass are left with the original name. However, if refactoring operations do not pose a threat of introducing such bugs, management will be less reluctant towards letting refactoring proceed.

What exactly is refactoring?

Refactoring simply means “*improving the design of existing code without changing its observable behaviour*”. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. While refactoring tools make it possible to apply refactoring very easily, its important that the developer understand what the refactoring does and why it will help in this situation e.g. allow reuse of a repetitive block of code.

Each refactoring is a simple process which makes one logical change to the structure of the code. When changing a lot of the code at one time it is possible that bugs were introduced. But when and where these bugs were created is no longer reproducible. If, however, a change is implemented in small steps with tests running after each step, the bug likely surfaces in the test run immediately after introducing it into the system. Then the step could be examined or, after undoing the step, it could be split in even smaller steps which can be applied afterwards.

This is the benefit of comprehensive unit tests in a system, something advocated by Extreme Programming techniques. These tests, give the developers and management confidence that the refactoring has not broken the system, the code behaves the same way as it behaved before.

A refactoring operation proceeds roughly in the following phases:

Action	Questions to ask, actions to take
Detect a problem	Is there a problem? What is the problem?
Characterise the problem	Why is it necessary to change something? What are the benefits? Are there any risks?
Design a solution	What should be the “goal state” of the code? Which code transformation(s) will move the code towards the desired state?
Modify the code	Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before.

Example refactoring:

Rename
A method, variable, class or other Java item has a name that is misleading or confusing. This requires all references, and potentially file locations to be updated. The process of renaming a method may include renaming the method in subclasses as well as clients. On the other hand, renaming a package will also involve moving files and directories, and updating the source control system.

Move Class
A Class is in the wrong package, it should therefore be moved to another package where it fits better. All import statements or fully qualified names referring to the given class need to be updated. The file will also have to be moved and updated in the source control system.

Extract Method
A long method needs to be broken up to enhance readability and maintainability. A section of code with a single logical task (e.g. find a record by id) is replaced with an invocation to a new method. This new method is given suitable parameters, return type and exceptions. By giving the method a clear and descriptive name (findRecordById), the original method becomes simpler to understand as it will read like pseudocode. Extracting the method also allows the method to be reused in other places, which is not possible when it was tangled amongst the larger method. If the extracted section is well chosen, this method may be a natural place to change the behaviour of the class through subclassing, rather than a copy and paste of the existing method before making changes.

Extract Superclass
An existing class provides functionality that needs to be modified in some way. An abstract class is introduced as the parent of the current class, and then common behaviour is “pulled up” into this new parent. Clients of the existing class are changed to reference the new parent class, allowing alternative implementations (polymorphism). Any methods which are common to the concrete classes are “pulled up” with definitions, while those that will vary in subclasses are left abstract. As well as aiding in efficient code re-use, it also allows new subclasses to be created and used without changing the client classes.

Replace Conditional with Polymorphism
Methods in a class currently check some value (if or switch statement) in order to decide the right action to perform. One trivial example is a class that draws a shape, which is defined by a width and type (circle or square). The code quickly becomes confusing as the same if or switch statements are repeated throughout the class, i.e. in methods that calculate the area or perimeter of the shape. By using polymorphism, the shape specific behaviour can be offloaded to subclasses, simplifying the code. This has the added benefit of allowing other subclasses, e.g. rectangle or star, to be introduced without extensive code changes.

With each problem above a more or less obvious solution has been stated, too. However, it is clear to every experienced software developer that there are more complicated code problems, for which simple solutions can not so easily be presented.

Obviously, a software developer will usually apply refactoring successfully only, if he/she knows how the software should look like in the end. In other words, before trying to refactor some code, one needs to familiarise oneself with the common object oriented design patterns and refactoring (see Gamma et al. 1994; Grand 1998).

When should one consider refactoring?

Ideally, refactoring would be part of a continuing quality improvement process. In other words, refactoring would be seamlessly interwoven with other day-to-day activities of every software developer.

Refactoring may be useful, when a bug has surfaced and the problem needs to be fixed or the code needs to be extended. Refactoring at the same time as maintenance or adding new features, also makes management and developers more likely to allow it, since it will not require an extra phase of testing. If the developer in charge finds it difficult to understand the code, he/she will (hopefully) ask questions, and begin to document the incomprehensible code. The phrases he/she coins may be a good starting point for the names of new methods or classes.

Often, however, schedule pressures do not permit to implement a clean solution right away. A feature may have to be added in a hurry, a bug patched rather than fixed. In these cases, the code in question should be marked with a FIXME note, in order to be reworked, when time permits. Such circumstances call not for individual refactoring, but for a whole refactoring project. When the time has come to address the accumulated problems, a scan for FIXMEs, TODOs, etc. over the code base will return all the trouble spots for review. They can then be refactored according to priority.

What are the benefits of refactoring?

Carrying out a few refactoring operations before or during code debugging may have immediate benefits. Often it becomes easier to spot the bug location. So *time is saved*, while at the same time the quality of the code is enhanced. Well-structured code is also less error-prone when it comes to extending it.

Kent Beck [Fowler, p.60] states that refactoring adds to the value of any program that has at least one of the following shortcomings:

- Programs that are hard to read are hard to modify.
- Programs that have duplicate logic are hard to modify
- Programs that need additional behaviour that requires you to change running code are hard to modify.
- Programs with complex conditional logic are hard to modify.

Summarising, while sometimes there are immediate benefits to be reaped from refactoring, the real benefits normally come in the *long term*. They consist in substantially reduced time that developers spend on debugging and maintenance work, as well as in improved extensibility and robustness of the code. In addition, code duplication is reduced, and code re-use fostered. Overall maintenance and development cost should come down, and the speed of the team to react to changing needs should improve.

What are the concerns, particularly of management?

If there are so many usually undisputed benefits from well-structured, comprehensible code, and if refactoring leads from chaotic, badly structured, error-prone code to well-designed code by performing a serious of small, controlled operations, why then is not every software developer engaged in refactoring?

Software developers are often reluctant, because some refactorings are simply tedious. And there is no visible external benefit for all the labour put in.

Also management may be at fault. As long as management only rewards externally visible code properties such as functionality and performance, but neglects to pay attention to the code's inner quality, e.g. via code review or by failing to set coding standards, it is only to blame itself, when software developers are reluctant to invest some of their precious time on refactoring operations.

Then there is the serious risk of breaking the code through refactoring operations. And if e.g. file-names change, traceability of modifications is likely to become an issue, too.

Even if the software developer, is keen to refactor some badly structured code. His/her manager, however, may have quite a different perspective, and may oppose any attempt to modify working code. These concerns cannot simply be ignored, but must be suitably addressed by both developers and management.

Why use an automated tool?

When doing refactoring the externally observable behaviour must be guaranteed to stay the same. If refactorings are carried out manually, one needs to frequently rebuild the system and run tests. Manual refactoring is therefore really practical only, when the following conditions hold:

1. The system, of which the refactored code is a part, can be rebuilt *quickly.*
2. There are automated "regression" tests that can be frequently run.

This situation is not very common, meaning that the applications of refactoring are limited. This situation is becoming more common, particularly as more people use XP (Extreme Programming) development methods.

Another hindrance is that many of these refactoring are tedious. Potentially requiring hours of precious development time, as the change is made and then thoroughly checked. Not many programmers would enjoy the task of renaming a method in a large code base. A simple search and replace will potentially find extra results. So each replacement must be examined by the programmer. However there is no great intelligence to the operation, all that is wanted is to rename any use of a method on a given class or its subclasses. A refactoring tool therefore can save hours of work. Even more importantly give confidence that the correct changes were made.

The speed of automated tools has another benefit, which is shown in team development environments. A developer is much less likely to perform refactoring operations if the source code involved is under the responsibility of other developers as well. However by using an automated tool, the refactorings can be completed quickly so that other developers are not held up waiting to make there changes when the refactoring is completed, or worse making their changes on the old code at the same time as the refactoring operation. This ensures that even when the responsibility for a section of code is shared, developers will not reach a stalemate, where none of the developers make the required changes.

Integration with the developers chosen IDE also bring many benefits. Firstly having the tools at hand, means that developers can more easily refactor. They do not have to switch between development and refactoring modes, and can instead see it as part of their normal development cycle. Secondly IDE features such as Source Control Integration can reduce the effort in

refactorings such as move class or rename package.

Conclusion

Refactoring is a well defined process that improves the quality of systems and allows developers to repair code that is becoming hard to maintain, without throwing away the existing source code and starting again. By careful application of refactorings the system's behaviour will remain the same, but return to a well structured design.

The use of automated refactoring tools, makes it more likely that the developer will perform the necessary refactorings, since the tools are much quicker and reduce the chance of introducing bugs.

How to Select a QA Collaboration Tool

Margaret Fross, mfross@prototest.com
Proto Test - www.prototest.com

Abstract

This paper examines the methods for identifying and choosing a QA collaboration tool, which can serve to support and reinforce process for an organization.

The following points will be addressed:

- Introduction - increasing importance of quality and what tools can facilitate quality
- Definition of the term QA collaboration tool - what is it and what can it do for you
- How to select a QA collaboration tool - key components of a good QA collaboration tool

Introduction

Because of customer expectations and some recent high-profile debacles, the software industry has begun to shift its focus toward improving software quality. For example, companies such as JD Edwards, Microsoft, and Key Bank were recently featured in an InformationWeek.com article touting “Quality First”

Of course time to market is still critical, but there is somewhat less willingness to sacrifice quality just to push software out the door.

When organizations turn their focus to quality, they can begin by asking themselves a few questions:

- What parts of the organization have a stake in quality?
- What key quality assurance components are missing from our current development methods?
- What tools can be adopted and used by all members within the organization to improve quality?

The answers are fairly simple. All members of an organization have a stake in quality. This includes people selling, answering phones, developing applications, testing applications, marketing, and managing at an upper executive level. Anyone who may have contact at any time with the product or the customer has a stake in quality.

Next, organizations need to evaluate what goes on—from soup to nuts—when a new product concept or release comes up, or is planned. This is the organization’s process to get from the drawing board to the marketplace.

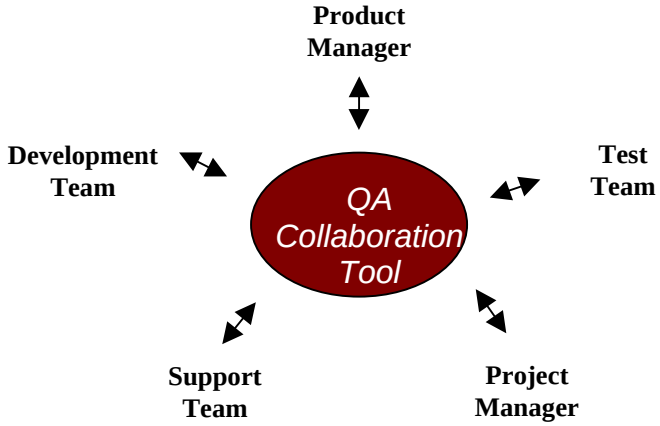
Ultimately, when talking about tools to unite all components of an organization to support its quality objectives, a good place to start is collaboration tools.

What is a QA Collaboration Tool?

A QA collaboration tool encompasses key aspects of the software quality process, providing many functions such as requirements, defect, and test case management in one easy-to-use tool. The purpose of a QA collaboration tool is to bring team members together, providing a central location for the functions above as well as provide a communication forum, document storage, and shared appointment creation. It can serve to alleviate many challenges faced by organizations, including:

Process: Lack of a defined process leads to chaotic unpredictable development. A defined process understood and used by all team members leads to repeatability and predictability. Project estimators can look at past projects to provide time estimate for future projects, predict defect rates, and make better decisions about resource needs.

Communication: Lack of communication leads to misunderstood requirements, missed deadlines, false status updates and general ignorance in the organization. This results in extra work for all team members. Cross-functional collaboration provides a forum for all team members, regardless of location or department, to communicate effectively, keeping everyone on the same page. This shared knowledge leads to a more coordinated effort resulting in on-time and on-budget delivery of high-quality software.



Traceability: Lack of a robust report tool leads to confusion among team members and upper management. Status reports are a critical component of any project. They serve to alert managers to staffing needs, development issues, and whether the project is on track or not. The ability to pull up accurate, real-time status reports at any given time allows decision makers to take action on items. Managers are better able to make informed decisions and mitigate risks. The team members in charge of delivering work items are empowered to take more ownership of their specific piece of the pie, giving them defined goals to shoot for and providing a better sense of accomplishment.

The benefits described above are just a few of the many organizations will experience should they choose to adopt a QA tool promoting collaboration.

What Constitutes a Good QA Collaboration Tool?

When selecting a QA Collaboration Tool, as with any tool, you must first assess the needs of the organization versus the budget and time that is allotted to bring in a new tool. Factors to consider are:

Cost: Examine the cost of each tool by looking at various licensing schemes (e.g., named users or concurrent users). Take into consideration any additional fees (e.g., yearly or other support fees). Identify any costs associated with implementation, conversion of existing data, staff training, and hardware.

Ease of use: Too many companies have "shelf-ware" (software collecting dust on shelves) because the tools purchased proved to be too difficult to implement or were too time-consuming to maintain. This is a tremendous waste of money and resources, remembering the time that originally went into selecting and attempting to implement the tool. Look for a tool that will fit well with what you may already be using. Pay particular attention to any import/export features that may prove useful when moving data from old systems to new systems. Good tools are ones in which an administrator can make changes on the fly, that don't require specialized training or development skills, and are intuitive to your target user. It cannot be stressed enough: the easier you can implement a tool, the more likely you are to effectively use the tool.

Reliability: Will the tool support the user activity you predict? Can all potential users adequately perform their job functions free from frustration? Is there an uptime or availability guarantee with the tool? Does the vendor discuss the frequency with which updates are sent and will those updates necessitate down time for installation? These are important points to consider when selecting a tool.

Support: Verify that the vendor provides support. If you are paying support fees or maintenance fees, what are you getting for those fees? Look at what type of support the vendor provides. Questions to ask are: what is the average call back time for phone support and how fast are e-mail inquiries resolved? What is the charge or availability for on-site support and training?

Now let's look at the **components that make up a QA collaboration tool**. Remember that the more functions you can get from one tool the easier the tool will be for your users, because they will not have to go into multiple applications to perform different functions and to get information. Desired components are:

Scheduling: A place where meetings and reminders can be created and added to team member's calendars; project deadlines can also be displayed here.

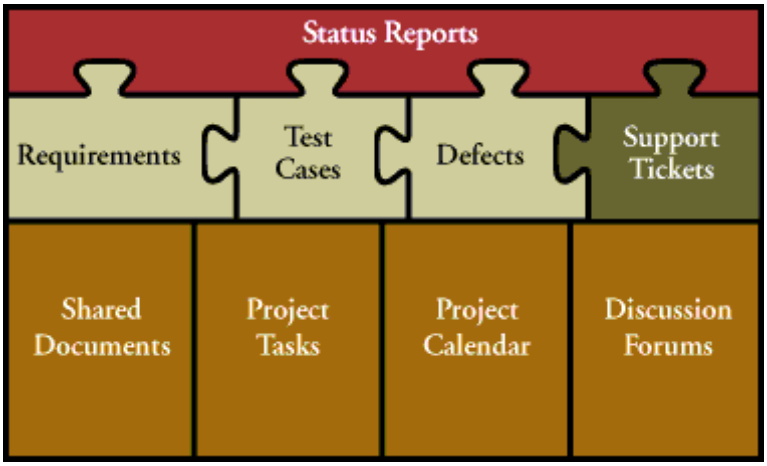
To Do List: All projects have items that do not fit into obvious categories like Requirements and Test Case management and do not require other team members. A To Do List is a place where individuals can add personal items (such as reminders to provide a status update).

Project Tasks: Every project has deliverables and milestones. Deliverables include deliverables from the project and deliverables that are necessary to complete the project. Milestones are important because they provide goals for the group as well as a sense of accomplishment when those goals are reached. Milestones provide a simple way to tell at a glance if a project is on schedule. Having a place where tasks like deliverables and milestones can be stored, edited, and viewed by all team members promotes the accountability and traceability aspects of a solid quality assurance process.

Requirements Management: Successful projects start with clearly defined and agreed-upon requirements. One way to ensure you get good requirements is to make sure everyone is providing the same information for every requirement. That's where a

function for requirements management comes in. Users have a format with specific fields in which to enter requirement data. Requirements are then assigned directly to team members. A central repository where users can add and edit requirements for specific projects provides a way to streamline and promote the requirements definition process. The history of that particular requirement is also tracked, which can help when inspecting requirements and comparing how a requirement evolved to its current state.

Test Case Management: Test cases should tie back to specific requirements. Using the same tool for both requirements and test case management instantly provides that traceability without impacting the user. Test cases should also be as detailed as possible. Use of a tool for test case development helps ensure the users are all adding critical details when writing their test case. Test cases can then be assigned to specific team members for maintenance and execution. With some tools, changing the status of a test case to "Failed" will automatically generate a Defect record. As with requirements, test cases will need frequent editing and the change history must be tracked.



Defect Management: Defects are typically filed when a test case fails. By selecting a tool with integrated components users will be able to trace their defects back to specific test cases that trace back to specific requirements. Managing defects in this way assists team members with risk mitigation and analysis of likely failure points in the application. Defect management tools also provide a place to track support issues and enhancement requests. Use of a standard form for reporting issues guarantees several thing: easy decision making when setting work priorities, faster resolution of items, and a faster turn around time once the item reaches the test team.

Reporting: Decision makers within any organization rely on accurate, real-time information upon which to base their decisions. This is where reporting is crucial. Look for a tool that allows users to create custom reports that they can then generate at will. A good reporting function facilitates easy distribution, resulting in information sharing. With robust reporting tools, organization members—particularly upper-management—will be able to obtain accurate status updates when they most need them. This reduces the amount of time a project or team lead spends providing status to different audiences. The reporting function also comes in handy when estimating future projects. Timelines and staffing needs are predicted more accurately when looking at specific data from past projects.

Document Storage: Over the course of a project many documents are created (design specifications, requirements matrices, test plans, test reports). Using a central repository for all documents created during the life of a project ensures that all team members can access the information they need to complete their tasks. Documents stored in a central location are easier to update and maintain than those stored on local hard drives or scattered around in different network locations.

Open Communication Forum: With the influx of telecommuting, companies have a difficult time getting team members together. Team members not working out of a central office often feel left out of the loop. Creating an open communication forum for both local and remote team members will open the lines of communication. Remote team members will feel more part of the team and less isolated. Team members will be able to easily communicate when customer needs change, when technical questions come up, and when project timelines shift. This promotes better decision-making and also provides an archive of what information was available when decisions were made. Remember that shared knowledge leads to more coordinated and successful efforts.

If you examine the functions detailed above and compare them to the process components of the Software Development Life Cycle you will see that the functions provide a vehicle to promote a stable, predictable quality process.

Additional Components and Considerations

Other than the core features described above, what else can you look for in a collaboration tool?

Look for a tool that supports some form of **customization**. You will want to be able to add your own fields as well as delete existing fields from the forms used to enter requirements, test cases, and defects. This helps you adapt the tool to your organizational needs.

Don't forget **security**. Most companies try to avoid airing their dirty laundry to clients (i.e. defects). If you find yourself needing to allow clients to view, modify, or add requirements, you probably do not want them to view other things—especially the defect list. Being able to add, delete, and modify users (and their access permissions) and change security configurations is a basic function that should be present in any tool you consider.

Any tool you look at seriously should be robust, expandable, and have no limits regarding the **number of projects** you can create. Look for a tool that will allow you to easily create and manage multiple projects simultaneously.

A good **Help** system can make a big difference in adoption and usage of any QA collaboration tool. This includes online help, documentation (including training and installation documents), and vendor support. All team members should be thoroughly trained on the product prior to usage and understand the chain of events for resolution of a question or support item. This can save your team members a lot of stress and aggravation and they will be more willing to accept the changes the tool promotes.

What about **related products**? Does the vendor have any other products in its suite that work with the tool to further enhance the collaborative nature of the tool? For instance, what about a trouble-ticket function? Something that your clients can access to log defects directly without allowing them to see existing defects filed internally or by other clients? Defect management can end up being time-consuming and frustrating if you have to monitor two different systems

depending on who filed the defect. Other peripheral components to consider are e-mail and database functions.

Select a tool that will work well with your e-mail system (since typically users are notified when items are assigned to them via e-mail).

Verify whether or not existing data can be imported into the tool you are evaluating. Examine the relative ease of getting data into the system, and of unloading the data if necessary.

Finally, do not purchase any tool without first receiving a **product demonstration**. Participate in a multi-week trial and try to start implementing the tool with your data. You would never purchase a car before driving it; purchasing a tool is no different.

Conclusion

Remember that process drives quality. A poorly defined or absent process is a sure-fire way to miss your product targets. Products lacking in quality will have a short life. The use of a QA collaboration tool can serve to reinforce a viable QA process.

A QA collaboration tool is not just pertinent to testing teams. Anyone who has a stake in quality can benefit from such a tool. Collaboration translates to shared knowledge, which in turn is beneficial for any team working under deadlines. Consolidating many functions already in use into one tool saves time, money, and promotes a common process.

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig; e-mail franco@martinig.ch ISSN 1023-4918
Free subscription: www.methodsandtools.com